ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Система автоматичного тестування на основі мови програмування Python»

на здобуття освітнього ступеня магістра
зі спеціальності 122 Комп'ютерні науки
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні науки
(назва)

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

_____ Гутнік Олексій
(підпис) *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти Гутнік Олексій
група КНДМ-63

Керівник: Юрій КАТКОВ
науковий ступінь, вчене звання д.т.н., професор

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

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук

Ступінь вищої освіти Магістр

Спеціальність 122 Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедрою Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
«_____» __ _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гутніку Олексію Михайловичу
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Система автоматичного тестування на основі мови програмування Python»
керівник кваліфікаційної роботи Юрій КАТКОВ д.т.н., професор,
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» 10.2024р. №320
2. Строк подання кваліфікаційної роботи «15» грудня 2024р.
3. Вихідні дані до кваліфікаційної роботи:
3.1. Науково-технічна література з питань, пов'язаних із дослідженням систем автоматичного тестування на основі мови програмування Python;
3.2. Практичний досвід застосування систем автоматичного тестування на основі мови програмування Python.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
4.1 Аналіз існуючих систем автоматизованого тестування на основі мови програмування Python, включаючи їх функціональні можливості та переваги.
4.2 Дослідження методів і технологій автоматизованого тестування, застосованих у різних галузях програмної інженерії, з акцентом на ефективність та результативність.

4.3 Розробка практичних рекомендацій щодо впровадження систем автоматизованого тестування в процес розробки програмного забезпечення, а також обґрунтування вибору інструментів та методів для оптимізації тестування.

5. Перелік графічного матеріалу: *презентація*

1. Тема дипломної роботи
2. Мета, об’єкт, предмет дипломної роботи
3. Постановка завдання дипломної роботи
4. Результати аналізу існуючих систем автоматизованого тестування на основі мови програмування Python.
5. Результати дослідження ефективності застосування автоматизованого тестування в процесі розробки програмного забезпечення.
6. Практичні рекомендації щодо впровадження автоматизованих систем тестування у розробку програмного забезпечення, включаючи обґрунтування вибору методів і інструментів.
7. Висновки.
8. Апробації дослідження

6. Дата видачі завдання«10» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури по темі «Автоматизоване тестування програмного забезпечення на основі мови програмування Python»	10.09-05.10.24	Виконано
2	Аналіз специфіки і характеристик основ систем автоматичного тестування	05.10-12.10.24	Виконано
3	Дослідження методів і технологій автоматизованого тестування	13.11-19.11.24	Виконано
4	Моделювання системи автоматизованого тестування на базі Python	20.11-25.11.24	Виконано
5	Написання основних розділів кваліфікаційної роботи	27.11-03.12.24	Виконано
6	Розробка обов’язкових матеріалів (презентація, графіки тощо)	04.12-10.12.24	Виконано
7	Попередній захист роботи.	11.12-15.12.24	Виконано
8	Пред'явлення роботи в деканат.	15.12-29.12.24	Виконано

Здобувач вищої освіти _____ Гутнік Олексій
(підпис) *(Ім’я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Юрій КАТКОВ
(підпис) *(Ім’я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина магістерської роботи: 101 с., 4 табл., 16 рис., 41 джерел.

Наукове завдання – обґрунтувати доцільність застосування сучасних підходів до автоматизованого тестування, вибіру оптимальних інструментів технологій Python, а також створення системи, здатної автоматично виконувати тестування різних компонентів програмного забезпечення для підвищення його якості та зменшення трудомісткості тестувального процесу..

Мета роботи: Підвищення ефективності застосування сучасних підходів до автоматизованого тестування, вибіру оптимальних інструментів технологій Python.

Об'єкт дослідження: Процес застосування автоматизованого тестування програмного забезпечення..

Предмет дослідження: Методи та інструменти автоматизованого тестування, зокрема на базі мови програмування Python..

Метод дослідження. Метод дослідження заснований на відомих методах системного підходу, а саме: аналітично-розрахунковий, теоретичного аналізу з використанням моделювання, реконструювання і типологізації на основі аналізу емпіричних даних.

Короткий зміст роботи: В роботі проведено обґрунтувати застосування сучасних підходів до автоматизованого тестування, вибіру оптимальних інструментів технологій Python, а також створення системи, здатної автоматично виконувати тестування різних компонентів програмного забезпечення для підвищення його якості та зменшення трудомісткості тестувального процесу. .

Ключові слова: автоматизоване тестування, тестування програмного забезпечення, автоматизація процесів.

ABSTRACT

Text part of the master's thesis: 101 p., 4 tables, 16 figures, 41 sources.

Scientific task – is to substantiate the feasibility of using modern approaches to automated testing, choosing optimal tools of Python technologies, as well as creating a system capable of automatically testing various software components to improve its quality and reduce the complexity of the testing process.

Purpose of the work: Increasing the effectiveness of using modern approaches to automated testing, choosing optimal tools of Python technologies.

Object of research: The process of using automated software testing.

Subject of research: Methods and tools of automated testing, in particular based on the Python programming language.

Research method. The research method is based on well-known methods of the system approach, namely: analytical and computational, theoretical analysis using modeling, reconstruction and typology based on the analysis of empirical data.

Summary of the work: The work substantiates the application of modern approaches to automated testing, the selection of optimal Python technology tools, as well as the creation of a system capable of automatically testing various software components to improve its quality and reduce the complexity of the testing process. .

Keywords: automated testing, software testing, process automation.

ЗМІСТ

Ст.

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ

API — Application Programming Interface (інтерфейс прикладного програмування).

TDD — Test-Driven Development (розробка, керована тестуванням).

BDD — Behavior-Driven Development (розробка, керована поведінкою).

CI/CD — Continuous Integration/Continuous Deployment (безперервна інтеграція/безперервне розгортання).

JSON — JavaScript Object Notation (формат обміну даними).

IDE — Integrated Development Environment (інтегроване середовище розробки).

SUT (System Under Test) — система, що тестується.

Test Case — набір умов і дій для перевірки певного аспекту системи.

Test Suite — група тестів, об’єднаних за певним критерієм.

Автоматизоване тестування — процес виконання тестів за допомогою програмних засобів для перевірки якості програмного забезпечення.

Юніт-тестування (Unit Testing) — тестування окремих модулів чи компонентів програми.

Інтеграційне тестування (Integration Testing) — тестування взаємодії між компонентами програми.

Фреймворк — набір бібліотек та інструментів для спрощення виконання певних завдань (наприклад, pytest).

Тестове покриття (Test Coverage) — показник, який демонструє, яку частину коду охоплено тестами.

ВСТУП

В атестаційній роботі магістра розглядається актуальна проблема застосування автоматизованого тестування програмного забезпечення з використанням мови програмування Python.

Обґрунтування вибору теми та її актуальність. В умовах сучасного програмного забезпечення, де швидкість розробки і якість продукту є критично важливими, автоматизоване тестування стає необхідним етапом в життєвому циклі програмного продукту. Таким чином, створення автоматизованої системи тестування є актуальним та своєчасним.

Формулювання завдання дослідження. Розглядається проблема застосування автоматизованого тестування для різних типів програмних продуктів. Треба визначити, в якому вигляді автоматизоване тестування може бути інтегроване у процес розробки, а також які методи і інструменти є найбільш ефективними для реалізації цієї системи.

Ступінь вивченої проблеми. Автоматизоване тестування є активно досліджуваною темою в сфері програмної інженерії, зокрема у контексті використання різних мов програмування та тестувальних фреймворків. Однак, незважаючи на значну кількість наукових робіт, дослідження щодо специфіки застосування Python для автоматизації тестування залишаються недостатньо вивченими, що підкреслює необхідність подальшого аналізу та розробки ефективних рішень у цій області.

Об'єкт дослідження: Процес застосування автоматизованого тестування програмного забезпечення.

Предмет дослідження: Методи та інструменти автоматизованого тестування, зокрема на базі мови програмування Python.

Мета роботи: Підвищення ефективності застосування сучасних підходів до автоматизованого тестування, вибіру оптимальних інструментів технологій Python.

Метод дослідження. Метод дослідження заснований на відомих методах системного підходу, а саме: аналітично-розрахунковий, теоретичного аналізу з використанням моделювання, реконструювання і типологізації на основі аналізу емпіричних даних.

Завдання роботи:

1. Провести аналіз передумов щодо впровадження автоматизованого тестування в процес розробки програмного забезпечення.

2. Дослідити критичні аспекти під час впровадження автоматизованого тестування, включаючи виклики та можливості.

3. Розробити практичні рекомендації щодо впровадження автоматизованого тестування у різних типах програмних продуктів.

Результати дослідження. Підвищення ефективності застосування автоматизованого тестування призводить до скорочення часу на випробування та покращення якості програмного забезпечення.

Наукова новизна. Наукова новизна цього дослідження полягає в розгляді специфіки застосування мови програмування Python для автоматизації тестування та розробки нових підходів до інтеграції тестування в життєвий цикл розробки програмного забезпечення.

Практична значущість результатів дослідження. Практична значущість полягає в розробці практичних рекомендацій щодо застосування сучасних інтелектуальних систем автоматизованого тестування. Враховуючи потреби сучасного автомобілебудування, таке завдання є актуальним та своєчасним.

Апробація результатів надається в 1 статті, 1 тезах в рецензованих наукових журналах і виданнях.

Стаття:

Гутнік О.М. Сучасні підходи до вибору оптимальних інструментів і технологій Python для автоматизованого тестування// Наукові записки Державного університету інформаційно-комунікаційних технологій №2, 2024, Подано до друку. https://journals.dut.edu.ua/index.php/sciencenotes/issue/view/186

В тезисах:

Гутнік О.М. ПРОБЛЕМИ РОЗВІТКУ СИСТЕМ АВТОМАТИЧНОГО ТЕСТУВАННЯ НА ОСНОВІ МОВИ ПРОГРАМУВАННЯ PYTHON // Науково-технічна конференція «Сучасні досягнення компанії Hewlett Packard Enterprise в галузі іт та нові можливості їх вивчення і застосування». Державний університет інформаційно-комунікаційних технологій. Збірник тез 14 грудня 2024. – К.: ДУІКТ, Подано до друку. https://duikt.edu.ua/ua/2661-konferencii-2024-roku-konferencii-ta-seminari

1 АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ТА ІНСТРУМЕНТІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

1.1 Поняття підходів та інструментів автоматизованого тестування

Автоматизоване тестування стає ключовою складовою процесу розробки програмного забезпечення, оскільки допомагає підвищити ефективність, зменшити витрати часу та забезпечити якість продуктів (Рис.1.1).

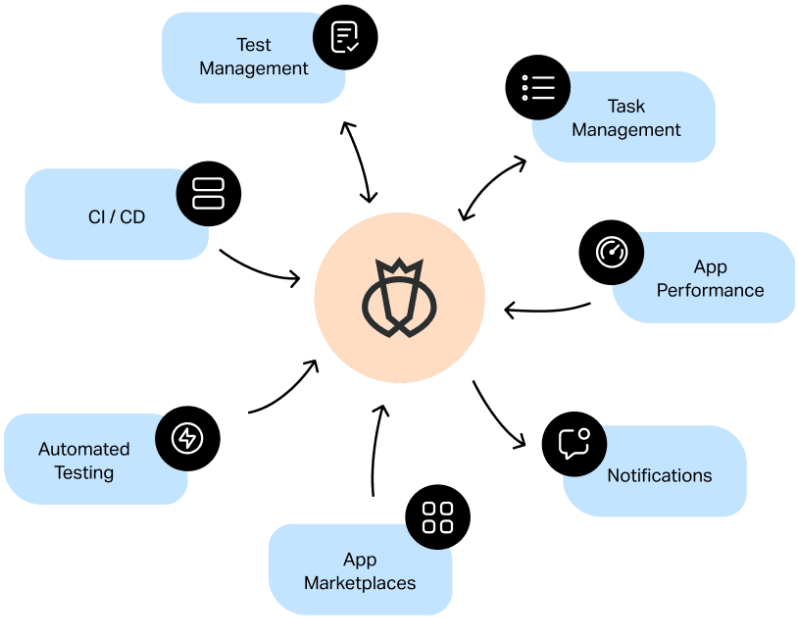


Рисунок 1.1 - Інструменти автоматизованого тестування програмного забезпечення[1]

Нижче наведено аналіз сучасних підходів та інструментів автоматизованого тестування.

Розглянемо сучасні підходи до автоматизованого тестування [6, 7, 8]:

1. Модульне тестування (Unit Testing) : Тестує окремі компоненти або модулі системи. Популярні фреймворки: JUnit (Java), pytest (Python), NUnit (C#). Використовується для забезпечення коректності базових функцій (Рис.1.2).

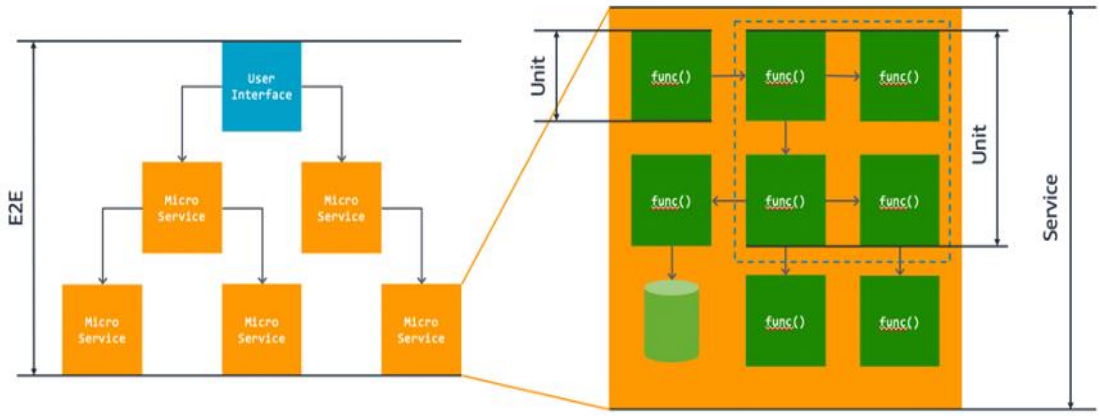


Рисунок 1.2 – Ілюстрація модульного тестування [2]

2. **Інтеграційне тестування (Integration Testing)**. Перевіряє взаємодію між компонентами. Інструменти: Postman (API тестування), Spring Test (Java).(Рис.1.3)

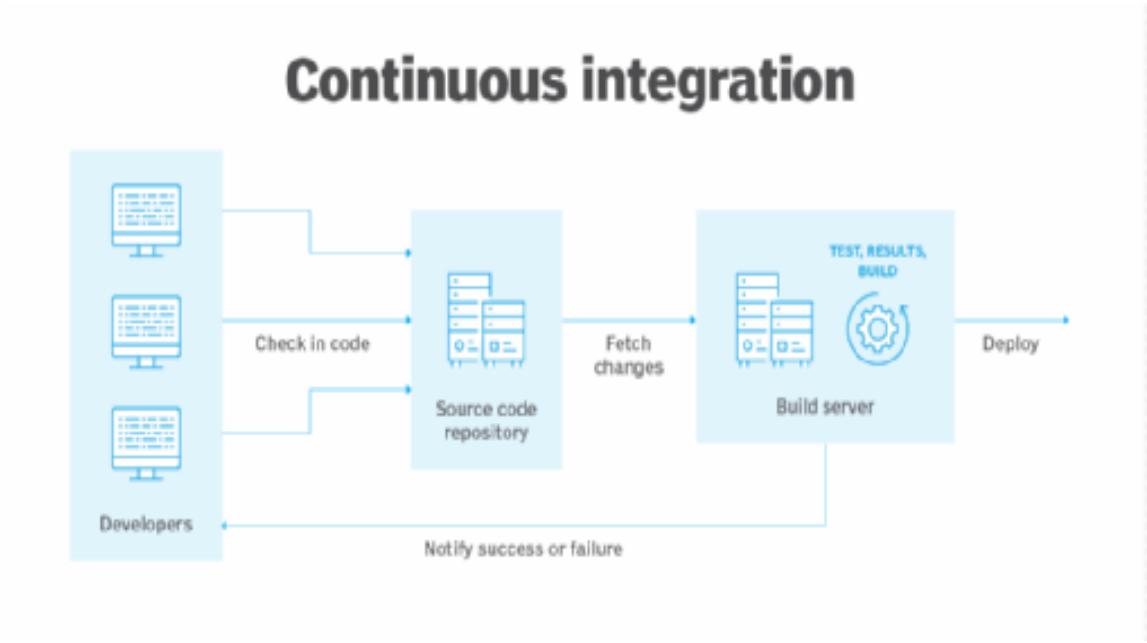


Рисунок 1.3 – Безперервна інтеграція дає змогу розробникам об'єднувати зміни коду в центральному репозиторії [3]

3. **Функціональне тестування**. Тестує відповідність функціональних вимог. Інструменти: Selenium, Cypress (Рис.1.4).

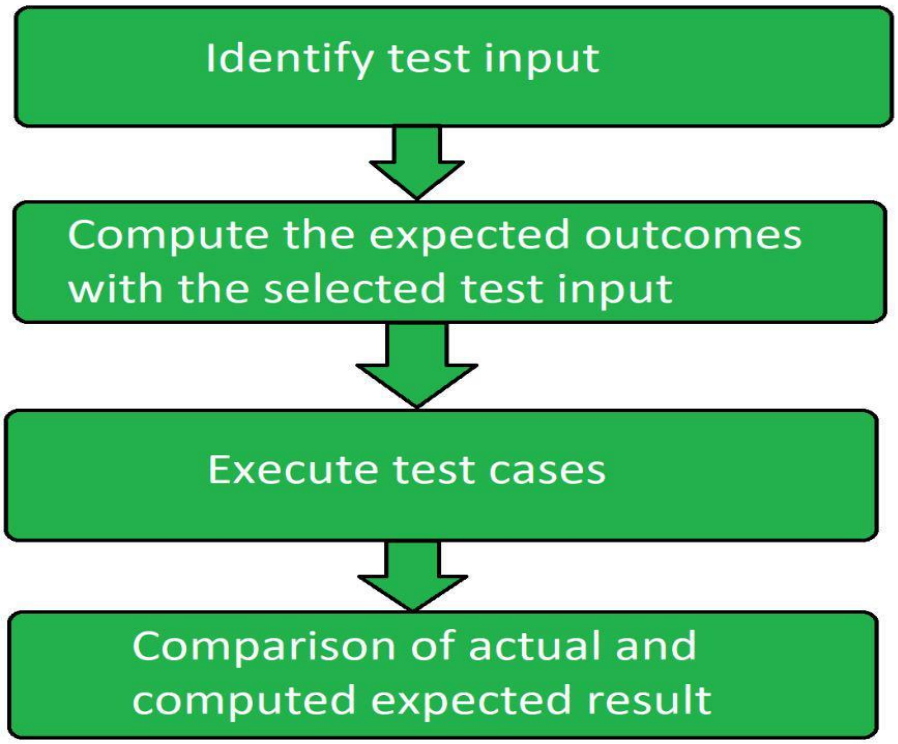


Рисунок 1.4 - Процес функціонального тестування [4]

4. **Навантажувальне тестування**. Аналіз продуктивності системи під різним навантаженням. Інструменти: JMeter, Gatling.

5. **Регресійне тестування**. Гарантує, що зміни в коді не порушили існуючий функціонал. Інструменти: TestComplete, Ranorex.

6. **Тестування на основі поведінки.**. Використовує зрозумілу для бізнесу мову опису тестів.. Інструменти: Cucumber, Behave.

Розглянемо сучасні інструменти автоматизованого тестування [8, 19, 20]:

1. Selenium. Для автоматизації веб-додатків. Підтримує різні мови програмування. Перевага: гнучкість і розширюваність. Недоліки: складність налаштування, обмеження для мобільних платформ.

2. Cypress. Проста інтеграція для тестування веб-додатків. Переваги: швидка установка, інтеграція в CI/CD. Недоліки: обмежена підтримка браузерів.

3. Appium. Для мобільного тестування (Android, iOS). Підтримує кросплатформність. Недоліки: висока залежність від налаштувань середовища.

4. JMeter. Навантажувальне тестування. Підтримує тестування API та веб-додатків. Недоліки: складність інтерфейсу.

5. Playwright. Альтернатива Selenium з підтримкою різних браузерів. Забезпечує швидше та стабільніше виконання тестів.

6. TestComplete. Для GUI та функціонального тестування. Підтримує запис і відтворення тестів.

7. Postman. Інтуїтивно зрозумілий інтерфейс.

8. CI/CD Інтеграція. Інструменти типу Jenkins, GitLab CI/CD, Travis CI забезпечують автоматичне виконання тестів у конвеєрі розробки.

Розглянемо тренди у автоматизованому тестуванні [9, 10, 11]:

1. **Використання штучного інтелекту**. Автоматизація написання та оптимізація тестів. Інструменти: Testim, Applitools.
2. **Контейнеризація**. Використання Docker для тестування в ізольованих середовищах.
3. **Shift-Left Testing**. Інтеграція тестування на ранніх етапах розробки.
4. **Кросбраузерне тестування**. Підтримка тестування на різних браузерах та пристроях.

Розглянкемо переваги автоматизованого тестування:

- Зниження витрат на довгострокову підтримку тестів.
- Прискорення процесу виявлення помилок.
- Висока повторюваність і точність тестів.

Розглянкемо виклики автоматизованого тестування:

- Високі початкові витрати на створення тестів.
- Залежність від інфраструктури.
- Складність підтримки тестів у динамічно змінюваних системах.

1.2 Аналіз сучасних підходів та інструментів автоматизованого тестування

Сучасні підходи та інструменти спрямовані на підвищення ефективності та якості цих процесів. Розглянемо основні підходи та їх особливості.(Рис.1.5)

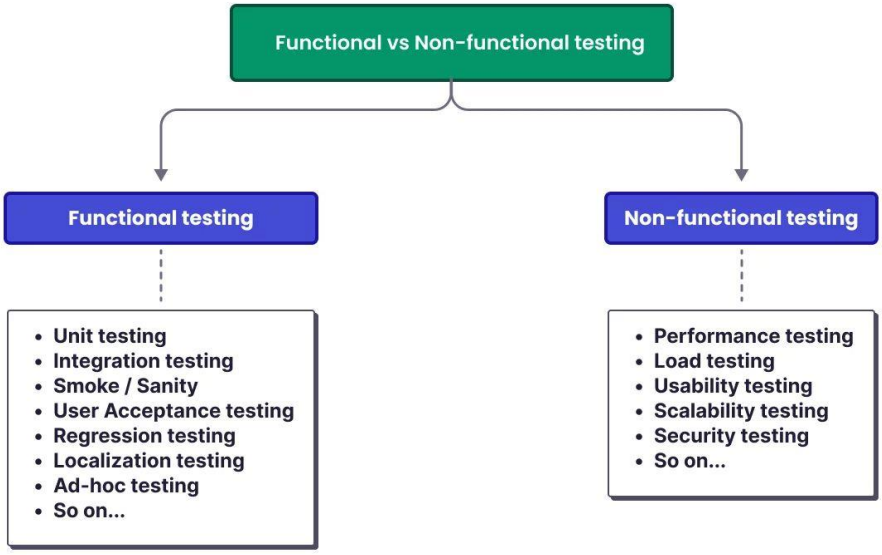


Рисунок 1.5 - Класифікація тестів, яка базується на об'єктах тестування [5]

Класифікація підходів до автоматизованого тестування [11, 12, 13]:

1.1. Підхіди до автоматизованого тестування з статичною аналітикою. Виконують аналіз коду без його виконання. Використовуються інструменти на основі лінтерів (наприклад, ESLint, PyLint). Автоматизоване тестування з використанням статичної аналітики включає кілька ключових підходів:

Вибір інструментів: Виберіть інструменти для статичної аналітики, які підтримують автоматизоване тестування.

Скриптінг: Напишіть скрипти для автоматизації тестування. Використовуйте мови програмування, такі як Python, Java або C#.

Тест-кейси: Створіть тест-кейси. Включіть в них тести на статичні аспекти, такі як перевірка на помилки, збереження даних та безпека.

Аналіз результатів: Використовуйте статичні аналітичні інструменти для аналізу результатів тестування. Це допоможе виявити помилки та проблеми, які можуть бути відсутніми при традиційному тестуванні.

Постійне покращення: Використовуйте результати аналізу для постійного покращення процесу тестування. Внесіть зміни в скрипти та тест-кейси на основі отриманих даних.

Ці підходи допоможуть вам ефективно використовувати статичну аналітику для автоматизації тестування та покращення якості програмного забезпечення.

1.2. Динамічне тестування. Динамічне тестування – це метод тестування програмного забезпечення, що включає виконання коду на реальному чи віртуальному середовищі. Мета цього тестування полягає у виявленні помилок, дефектів та недоліків в роботі програми під час її виконання. Основні аспекти динамічного тестування:

Функціональне тестування: Перевірка функцій програми відповідно до вимог. Включає в себе юніт-тестування, інтеграційне тестування, системне тестування та тестування прийнятності.

Нефункціональне тестування: Перевірка аспектів, які не пов'язані безпосередньо з функціональністю програми. Включає тестування продуктивності, безпеки, зручності використання та надійності.

Тестування безпеки: Виявлення вразливостей і загроз для забезпечення безпеки даних і захисту від атак.

Тестування продуктивності: Оцінка часу відгуку, пропускної здатності та стабільності програми під навантаженням.

Динамічне тестування важливо для забезпечення якості програмного забезпечення, оскільки воно дозволяє виявити реальні помилки під час виконання програмного коду. Це, в свою чергу, сприяє поліпшенню користувацького досвіду та зниженню ризиків, пов'язаних з розгортанням дефектного програмного забезпечення.

1.3. Підхід безперервної інтеграції (CI). Безперервна інтеграція (Continuous Integration, CI) є підходом у розробці програмного забезпечення, який полягає у

частому інтегруванні коду розробниками в загальний репозиторій кілька разів на день. Основна мета CI полягає у виявленні та вирішенні помилок на ранніх етапах розробки, що забезпечує високу якість коду та стабільність програмного забезпечення. Ключові аспекти CI:

Автоматизація збірки: Кожна зміна коду тригерить автоматичний процес збірки, що дозволяє швидко виявляти помилки.

Автоматичне тестування: Після збірки код проходить через набір автоматизованих тестів для перевірки його працездатності та якості.

Безперервний зворотній зв'язок: Розробники отримують миттєвий зворотній зв'язок про стан коду, що дозволяє швидко виправляти помилки.

Репозиторій коду: Використання версійного контролю (наприклад, Git) для зберігання та управління змінами в коді.

Інфраструктура: Використання CI-серверів (наприклад, Jenkins, Travis CI, CircleCI), які автоматично запускають збірку та тестування коду.

CI допомагає забезпечити стабільність та якість програмного забезпечення, зменшуючи час на інтеграцію та тестування. Це дозволяє розробникам зосередитися на створенні нових функцій і виправленні помилок, не витрачаючи багато часу на ручну перевірку та інтеграцію.

1.3 Аналіз системи автоматичного тестування на основі мови програмування Python

Система автоматичного тестування на основі Python може бути корисною для перевірки програмного забезпечення, веб-застосунків, а також для оцінювання знань студентів чи працівників [13, 14, 15]:

Особливості системи автоматичного тестування на основі мови програмування Python:

1. **Інтерфейс**: Веб-інтерфейс для користувачів та адміністраторів. Можливість додавання завдань, тестових сценаріїв та питань. Підтримка інтерактивного введення коду.

2. **Підтримка тестів**: Юніт-тестування: автоматична перевірка результатів функцій. Тестування продуктивності: перевірка швидкості виконання коду. Автоматична перевірка правильності виводу.

3. **Безпека**: Виконання коду в ізольованому середовищі (наприклад, Docker). Ліміти на час виконання та використання ресурсів.

4. **Звіти**: Генерація детальних звітів про результати тестування. Аналіз помилок та рекомендації щодо їх виправлення.

Технології системи автоматичного тестування на основі мови програмування Python:

Основний стек: Frontend: React, Angular, або простий HTML/CSS. Backend: Flask або Django. Ізольоване середовище: Docker, використання бібліотек типу execnet або subprocess.

Бібліотеки для тестування:

- `unittest` — стандартна бібліотека для тестування.
- `pytest` — популярна бібліотека з розширеними можливостями.
- `doctest` — перевірка прикладів у документації.
- `Pytest-Notebook` — для тестування Jupyter Notebook.

Основні етапи створення

1. Розробка інтерфейсу: Форма для введення коду. Панель адміністрування для створення та редагування завдань.

2. Реалізація тестування: Виконання коду у безпечному середовищі. Застосування юніт-тестів до студентських рішень.

3. Обробка результатів: Порівняння результатів виконання з очікуваними. Автоматичне виставлення балів.

4. Звіти та зворотний зв’язок: Відображення результатів у вигляді графіків або текстових звітів. Рекомендації щодо покращення рішень.

Приклад коду надається у Додатку А.

1.4 Аналіз ролі автоматизованого тестування у забезпеченні якості програмного забезпечення

Автоматизоване тестування відіграє вирішальну роль у забезпеченні високої якості програмного забезпечення, адже воно підвищує ефективність процесу тестування, мінімізує людські помилки та скорочує час виходу продукту на ринок. Нижче розглянемо основні аспекти його впливу [16, 17, 18].

1. Підвищення ефективності тестування:

- Швидкість виконання: автоматичні тести виконуються значно швидше, ніж ручне тестування.
- Повторюваність: одні й ті ж тести можна запускати необмежену кількість разів, що особливо важливо під час регресійного тестування.
- Масштабованість: легко тестувати складні та великі системи, що потребують багато сценаріїв перевірки.

2. Мінімізація людського фактора:

- Зменшення помилок: автоматизовані скрипти виконують тести з високою точністю, виключаючи ризик пропуску дефектів через неуважність.
- Стабільність: однакові умови виконання забезпечують постійність результатів.

3. Забезпечення всебічного тестування:

- Різноманітність типів тестування: Функціональне тестування перевіряє відповідність вимогам. Нефункціональне тестування (навантаження, продуктивність) забезпечує стабільність роботи системи за екстремальних умов.
- Кросплатформність: автоматизація дозволяє легко перевіряти програмне забезпечення на різних пристроях, операційних системах і браузерах.

4. Інтеграція в процес розробки

- CI/CD (Continuous Integration/Continuous Deployment): автоматизоване тестування є невід’ємною частиною сучасних конвеєрів розробки. Воно дозволяє швидко перевіряти якість коду на кожному етапі розробки.

- Shift-Left Testing: тести інтегруються на ранніх етапах розробки, що допомагає виявляти дефекти ще до стадії розгортання.

5. Покращення якості програмного забезпечення

- Раннє виявлення дефектів: виявлення помилок на ранніх стадіях значно скорочує витрати на їх виправлення.
- Підвищення продуктивності команди: розробники можуть більше зосередитися на розробці нових функцій, а не на виправленні багів.
- Забезпечення відповідності вимогам: тести перевіряють, чи відповідає функціонал заданим критеріям.

6. Переваги автоматизованого тестування для забезпечення якості

1. Економія ресурсів: зменшення витрат на повторне тестування.
2. Швидший вихід продукту на ринок: автоматизація пришвидшує процес розробки та релізу.
3. Підвищення задоволеності користувачів: стабільність і відсутність критичних помилок покращують досвід використання продукту.

7. Обмеження автоматизованого тестування

1. Початкові витрати: створення тестів і налаштування середовища потребують часу та ресурсів.
2. Необхідність постійної підтримки: автоматизовані тести потребують оновлення при зміні функціоналу.
3. Обмеження на творчі тести: деякі аспекти, такі як UX/UI, важко повністю автоматизувати.

Таким чином автоматизоване тестування є важливим елементом забезпечення якості програмного забезпечення. Воно дозволяє підвищити продуктивність команд розробників, забезпечити стабільність продукту та вчасно виявляти дефекти. Інтеграція цього підходу в життєвий цикл розробки сприяє створенню конкурентоспроможного та надійного програмного забезпечення.

1.5 Огляд існуючих систем автоматизованого тестування

1.5.1 Фреймворки автоматизації тестування: pytest, unittest, Selenium.

Фреймворки автоматизації тестування забезпечують структуру для створення, виконання та аналізу тестів, полегшуючи перевірку якості програмного забезпечення. Нижче розглянемо три популярні фреймворки: **pytest**, **unittest** та **Selenium** [19, 20, 21, 22]

1. Pytest. Pytest — популярний фреймворк для автоматизованого тестування на Python, який використовується для тестування модулів, функцій, API та інтеграцій. Має наступні особливості:

- Підтримка написання тестів з мінімумом коду.
- Зручна структура тестів.
- Просте налаштування параметризованих тестів.
- Інтеграція з іншими бібліотеками, такими як requests для тестування API.
- Плагіни для розширення функціональності, наприклад, pytest-html для звітів.
- Переваги: Гнучкість і простота. Потужна система плагінів. Добре підходить для проєктів будь-якого масштабу.
- Недоліки: Для великих проєктів може потребувати додаткового налаштування.

2. Unittest. Unittest — стандартний фреймворк тестування, вбудований у Python. Він надає базову структуру для модульного тестування. Має наступні особливості:

- Частина стандартної бібліотеки Python.
- Інтеграція з іншими інструментами, такими як mock для створення замінників об'єктів.
- Створення тестів на основі класів.
- Переваги: Надійність і доступність (вбудований в Python). Підтримує групування тестів за класами.

- Недоліки: Більш громіздкий синтаксис у порівнянні з pytest. Менше можливостей для параметризації.

3. Selenium. Selenium — фреймворк для автоматизації тестування веб-додатків. Підходить для функціонального та регресійного тестування браузерів. Має наступні особливості:

- Підтримує основні мови програмування: Python, Java, C#, Ruby тощо.
- Може автоматизувати роботу з будь-яким сучасним веб-браузером.
- Підтримує тестування на реальних пристроях.
- Переваги: Потужний інструмент для тестування UI. Кросбраузерна підтримка. Інтеграція з іншими бібліотеками, такими як Pytest.
- Недоліки: Потребує значних ресурсів для налаштування. Тести можуть бути крихкими через зміни в DOM.

Порівняння фреймворків:Ppytest, Unittest та Selenium розглянемо в табл. 1.1

Таблиця 1.1 - Порівняльна таблиця

Фреймворк	Призначення	Простота використання	Підтримка параметризації	Інтеграція з іншими інструментами
Pytest	Модульне, API	Висока	Так	Висока
Unittest	Модульне	Середня	Обмежена	Середня
Selenium	Функціональне (UI)	Низька (складне налаштування)	Немає	Висока

1.2.2 Інструменти для безперервної інтеграції: Jenkins, GitHub Actions. Інструменти для **безперервної інтеграції (Continuous Integration, CI)** забезпечують автоматизацію процесів збірки, тестування та розгортання програмного забезпечення. Вони допомагають командам швидко інтегрувати зміни в коді, виявляти помилки та прискорювати релізи. Розглянемо два популярні інструменти: **Jenkins** та **GitHub Actions**.[23, 24, 25]

1. Jenkins. Jenkins — це один із найпопулярніших інструментів для CI/CD, що підтримує великий набір плагінів для інтеграції з різними інструментами розробки. Має наступні основні функції:

- Автоматизація збірки, тестування та розгортання.
- Гнучке налаштування через **Pipeline Scripts** (Groovy).
- Підтримка інтеграції з Git, Docker, Kubernetes та іншими інструментами.
- Понад 1800 плагінів для розширення можливостей.
- Переваги: Підтримка складних сценаріїв CI/CD. Платформа з відкритим вихідним кодом і великою спільнотою. Широка інтеграція з інструментами розробки.
- Недоліки: Складність налаштування та обслуговування. Потребує окремого сервера для запуску.

  Приклад коду надається у Додатку Б.

2. GitHub Actions. GitHub Actions — це CI/CD інструмент, інтегрований у платформу GitHub. Він дозволяє автоматизувати робочі процеси безпосередньо у сховищах GitHub. Має наступні основні функції:

- Автоматизація збірки, тестування, розгортання та інших завдань.
- Підтримка широкого спектра подій, таких як push, pull request, issue тощо.
- Підтримка контейнерів Docker для запуску тестів.
- Велика бібліотека готових Action-скриптів.
- Переваги: Інтеграція з GitHub дозволяє легко налаштовувати робочі процеси. Безкоштовне використання для публічних репозиторіїв. Велика кількість готових Action-скриптів для поширених завдань.
- Недоліки: Менше можливостей для кастомізації, ніж у Jenkins. Залежність від платформи GitHub.

Порівняння фреймворків:Ppytest, Unittest та Selenium розглянемо в табл. 1.2

Таблиця 1.2- Порівняння: Jenkins vs GitHub Actions

Параметр	Jenkins	GitHub Actions
Тип	Автономний CI/CD сервер	Інтегрований в GitHub
Налаштування	Складне, потребує окремого сервера	Просте, налаштовується через YAML
Плагіни/інтеграції	1800+ плагінів	Багато готових Action-скриптів
Гнучкість	Висока, підтримує кастомні сценарії	Середня, але покриває більшість потреб
Вартість	Безкоштовний, але потребує сервера	Безкоштовно для публічних репозиторіїв
Швидкість розгортання	Повільне через складність налаштування	Швидке, працює з репозиторіями GitHub

Таким чином:

- Jenkins підходить для великих проєктів, де потрібна гнучкість і складні сценарії CI/CD.
- GitHub Actions є ідеальним вибором для невеликих або середніх проєктів, особливо якщо вони вже використовують GitHub.

1.6 Аналіз можливостей Python для автоматизації тестування

Python є мовою програмування для автоматизації тестування. Він пропонує широкий спектр інструментів для різних видів тестування, від модульного до інтеграційного, а також для тестування інтерфейсу користувача (UI) та API.(Рис.1.6)

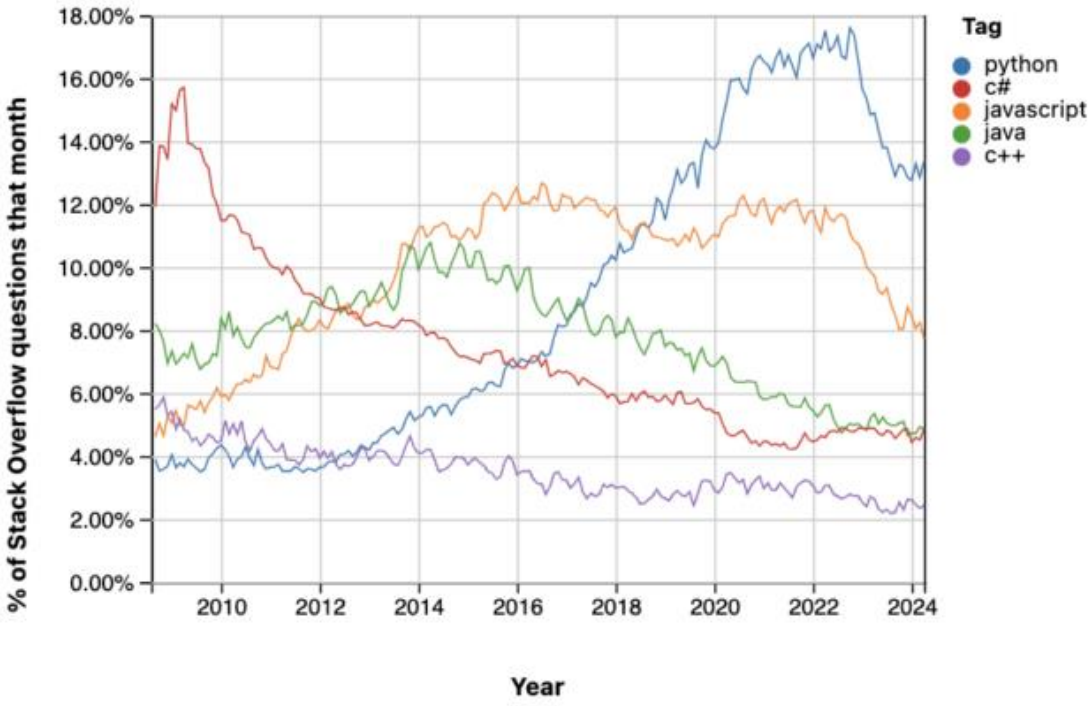


Рисунок 1.6 – Статистика популярності мов программування за даними StackOverflow [14]

1.6.1 Ключові переваги Python для автоматизації тестування. Ключові переваги Python для автоматизації тестування наступні:

Простота у використанні. Python має легкий синтаксис, який робить тестові скрипти зрозумілими навіть для початківців. Читабельний код спрощує розуміння та підтримку тестів.

Широкий спектр бібліотек. Python має потужні бібліотеки та фреймворки, які покривають всі аспекти автоматизації тестування:

- Для модульного тестування: unittest, pytest, nose2.
- Для веб-автоматизації: Selenium, Playwright.
- Для тестування API: requests, pytest, Postman API.
- Для навантажувального тестування: locust, JMeter API.
- Для тестування баз даних: sqlite3, SQLAlchemy.

Кросплатформність. Python працює на різних операційних системах (Windows, macOS, Linux), що дозволяє створювати універсальні рішення для автоматизації тестування.

Інтеграція з іншими інструментами. Python легко інтегрується з інструментами для безперервної інтеграції, такими як Jenkins, GitHub Actions, GitLab CI. Можливість використовувати Docker для створення ізольованих середовищ тестування.

1.6.2 Фреймворки та бібліотеки Python для автоматизації тестування

Pytest. Pytest є одним із найпопулярніших фреймворків для автоматизації тестування на Python. Опис: Легкий і потужний фреймворк для тестування, що дозволяє легко писати тест-кейси, сконцентровані на читабельності та розширюваності. Основні функції: Підтримка фікстур для налаштування середовища тестування. Легка інтеграція з іншими інструментами, такими як Jenkins та Docker. Можливість запуску окремих тестів чи груп тестів. Вбудована підтримка для поведінкового тестування (BDD) через плагін pytest-bdd.

Unittest. Unittest - це вбудований фреймворк для тестування в стандартній бібліотеці Python, що базується на моделі "unit test". Він базується на моделі тестування unit test і дозволяє створювати організовані та модульні тест-кейси для вашого коду. Вбудований фреймворк для тестування

Основні функції: Наслідується від unittest.TestCase для створення тестових класів. Підтримка створення наборів тестів і запуску тестів у певному порядку. Легко інтегрується з іншими інструментами.

Основні компоненти Unittest:

1. TestCase: Базовий клас, який використовується для створення нових тестів. Ви наслідуєте від unittest.TestCase і додаєте методи, які містять ваші тест-кейси.
2. Assertions: Методи, які використовуються для перевірки результатів тестування. Наприклад, assertEqual, assertTrue, assertFalse тощо.
3. Test Suite: Група тестів, яка об'єднує кілька тестових випадків для спільного запуску.

4. Test Runner: Компонент, який відповідає за виконання тестів та відображення результатів. Unittest включає в себе командний рядок тестовий раннер.

Nose2. Nose2 є продовженням і розвитком оригінального Nose фреймворку для тестування. Фреймворк, що базується на Unittest, але додає багато додаткових функцій і зручностей. Він надає багато корисних функцій для автоматизації тестування у Python і базується на бібліотеці Unittest, розширюючи її можливості.

Основні функції: Автоматичне виявлення тестів. Підтримка плагінів для розширення функціональності. Інтеграція з Unittest і підтримка його тестів.

Основні можливості Nose2:

1. Автоматичне виявлення тестів: Nose2 автоматично знаходить і виконує тести у вашому проекті, що робить процес налаштування і запуску тестів більш простим.
2. Розширюваність за допомогою плагінів: Nose2 підтримує плагіни, які можна використовувати для розширення функціоналу. Існують плагіни для різних цілей, таких як генерація звітів, паралельне виконання тестів і багато іншого.
3. Сумісність з Unittest: Ви можете використовувати тести, написані для Unittest, без необхідності змінювати їх. Це забезпечує легку міграцію існуючих проектів.
4. Гнучкість конфігурації: Можна налаштовувати поведінку Nose2 за допомогою конфігураційних файлів або командного рядка, що дозволяє легко інтегрувати його в існуючі CI/CD процеси.

Robot Framework. Robot Framework – це потужний фреймворк для автоматизації тестування та робить його особливо зручним для не технічних користувачів завдяки використанню ключових слів. Фреймворк для автоматизації тестування, що використовує ключові слова для опису тест-кейсів, що робить

його особливо підходящим для не технічних користувачів. Він дозволяє писати тести у вигляді простих текстових файлів, які легко зрозуміти і підтримувати.

Основні функції: Можливість створення власних бібліотек ключових слів на Python та інших мовах. Підтримка тестування веб-додатків, API та інших видів систем. Легка інтеграція з іншими інструментами, такими як Selenium, Appium, і багато інших.

Основні можливості Robot Framework:

1. Ключові слова: Тести пишуться у вигляді ключових слів, що робить їх читабельними та зрозумілими для всіх учасників команди.
2. Розширюваність: Ви можете створювати власні бібліотеки ключових слів на Python або Java для розширення функціональності фреймворку.
3. Плагіни та бібліотеки: Підтримка великої кількості бібліотек, таких як SeleniumLibrary для тестування веб-додатків, DatabaseLibrary для тестування баз даних та багато інших.
4. Звіти та логи: Автоматичне генерування детальних звітів та логів, що допомагає у відстеженні результатів тестування і виявленні проблем.

Behave. Behave — це фреймворк для поведінкового тестування (BDD) на Python. Він дозволяє описувати тести у вигляді сценаріїв на природній мові, використовуючи формат Gherkin. Фреймворк для поведінкового тестування (BDD), що дозволяє писати тести у вигляді історій користувача. Це робить його особливо корисним для спільної роботи між розробниками, тестувальниками та іншими учасниками проекту.

Основні функції: Використання Gherkin мови для написання тест-кейсів. Легка інтеграція з іншими інструментами і бібліотеками Python. Підтримка декількох форм співпраці між розробниками, тестувальниками та іншими учасниками проекту.

Основні особливості Behave:

1. Формат Gherkin: Використання Gherkin мови для написання тестів, що дозволяє описувати сценарії у вигляді "історій користувача".

2. Легка інтеграція: Проста інтеграція з іншими інструментами та бібліотеками Python.
3. Розширюваність: Можливість створення власних кроків (step definitions) на Python.

Недоліки Python для автоматизації тестування:

- Повільність: для великих і критичних проєктів може бути менш ефективним у порівнянні з Java або C#.
- Обмеження багатопотоковості: через GIL (Global Interpreter Lock) у деяких випадках Python поступається за продуктивністю.

Таким чином Python є потужним інструментом для автоматизації тестування завдяки своїй універсальності, багатому набору бібліотек та фреймворків. Він підходить для модульного, інтеграційного, навантажувального тестування, а також для тестування API та інтерфейсів.

2 ОБГРУНТУВАННЯ ЗАСТОСУВАННЯ СУЧАСНИХ ПІДХОДІВ ДО ВИБІРУ ОПТИМАЛЬНИХ ІНСТРУМЕНТІВ ТЕХНОЛОГІЇ PYTHON ДЛЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

2.1 Аналіз факторів, що мають вплив на вибір оптимальних інструментів і технологій для автоматизації тестування на Python

Вибір оптимальних інструментів і технологій для автоматизації тестування на Python залежить від багатьох факторів, таких як тип тестування, масштаб проєкту, доступні ресурси, інтеграційні можливості та вимоги до продуктивності. Сучасні підходи передбачають застосування систематичного аналізу потреб проєкту та доступних технологій (Рис.2.1).

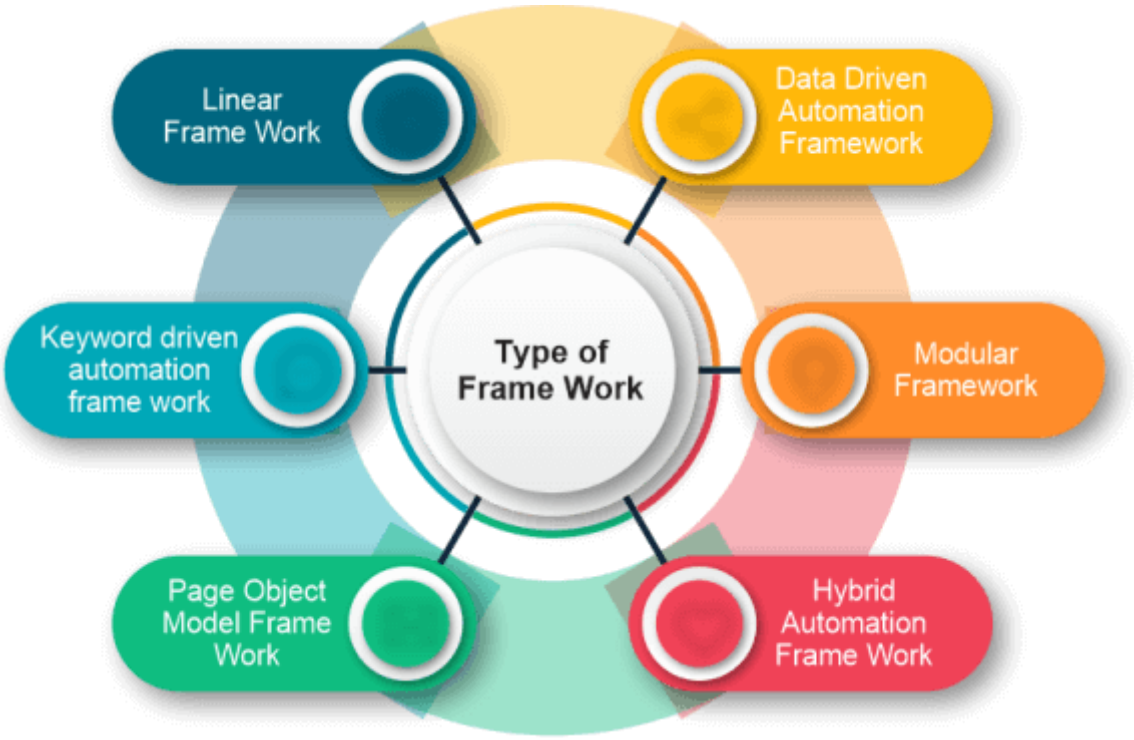


Рисунок 2.1 – Типи фреймворків тестування [15]

1. Ключові фактори вибору інструментів

1.1 Тип тестування:

Модульне тестування: вибір фреймворків, таких як pytest або unittest.

- Інтеграційне тестування: використання Pytest із плагінами для роботи з базами даних і API.
- Функціональне тестування: використання бібліотек на кшталт Selenium або Playwright.
- Навантажувальне тестування: Locust для тестування продуктивності.
- Тестування API: requests та pytest для автоматизації REST-запитів.

1.2 Масштаб проєкту: Для невеликих проєктів підходять прості інструменти (unittest, pytest). Для великих проєктів потрібні масштабовані рішення із підтримкою CI/CD (pytest, інтеграція з Jenkins або GitHub Actions).

1.3 Інтеграція з іншими інструментами: Інструменти мають бути сумісними з CI/CD платформами (Jenkins, GitHub Actions, GitLab CI). Підтримка Docker для ізоляції тестового середовища.

1.4 Простота використання та навчання: Для команд з різним рівнем досвіду слід обирати інструменти з доступною документацією та активною спільнотою.

2. Огляд сучасних технологій Python для тестування

2.1 Pytest Переваги: Простота створення тестів. Потужна система плагінів. Підтримка параметризації та асинхронного тестування. Коли вибирати: Для проєктів будь-якого масштабу, особливо з інтеграцією CI/CD.

2.2 Unittest. Переваги: Частина стандартної бібліотеки Python. Легкість інтеграції з іншими модулями. Коли вибирати: Для невеликих проєктів або якщо потрібно мінімізувати залежності.

2.3 Selenium. Переваги: Підтримка автоматизації для всіх популярних браузерів. Інтеграція з Pytest для створення тестів. Коли вибирати: Для тестування веб-інтерфейсів.

2.4 Playwright Переваги: Швидший за Selenium. Підтримка тестування UI для кількох браузерів. Коли вибирати: Для сучасних веб-додатків з високими вимогами до продуктивності.

2.5 Locust. Переваги: Легка вивченість. Масштабованість для великих навантажувальних тестів. Коли вибирати: Для навантажувального тестування API або веб-додатків.

3. Сучасні підходи до вибору технологій

3.1 Аналіз вимог. Перед вибором інструментів потрібно врахувати:

- Тип тестів (модульні, інтеграційні, UI, навантажувальні).
- Чи потрібна інтеграція з іншими системами.
- Частота та масштаб тестування (локальне чи в хмарі).

3.2 Оцінка технологій. Використовуються такі критерії:

- Простота використання: час, потрібний для навчання та початку роботи.
- Гнучкість: можливість кастомізації та розширення.
- Документація: наявність прикладів, гайдів та активної спільноти.

3.3Прототипування. Перед повною інтеграцією нової технології варто створити прототип для оцінки її ефективності в реальних умовах проєкту.

4. Рекомендації щодо оптимального вибору. Рекомендації щодо оптимального вибору надано в таблиці 2.1

Таблиця 2.1 Рекомендації щодо оптимального вибору

Тип тестування	Рекомендовані інструменти	Додаткові примітки
Модульне тестування	Pytest, Unittest	Pytest має більше функцій для масштабних проєктів.
Інтеграційне тестування	Pytest + плагіни	Використовуйте з базами даних чи API.
Тестування UI	Selenium, Playwright	Playwright швидший і простіший для сучасних веб-додатків.
Навантажувальне тестування	Locust	Підходить для великих навантажень.
API-тестування	Requests, Pytest	Використовуйте з параметризацією

Тип тестування	Рекомендовані інструменти	Додаткові примітки
		для багатьох сценаріїв.

5. Приклад практичного підходу

Сценарій: автоматизація тестування веб-додатка

1. Аналіз потреб:
    - Перевірка API і функціоналу UI.
    - Інтеграція з Jenkins.
2. Вибір інструментів:
    - API: Pytest + Requests.
    - UI: Playwright.
3. Реалізація:
    - Створення тестів на основі Pytest з інтеграцією в CI/CD.
    - Використання Playwright для швидкої автоматизації UI.
4. Оцінка: аналіз звітів із CI/CD для моніторингу стабільності.

Таким чином Python забезпечує багатий набір інструментів для автоматизації тестування, який можна адаптувати до будь-яких вимог. Сучасний підхід до вибору технологій включає аналіз вимог, оцінку доступних рішень та тестування їх ефективності в реальних умовах.

2.2 Основні види тестування

2.2.1 Юніт-тестування (Unit Testing). Юніт-тестування — це процес тестування окремих модулів або компонентів програмного забезпечення для перевірки їхньої коректності. Воно є основою автоматизованого тестування та забезпечує високу якість програмного продукту на ранніх етапах розробки.(рис.2.2)

Рисунок 2.2 – Робочий процес модульного тестування [26]

1. Основні аспекти юніт-тестування

1.1 Мета юніт-тестування. Перевірка правильності роботи окремих функцій, методів або класів. Швидке виявлення та виправлення помилок на ранніх стадіях розробки. Підвищення впевненості в стабільності системи при внесенні змін до коду.(Рис.2.3)

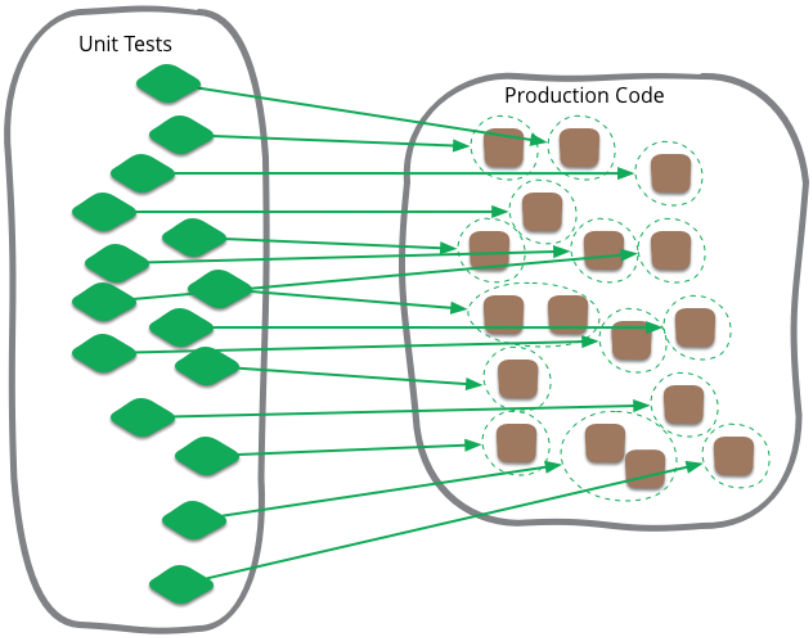


Рисунок 2.3 – Модульне тестування [27]

1.2 Вимоги до юніт-тестів:

- Автономність: тест не повинен залежати від інших тестів.
- Простота: тест повинен перевіряти одну конкретну функціональність.

- Відтворюваність: результати тесту мають бути однаковими при повторних запусках.
- Швидкість: юніт-тести повинні виконуватися швидко, щоб не уповільнювати розробку.

2. Інструменти для юніт-тестування на Python

2.1 Unittest (код у додатку Г п. 2.1):

- Стандартна бібліотека: не потребує додаткових установок.
- Основні можливості: створення тестових класів, виконання тестів, перевірка результатів.
- Переваги: простота інтеграції та сумісність із будь-якими середовищами.
- Недоліки: менш гнучкий у порівнянні з іншими фреймворками.

2.2 Pytest (код у додатку Г п.2.2):

- Більш сучасний підхід: дозволяє писати компактні й читабельні тести.
- Основні можливості: підтримка параметризації, інтеграція з плагінами.
- Переваги: простота, розширюваність, гнучкість.
- Недоліки: потребує додаткової установки.

2.3 Nose2

- Розширення Unittest: полегшує організацію тестів.
- Переваги: автоматичне виявлення тестів, підтримка плагінів.
- Недоліки: менш популярний, ніж Pytest.

3. Процес створення юніт-тестів

3.1 Етапи

1. Підготовка середовища: Вибір інструменту (`unittest`, `pytest` тощо). Організація тестових файлів і каталогів.
2. Написання тестів: Кожна функція тесту перевіряє одну конкретну функціональність. Використання тверджень (`assert` у Pytest або `assertEqual` у Unittest).
3. Запуск тестів: Автоматизація виконання через CI/CD системи (наприклад, Jenkins, GitHub Actions).
4. Аналіз результатів: Виправлення помилок і повторне тестування.

3.2 Рекомендації

- Використовуйте mocking для імітації залежностей, таких як база даних або API.
- Використовуйте TDD (Test-Driven Development): спочатку пишіть тести, а потім код.

4. Mocking у юніт-тестуванні. Mocking — це створення підроблених об'єктів для заміни реальних залежностей (наприклад, баз даних, API), щоб ізолювати модуль під час тестування. Процес надано на Рис.2.4.

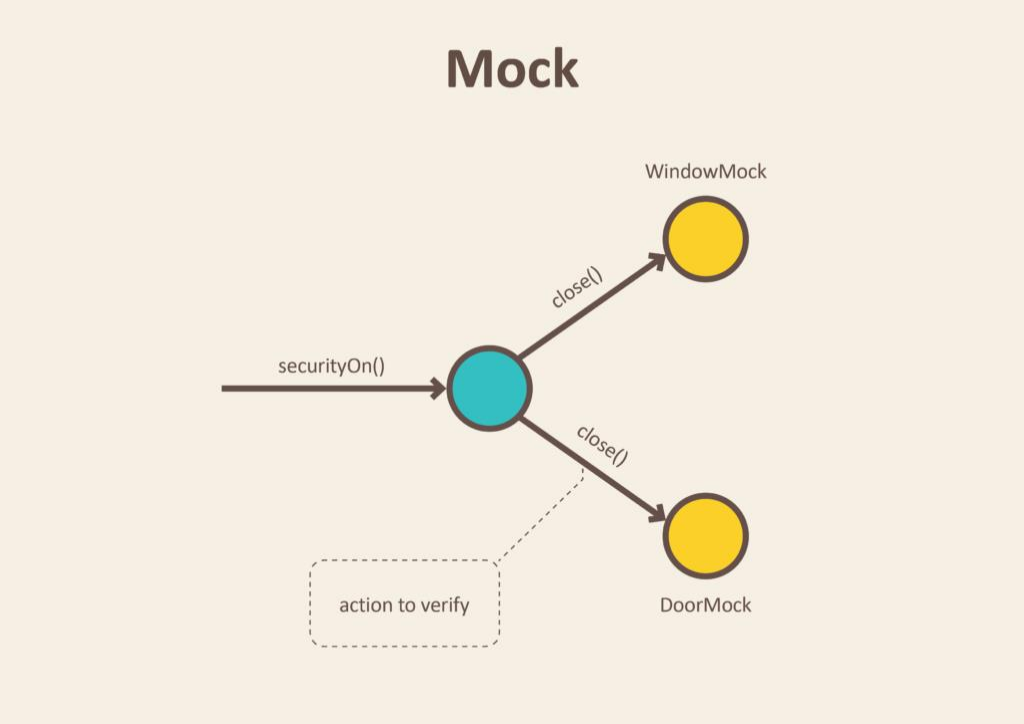


Рисунок 2.4 – Імітації замінюють реальні об'єкти «Двері» та «Вікно»[28]

Приклад використання unittest.mock надається у Додатку Г (п.2.3)

5. Інтеграція юніт-тестування з CI/CD. Юніт-тести повинні автоматично запускатися при кожному коміті або створенні pull request. Використовуйте інструменти, такі як GitHub Actions, для автоматизації (див Додаьтлок Г п.2.4):

6. Переваги юніт-тестування: Раннє виявлення помилок. Підтримка стабільності коду при внесенні змін. Полегшує рефакторинг. Підвищує довіру до функціональності модулів.

7. Недоліки юніт-тестування: Не перевіряє інтеграцію між компонентами. Може потребувати багато часу на написання тестів.

Таким чином Юніт-тестування є базовою практикою забезпечення якості програмного забезпечення. Python пропонує гнучкі інструменти, такі як unittest і pytest, які дозволяють ефективно реалізовувати юніт-тести. Інтеграція тестів із CI/CD процесами забезпечує стабільність і надійність проєктів.

2.2.2 Інтеграційне тестування. Інтеграційне тестування перевіряє взаємодію між різними модулями або компонентами системи, щоб переконатися, що вони працюють разом правильно. На відміну від юніт-тестів, які зосереджені на окремих функціях або класах, інтеграційне тестування охоплює ширший контекст, включаючи зовнішні залежності (бази даних, API, мікросервіси).

1. Основні аспекти інтеграційного тестування

1.1 Мета інтеграційного тестування: Виявлення дефектів, які виникають через неправильну взаємодію між модулями. Перевірка коректності передачі даних між компонентами. Забезпечення стабільної роботи системи при інтеграції нових функцій.

1.2 Стратегії інтеграційного тестування:

- Bottom-up (знизу вгору): Тестування починається з нижчих рівнів (наприклад, модулів бази даних) і поступово рухається до вищих рівнів (інтерфейсів).
- Top-down (згори вниз): Починається з верхніх рівнів (інтерфейсів) і поступово спускається до нижчих модулів.
- Big Bang: Усі модулі інтегруються одночасно, і система тестується як ціле.
- Incremental: Модулі інтегруються й тестуються поступово.

2. Інструменти для інтеграційного тестування на Python

2.1 Pytest: Використовується для написання інтеграційних тестів завдяки простоті й потужній системі плагінів. Підтримує параметризацію та використання фікстур для підготовки тестового середовища**.** Приклад у Додатку Ґ п.2.1.

2.2 Unittest: Може використовуватися для інтеграційних тестів разом із модулями, такими як mock для імітації залежностей. Приклад у Додатку Ґ п.2.2.

2.3 Postman + Newman: Використовується для тестування API, які є важливою частиною інтеграційного тестування. Можна автоматизувати тестування за допомогою CLI.

2.4 Docker:Використовується для створення ізольованого середовища для інтеграційного тестування (наприклад, бази даних, API). Забезпечує стабільність і відтворюваність середовища. Приклад у Додатку Ґ п.2.4

3. Тестування залежностей: бази даних та API

3.1 Тестування з базою даних: Для інтеграційного тестування баз даних можна використовувати:

- Фікстури в Pytest: для створення тестових даних.
- Інструменти на основі SQL: такі як sqlalchemy для тестування запитів. Приклад у Додатку Ґ п.3.1

3.2 Тестування API: Використовуйте бібліотеку requests для перевірки RESTful API. Застосовуйте mocking для імітації зовнішніх сервісів (responses бібліотека).

4. Mocking у інтеграційному тестуванні: Mocking дозволяє імітувати залежності, які не є під контролем тестувальників. Це особливо важливо для інтеграційного тестування. Приклад у Додатку Ґ п.4

5. Інтеграція з CI/CD. Інтеграційні тести є невід'ємною частиною процесу CI/CD. Інструменти, такі як Jenkins або GitHub Actions, дозволяють автоматично виконувати інтеграційні тести при кожному оновленні коду. Приклад у Додатку Ґ п.5

6. Переваги інтеграційного тестування: Забезпечує правильну взаємодію між компонентами. Виявляє дефекти, що виникають через неправильну інтеграцію. Полегшує тестування складних систем.

7. Недоліки: Потребує більше часу, ніж юніт-тестування. Вимагає налаштування середовища (наприклад, баз даних або мікросервісів). Можуть бути складнощі у відтворенні помилок.

Таким чином Інтеграційне тестування є важливим етапом забезпечення якості програмного забезпечення, що дозволяє виявляти проблеми у взаємодії між компонентами. Python пропонує потужний набір інструментів для реалізації інтеграційних тестів, таких як pytest, unittest, Docker та інші. Інтеграція тестів у CI/CD процеси забезпечує стабільність розробки та скорочує час виявлення помилок.

2.2.3 Функціональне тестування. Функціональне тестування є одним із основних видів тестування програмного забезпечення, яке перевіряє, чи відповідає програмний продукт заданим вимогам і специфікаціям. Метою функціонального тестування є впевненість у тому, що кожна функція програми працює відповідно до очікувань.

1. Основні аспекти функціонального тестування

1.1 Мета функціонального тестування: Перевірка того, чи відповідає програмне забезпечення бізнес-вимогам та специфікаціям. Оцінка правильності роботи функціональних елементів системи, таких як інтерфейси, обробка даних, бізнес-логіка.

1.2 Методи функціонального тестування:

- Чорний ящик (Black Box Testing): Тестування без знання внутрішньої структури програми. Тестувальник перевіряє функціональність.
- Білий ящик (White Box Testing): Тестування з глибоким знанням внутрішньої структури коду. Зазвичай застосовується для створення юніт-тестів.

2. Інструменти для функціонального тестування на Python

2.1 Selenium. Це інструмент для автоматизованого тестування веб-додатків. Підтримує різні мови програмування, зокрема Python. Основні можливості: Підтримка взаємодії з веб-сторінками (натискання кнопок, заповнення форм, перевірка елементів на сторінці). Переваги: Потужний для автоматизації веб-тестів, може інтегруватися з іншими інструментами CI/CD. Недоліки: Потребує налаштування WebDriver, більш складний у порівнянні з

іншими інструментами для простих тестів. Приклад використання Selenium у Додатку Д:

2.2 Pytest. Це популярний фреймворк для тестування, який також підтримує функціональні тести. Основні можливості: Можливість писати тести у вигляді простих функцій, підтримка фікстур для підготовки середовища, розширення через плагіни. Переваги: Простота написання та читання тестів, великий набір плагінів, інтеграція з Selenium для веб-тестування. Недоліки: Може бути менш інтуїтивним для початківців у тестуванні. Приклад використання Pytest для тестування API у Додатку Д:

2.3 Postman + Newman: Це інструмент для тестування API, який дозволяє створювати та запускати запити. Основні можливості: Можливість автоматизації тестів API за допомогою Newman, який є командним інтерфейсом для Postman. Переваги: Легкість створення тестів API, підтримка перевірки відповідей і параметризації. Недоліки: Потрібна окрема установка Postman і Newman.

3. Етапи функціонального тестування

3.1 Підготовка тестового середовища: Налаштування середовища (сервери, бази даних, API). Підготовка даних для тестування.

3.2 Написання тестів: Створення тестових сценаріїв, які відображають реальні сценарії використання. Використання інструментів, таких як Selenium для автоматизації взаємодії з інтерфейсами, або Pytest для тестування логіки.

3.3 Виконання тестів: Запуск тестів вручну або автоматизовано. Аналіз результатів тестування.

3.4 Виправлення помилок і повторне тестування. Виправлення дефектів, знайдених під час тестування. Повторний запуск тестів для перевірки виправлень.

4. Переваги функціонального тестування. Перевірка відповідності продукту бізнес-вимогам. Виявлення помилок у роботі інтерфейсу та логіки. Підвищення якості програмного забезпечення через перевірку функціональних аспектів.

5. Недоліки функціонального тестування. Може бути недостатньо для виявлення проблем із продуктивністю. Вимагає значних ресурсів для тестування всіх функцій. Залежність від правильності тестових сценаріїв.

Таким чином. Python надає потужні інструменти, такі як Selenium і Pytest, для ефективного виконання функціональних тестів, а також підтримку тестування API через Postman і Newman.

2.3 Методології тестування: TDD та BDD

Методології тестування, як TDD (Test-Driven Development) та BDD (Behavior-Driven Development), є підходами, що сприяють покращенню якості коду та ефективності розробки програмного забезпечення. Вони дозволяють інтегрувати тестування безпосередньо в процес розробки, підвищуючи зручність співпраці між розробниками та тестувальниками Рис.2.5.

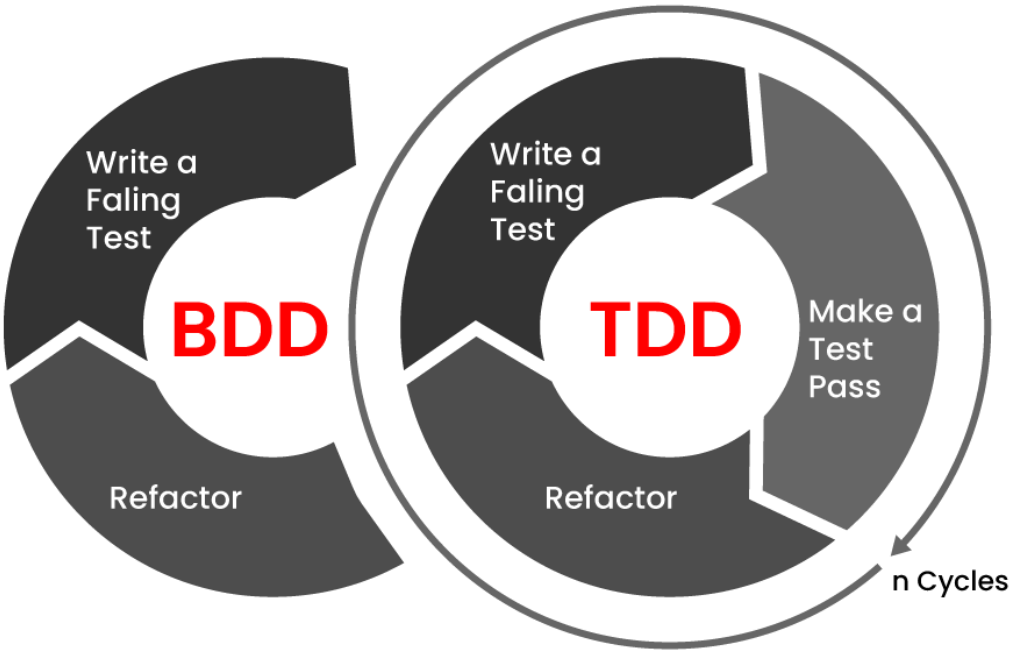


Рисунок 2.5 – Візуалізація TDD, BDD [36]

TDD (Test-Driven Development).

TDD — це методологія розробки програмного забезпечення, в якій тестування є початковим етапом перед написанням коду. Головна ідея TDD полягає в тому, що спочатку створюється тест, який не проходить, а потім пишеться код для його успішного виконання.

Основні етапи TDD: Написання тесту: Створюється тест, що описує очікувану поведінку коду. Запуск тесту: Тест виконується і не проходить (оскільки код ще не реалізовано). Розробка коду: Пишеться мінімальний код, необхідний для проходження тесту. Запуск тесту: Тест знову виконується, і якщо він проходить, переходять до наступного етапу. Рефакторинг: Код може бути оптимізований без зміни його функціональності. Повторення: Цикл повторюється для кожної нової функції.

Переваги TDD: Рання виявленість помилок: Помилки знаходяться на ранніх етапах розробки. Чистий код: Створення коду, орієнтованого на тестування, часто веде до написання більш читабельного коду. Зниження технічного боргу: Підходить для створення надійної архітектури та підтримки коду.

Недоліки TDD: Часовитратність: Потребує більше часу на написання тестів. Складність у великих системах: Тестування інтеграційних аспектів може бути складним. Може знижувати гнучкість: Зміни в вимогах можуть вимагати значних змін у тестах.

Інструменти для TDD на Python: Pytest: Може використовуватись для написання тестів у стилі TDD. unittest: Стандартний модуль для тестування в Python, також підтримує TDD. Hypothesis: Для тестування на основі властивостей, що допомагає створювати більш обширні тести. Приклад TDD з Pytest надається у Додатку Е п.1:

BDD (Behavior-Driven Development)

BDD — це методологія розробки, яка акцентує увагу на поведінці програми з точки зору користувачів. Вона розширює ідеї TDD, включаючи більш зрозумілу

мову для опису тестів, що робить їх доступними для технічних і нетехнічних учасників команди (розробників, тестувальників, аналітиків).

Основні принципи BDD.

- Поведінка замість тестів: Описуються сценарії використання, які демонструють, як програма повинна поводитися.
- Зрозумілий синтаксис: Використовується мова, яку розуміють усі учасники команди.
- Співпраця: Сценарії тестів пишуться разом з усіма учасниками процесу.

Формат сценаріїв BDD. Сценарії в BDD часто пишуться у вигляді Gherkin-синтаксису: Given, When, Then.

Приклад сценарію BDD надається у Додатку Е п.2.

Переваги BDD. Спільне розуміння: Сценарії тестів зрозумілі для всіх учасників проєкту. Краща комунікація: Підходить для міжфункціональної співпраці. Документація: Тести стають частиною документації проєкту.

Недоліки BDD. Потребує більше часу на написання: Створення сценаріїв може бути часу затратним. Вимагає навчання: Команді потрібно навчитися писати сценарії у відповідному форматі. Може стати надмірним: Для деяких проєктів описання всіх сценаріїв може бути непотрібним.

Інструменти для BDD на Python

- Behave: Популярний інструмент для BDD, який підтримує Gherkin-синтаксис.
- Lettuce: Ще один інструмент для BDD, схожий на Behave.
- pytest-bdd: Плагін для Pytest, який додає підтримку BDD.

Приклад використання Behave надається у Додатку Е п.3.

Приклад Python-реалізація для Behave надається у Додатку Е п.4.

Порівняння TDD та BDD надається в Табл.2.2

Табліця 2.2 - Порівняння TDD та BDD

Критерій	TDD	BDD
Опис тестів	Тестів у вигляді коду	Сценарії на зрозумілій мові
Ціль	Написання правильного коду	Спільне розуміння вимог
Співпраця	Розробники	Розробники, тестувальники, аналітики
Документація	Обмежена	Документована у вигляді сценаріїв

Таким чином TDD і BDD є важливими методологіями для підвищення якості програмного забезпечення. TDD забезпечує надійність коду за рахунок раннього тестування, тоді як BDD покращує розуміння вимог і забезпечує співпрацю між усіма учасниками проєкту. Використання цих методів разом може значно покращити процес розробки та тестування програмного забезпечення.

2.4 Вимоги до автоматизованої системи тестування

Автоматизована система тестування (AST) повинна відповідати низці вимог, щоб забезпечити ефективне і надійне виконання тестів. Вимоги до такої системи включають функціональні та нефункціональні аспекти, що забезпечують її ефективність і зручність у використанні.

1. Функціональні вимоги

- Підтримка різних типів тестування: Юніт-тестування. Інтеграційне тестування. Функціональне тестування. Системне тестування. Тестування інтерфейсу. Тестування продуктивності.

  1.2 Підтримка різних технологій
- Мови програмування: Підтримка популярних мов програмування, таких як Python, Java, JavaScript тощо.

- Фреймворки тестування: Можливість інтеграції з такими фреймворками, як Pytest, Selenium, JUnit, TestNG.
- Веб-додатки: Підтримка автоматизації тестування веб-додатків за допомогою інструментів типу Selenium, Cypress, Playwright.
- API-тестування: Підтримка інтеграційних тестів для RESTful та SOAP API.

1.3 Управління тестами та тестовими даними

- Створення тестових сценаріїв: Можливість легко створювати та оновлювати тестові сценарії.
- Управління тестовими даними: Інструменти для імпортування, збереження та обробки тестових даних.
- Тестова документація: Автоматичне генерування звітів і документів за результатами тестів.

1.4 Інтеграція з CI/CD

- Підтримка CI/CD-платформ: Можливість інтеграції з системами безперервної інтеграції та доставки, такими як Jenkins, GitHub Actions, GitLab CI/CD.
- Запуск тестів на вимогу: Здатність запускати тести при кожному коміті або за розкладом.

1.5 Гнучкість і масштабованість

- Конфігурація середовища: Можливість налаштування середовища тестування (локально, в хмарі, на сервері).
- Масштабування: Підтримка виконання тестів на різних пристроях і платформах.

2. Нефункціональні вимоги

2.1 Продуктивність і швидкість

- Швидкість виконання тестів: Здатність швидко запускати великі обсяги тестів, зокрема регресійне тестування.
- Ефективне використання ресурсів: Мінімальне використання оперативної пам’яті та процесора під час виконання тестів.

2.2 Надійність і відмовостійкість

- Стійкість до помилок: Система повинна бути здатна обробляти несподівані помилки та збою у тестах.
- Аналіз результатів: Автоматичне збереження журналів тестування та створення звітів для аналізу.

2.3 Легкість у використанні

- Інтуїтивно зрозумілий інтерфейс: Можливість легкого створення, запуску та моніторингу тестів навіть без глибоких знань у програмуванні.
- Документація і підтримка: Наявність детальної документації та технічної підтримки.

2.4 Безпека

- Захист тестових даних: Забезпечення конфіденційності тестових даних.
- Обмеження доступу: Можливість налаштування прав доступу до різних частин системи тестування.

2.5 Підтримка різних середовищ

- Крос-платформність: Підтримка виконання тестів на різних операційних системах (Windows, Linux, macOS).
- Підтримка хмарних середовищ: Можливість запуску тестів у хмарі (наприклад, за допомогою хмарних CI/CD сервісів).

2.6 Модульність і розширюваність

- Розширюваність: Можливість інтеграції з іншими інструментами та бібліотеками через плагіни або API.
- Модульна структура: Здатність розширювати функціонал системи тестування, додаючи нові модулі або плагіни.

Таким чином Автоматизована система тестування повинна відповідати низці функціональних і нефункціональних вимог для забезпечення ефективного, надійного та зручного тестування програмного забезпечення. Вибір відповідної

системи залежить від конкретних потреб проєкту, складності тестів та середовища розробки. Важливо враховувати інтеграцію з існуючими інструментами та можливість масштабування, щоб система могла адаптуватися до змін у процесі розробки.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ АВТОМАТИЧНОГО ТЕСТУВАННЯ

3.1. Методика практичної реалізація системи автоматичного тестування

Практична реалізація системи автоматичного тестування включає кілька основних етапів, від планування та вибору інструментів до написання тестів, інтеграції з процесами розробки та запуску тестування. Ось як можна реалізувати таку систему (Рис.3.1):

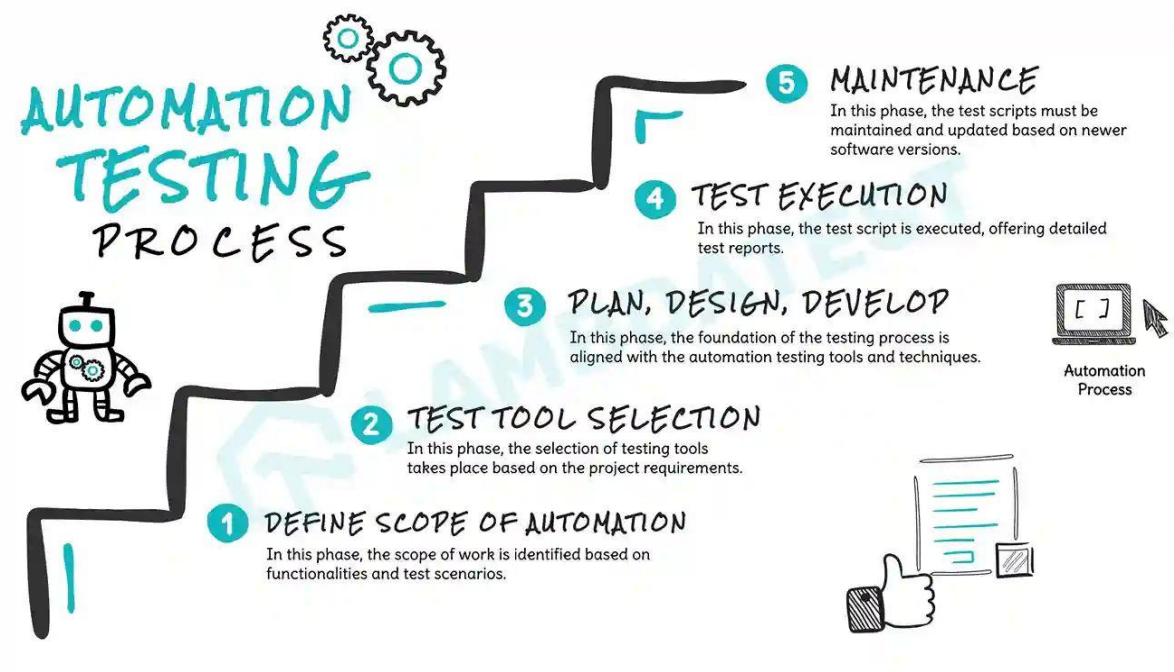


Рисунок 3.1 – Процес автоматичного тестування [37]

1. Планування та вибір інструментів

1.1 Визначення цілей тестування: Типи тестування: Визначити, які типи тестів потрібно автоматизувати (юніт-тестування, інтеграційне тестування,

функціональне тестування тощо). Вимоги до продуктивності: Оцінити, наскільки швидко потрібно виконувати тести та чи потрібна паралельна обробка.

1.2 Вибір інструментів:

- Мова програмування: Python є одним із найпопулярніших виборів завдяки своїй простоті та багатству бібліотек.
- Фреймворки для тестування: Вибір між Pytest, unittest, Selenium, Cypress, Playwright.
- Інструменти для CI/CD: Jenkins, GitHub Actions, GitLab CI/CD для автоматичного запуску тестів.
- Інструменти для управління тестами: Allure Report, TestRail для створення звітів та моніторингу результатів.

1.3 Налаштування середовища. Віртуальні середовища: Створення віртуальних середовищ (наприклад, за допомогою venv або virtualenv) для ізоляції залежностей. Управління залежностями: Використання requirements.txt або Pipfile для автоматичного встановлення потрібних бібліотек. Приклад встановлення Pytest надається у Додатку Ж п.1.

2. Написання тестів

2.1 Створення тестових функцій:

- Юніт-тести: Перевірка окремих функцій і методів.
- Інтеграційні тести: Тести, що перевіряють взаємодію між модулями.
- Функціональні тести: Оцінка роботи додатка з точки зору користувача.

Приклад юніт-тесту з Pytest надається у Додатку Ж п.2.

2.2 Використання фреймворків для UI-тестування

- Selenium: Для автоматизації веб-тестування.
- Playwright: Сучасний інструмент для автоматизації браузерів, з підтримкою JavaScript і Python.

Приклад використання Selenium для тестування веб-сайту надається у Додатку Ж п.3.

2.3 Використання CI/CD для автоматичного запуску тестів

- Jenkins: Налаштування пайплайну для запуску тестів після кожного коміту.
- GitHub Actions: Створення workflow-файлів для автоматичного запуску тестів у репозиторії.

Приклад GitHub Actions для запуску Pytest надається у Додатку Ж п.4.

3. Генерація звітів і аналіз результатів

3.1 Використання звітів

- Allure Report: Звіт про результати тестування з детальною інформацією про кожен тест, його статус і помилки.
- Pytest HTML Report: Створення звітів у форматі HTML.

Приклад використання Pytest для генерації звіту надається у Додатку Ж п.5.

3.2 Візуалізація результатів

- Графіки і діаграми: Використання додаткових бібліотек для створення графіків продуктивності.
- Інтеграція з системами моніторингу: Наприклад, інтеграція з Grafana або Kibana для реального часу аналізу даних.

4. Підтримка та обслуговування системи

4.1 Регулярне оновлення тестів

- Актуалізація тестів: Оновлення тестів у разі змін у функціоналі програми.
- Виправлення помилок: Виправлення тестів, що не проходять через зміни в коді.

4.2 Використання методологій TDD і BDD

- TDD: Написання тестів перед кодом для забезпечення високої якості.
- BDD: Використання сценаріїв на зрозумілій мові для кращої співпраці між розробниками та тестувальниками.

5. Переваги та виклики

5.1 Переваги автоматизованого тестування

- Підвищення ефективності: Зменшення часу, необхідного для проведення тестів.
- Підвищення надійності: Зниження ймовірності людських помилок.
- Забезпечення швидкого зворотного зв’язку: Раннє виявлення помилок у CI/CD-процесах, які показані на Рис.3.2.

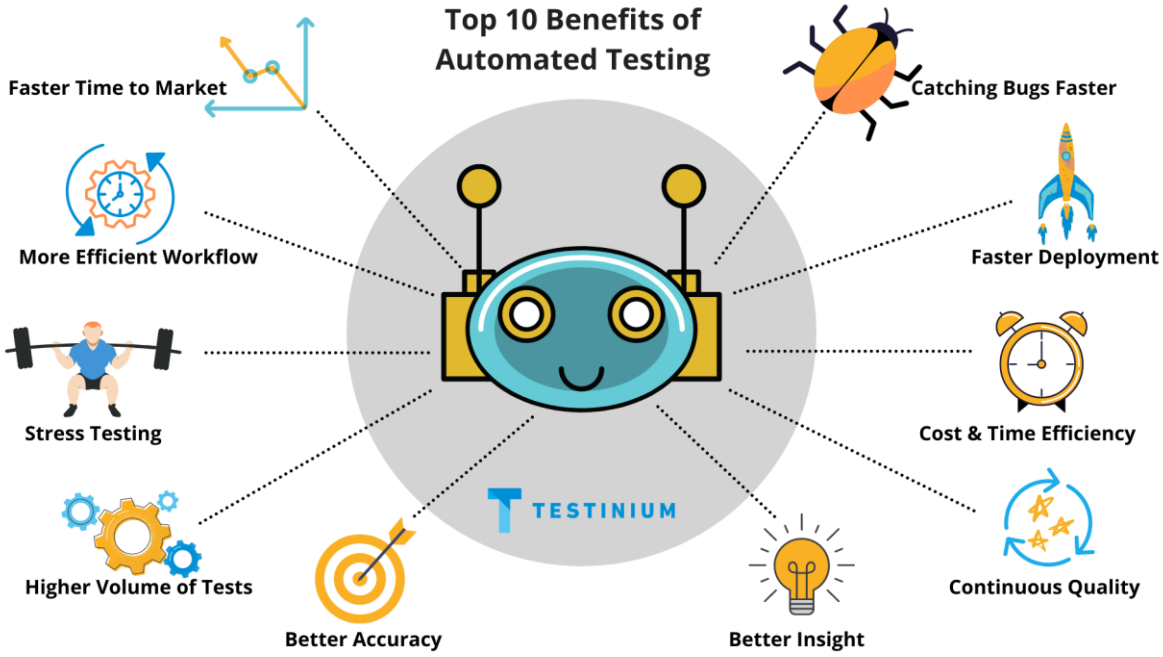


Рисунок 3.2 – Переваги автоматизованого тестування [38]

5.2 Виклики та проблеми

- Складність підтримки: Потреба в оновленні тестів і обслуговуванні фреймворків.
- Витрати часу на створення тестів: Створення і налаштування тестів може бути витратним за часом.
- Тестування складних сценаріїв: Може вимагати додаткових зусиль для інтеграційних і системних тестів.

Таким чином практична реалізація системи автоматичного тестування передбачає вибір відповідних інструментів, написання та інтеграцію тестів, а також налаштування процесів CI/CD для регулярного запуску тестів.

Використання звітів і аналітики дозволяє ефективно моніторити та підтримувати якість програмного забезпечення.

3.2 Рекомендації щодо побудови архітектури системи автоматичного тестування

Правильна архітектура системи автоматичного тестування (CAT) є ключем до ефективності процесів тестування та підтримки високої якості програмного забезпечення. Ось кілька основних рекомендацій для побудови архітектури CAT:

1. Визначення структури системи

1.1 Модульність та розширюваність

- Модульна архітектура: Побудуйте систему так, щоб її компоненти були незалежними, що спрощує тестування, обслуговування та розширення. Наприклад, окремі модулі для тестів, звітів, управління тестовими даними та інтеграцій з CI/CD.
- Гнучкість: Використовуйте дизайн, який дозволяє легко додавати нові функції та змінювати існуючі модулі без впливу на інші частини системи.

1.2 Вибір архітектурних патернів

- Патерн Page Object: Використовуйте для автоматизації веб-додатків, щоб зменшити дублювання коду та забезпечити зручність у підтримці тестів.
- Структура мікросервісів: Якщо система тестування інтегрується з іншими системами, розгляньте використання мікросервісної архітектури для кращої масштабованості та управління.

2. Організація тестових середовищ

2.1 Локальне та віддалене тестування

- Локальні середовища: Налаштування середовища для розробників та тестувальників на їхніх машинах для запуску тестів перед інтеграцією.
- Хмарні середовища: Використання хмарних платформ (наприклад, AWS, Azure, Google Cloud) для запуску тестів у масштабованих середовищах.

2.2 Контейнери та віртуалізація

- Docker: Використовуйте контейнеризацію для створення ізольованих середовищ для тестування, що спрощує розгортання та підтримку.
- Віртуальні машини: Використання VM для більш комплексного тестування, включаючи перевірку роботи на різних ОС.

3. Підбір інструментів і технологій

3.1 Вибір мов програмування

- Python: Найпопулярніша мова для автоматизації завдяки своїй простоті та великій кількості бібліотек, таких як Pytest, Selenium, Playwright.
- Java: Підходить для інтеграції з великими корпоративними системами.
- JavaScript: Ідеально підходить для тестування веб-додатків за допомогою таких фреймворків, як Cypress і Puppeteer.

3.2 Використання фреймворків

- Pytest: Потужний фреймворк для юніт-тестування та інтеграційного тестування.
- Allure Report: Для створення красивих та інформативних звітів.
- Selenium: Для автоматизації веб-тестування та перевірки інтерактивності інтерфейсу.(Рис.3.3)

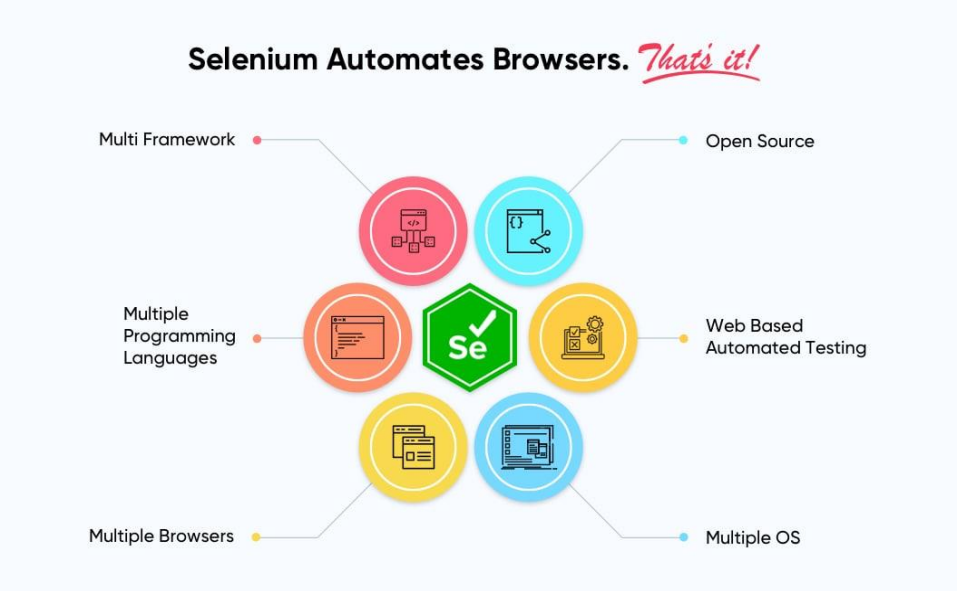


Рисунок 3.3 – Переваги Selenium[39]

3.3 Інструменти для CI/CD

- Jenkins: Гнучкий інструмент для налаштування пайплайнів тестування.
- GitHub Actions: Для автоматизації тестів у репозиторіях GitHub.
- GitLab CI/CD: Підходить для інтеграції з GitLab-репозиторіями.

4. Інтеграція з іншими системами

4.1 Інтеграція з системами управління проектами

- Jira: Використовуйте інтеграцію з Jira для відстеження багів і зв'язування тестів із завданнями.
- TestRail: Для управління тестовими планами, звітами та результатами тестування.

4.2 Підключення до систем моніторингу та аналітики

- Grafana: Для візуалізації метрик виконання тестів.
- Elasticsearch: Для збору та аналізу логів тестування.

5. Стратегії управління тестами

5.1 Розподіл тестів за типами

- Мікротести: Швидкі юніт-тести для перевірки базової функціональності.
- Інтеграційні тести: Перевірка інтеграцій між модулями.
- Кінцеві тести (End-to-End): Перевірка всього циклу взаємодії користувача з додатком.

5.2 Використання TDD і BDD

- TDD (Test-Driven Development): Підхід, при якому тести пишуться перед кодом, що забезпечує високу якість коду.
- BDD (Behavior-Driven Development): Використання сценаріїв на зрозумілій мові для кращої співпраці між розробниками та тестувальниками. Інструменти, такі як Cucumber або Behave, допомагають у впровадженні BDD.

6. Підтримка і масштабування

6.1 Управління тестовими даними

- Ізоляція тестів: Використання спеціальних тестових баз даних або інструментів для створення та скидання тестових даних.
- Mock-об'єкти: Використання бібліотек для створення mock-об'єктів, що симулюють поведінку реальних компонентів.

6.2 Масштабування тестування

- Паралельне виконання тестів: Використання фреймворків та інструментів, які підтримують запуск тестів паралельно, як-от Selenium Grid, Pytest-xdist.
- Контроль версій тестів: Використання систем контролю версій (наприклад, Git) для відстеження змін у тестах.

6.3 Регулярне оновлення тестів

- Рефакторинг тестів: Оновлення тестів у разі змін у функціоналі.
- Аудит тестів: Періодичне перегляд тестів для забезпечення їх актуальності та ефективності.

Таким чином створення архітектури системи автоматичного тестування вимагає ретельного планування, вибору правильних інструментів та інтеграції з іншими системами розробки та управління проектами. Рекомендується розробляти модульну, масштабовану і гнучку архітектуру, яка дозволить адаптуватися до змін у вимогах і технологіях.

3.3 Інструменти та бібліотеки для розробки системи автоматичного тестування

Для створення системи автоматичного тестування використовуються різноманітні інструменти та бібліотеки, залежно від потреб проєкту, типу тестів та середовища розробки. Ось найпоширеніші інструменти та бібліотеки, що застосовуються в процесі розробки систем автоматизованого тестування:

1. Інструменти для написання тестів

1.1 Фреймворки для юніт-тестування

- Pytest: Один із найбільш популярних і гнучких фреймворків для написання юніт-тестів, який підтримує багато корисних функцій, таких як параметризація тестів, плагіни та зручний синтаксис.
- Unittest: Вбудований фреймворк Python для юніт-тестування, який є частиною стандартної бібліотеки Python.
- Nose2: Розширення для Unittest, яке додає можливості автоматичного виявлення тестів і їх запуску.

Приклад тесту з Pytest надається у Додатку И п.1

1.2 Фреймворки для інтеграційного та функціонального тестування

- Selenium: Популярний інструмент для автоматизації веб-браузерів. Використовується для тестування інтерфейсу веб-додатків.
- Playwright: Сучасний інструмент для автоматизації браузерів, підтримує всі основні браузери і має потужні можливості для роботи з сучасними веб-додатками.
- Cypress: Ідеально підходить для інтеграційного та функціонального тестування сучасних веб-додатків. Має зручний API і підтримує тестування в реальному часі.

Приклад тесту з Selenium надається у Додатку И п.2

1.3 Інструменти для тестування API

- Requests: Бібліотека для здійснення HTTP-запитів, яку можна використовувати для тестування REST API.
- Locust: Інструмент для навантажувального тестування API.
- Postman: Платформа для тестування API з можливістю створення автоматичних тестів через скрипти.

Приклад тесту API за допомогою Requests надається у Додатку И п.3.

2. Інструменти для управління тестами та звітами

2.1 Генерація звітів

- Allure Report: Інструмент для створення інтерактивних звітів з інформацією про тестування, що дозволяє легко аналізувати результати.

- Pytest HTML Report: Плагін для Pytest, що створює HTML-звіти з результатами виконання тестів.
- ExtentReports: Бібліотека для створення звітів із докладною інформацією про виконання тестів.(Рис.3.4)

Рисунок 3.4 – Інструменти управління тестуванням [40]

Приклад використання Pytest для генерації звіту надається у Додатку И п.4.

2.2 Системи управління тестами

- TestRail: Платформа для управління тестовими планами та звітами.
- Zephyr: Плагін для Jira, який дозволяє інтегрувати управління тестами з процесом розробки.
- Xray: Інструмент для управління тестуванням у Jira.

3. Інструменти для CI/CD

3.1 Платформи для автоматизації процесів CI/CD

- Jenkins: Один із найпопулярніших інструментів для автоматизації процесів CI/CD, який підтримує різні плагіни та інтеграції з іншими системами.
- GitHub Actions: Інструмент для автоматизації CI/CD процесів у репозиторіях GitHub.
- GitLab CI/CD: Інтегрований інструмент CI/CD для репозиторіїв GitLab.
- CircleCI: Платформа для CI/CD, яка добре інтегрується з GitHub та Bitbucket.

Приклад конфігурації GitHub Actions для запуску Pytest надається у Додатку И п.5.

4. Інструменти для управління середовищем тестування

Інструменти для управління середовищем тестування показано на Рис.3.5.

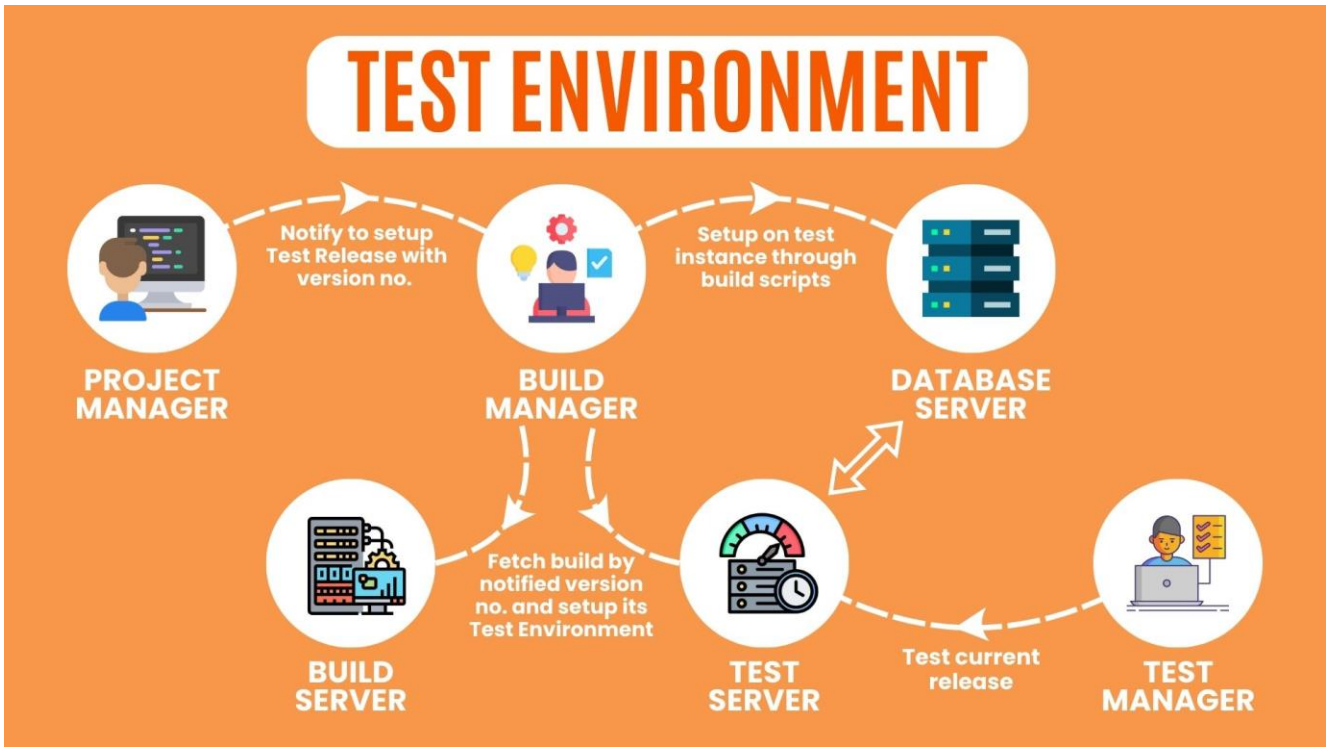


Рисунок 3.5 – Визначення тестового середовища [41]

4.1 Контейнеризація

- Docker: Використовується для створення контейнерів, що дозволяє створити ізольоване середовище для запуску тестів.
- Kubernetes: Підходить для масштабованих систем тестування, що потребують автоматичного управління контейнерами.

4.2 Віртуалізація середовищ

- Vagrant: Інструмент для створення віртуальних середовищ для тестування.
- Virtualenv: Використовується для створення ізольованих середовищ Python.

5. Інші корисні бібліотеки

5.1 Для мокінгу та створення тестових даних

- Mockito: Бібліотека для створення мок-об'єктів у Java. Для Python можна використовувати unittest.mock.
- Faker: Бібліотека для генерації фейкових даних для тестування (імена, адреси, електронні листи тощо).

5.2 Для аналізу та моніторингу

- Grafana: Для візуалізації даних про виконання тестів.
- Prometheus: Для збору метрик і аналізу продуктивності тестів.

Таким чином для створення ефективної системи автоматичного тестування необхідно поєднувати різні інструменти та бібліотеки, залежно від вимог проєкту. Обираючи інструменти, варто орієнтуватися на масштабованість, зручність інтеграції з іншими системами та підтримку сучасних стандартів тестування.

3.4 Розробка та реалізація тестових сценаріїв системи автоматичного тестування

Розробка тестових сценаріїв є важливим етапом у створенні системи автоматичного тестування. Якісно написані тестові сценарії забезпечують ефективну перевірку функціональності програмного забезпечення та дозволяють швидко виявляти і виправляти помилки. Ось кроки, які слід дотримуватися:

1. Визначення цілей тестування

1.1 Визначення типів тестів

- Юніт-тести: Перевірка окремих модулів або функцій.
- Інтеграційні тести: Перевірка взаємодії між модулями.
- Функціональні тести: Перевірка функціоналу програми відповідно до вимог.
- Системні тести: Перевірка програмного забезпечення в цілому.
- Кінцеві тести (End-to-End): Перевірка повного сценарію роботи користувача.

1.2 Визначення метрик успіху

- Кількість пройдених тестів.
- Час виконання тестів.
- Відсоток покриття коду тестами.

2. Створення тестових сценаріїв

2.1 Структура тестового сценарію. Кожен тестовий сценарій повинен мати чітку структуру, яка включає:

- Назву тесту: Опис того, що тест перевіряє.
- Передумови: Що необхідно підготувати перед запуском тесту.
- Кроки виконання: Покроковий опис дій, які потрібно виконати під час тесту.
- Очікуваний результат: Те, що має відбутися, якщо тест пройде успішно.

Приклад тестового сценарію надається у додатку И п.6.

2.2 Використання BDD для створення тестових сценаріїв.

Behavior-Driven Development (BDD) дозволяє писати тестові сценарії, що використовують природню мову, яку легко розуміють всі учасники команди (розробники, тестувальники, бізнес-аналітики). Приклад сценарію BDD за допомогою Gherkin надається у Додатку И п. 7.

2.3 Використання інструментів для автоматизації створення тестів

- Cucumber: Для автоматизації тестів BDD.

- Behave: Python-інструмент для BDD, що підтримує написання сценаріїв на Gherkin.
- Pytest-bdd: Плагін для Pytest, що дозволяє писати BDD-сценарії на основі Gherkin.

3. Реалізація тестових сценаріїв

3.1 Написання тестів за допомогою фреймворків. Використовуйте обраний фреймворк для автоматизації тестування, щоб реалізувати тестові сценарії у вигляді коду. Приклад тесту з використанням Pytest наданій у Додатку И п.8.

3.2 Використання фреймворків для тестування веб-додатків

- Selenium: Автоматизація тестування веб-додатків.
- Playwright: Модерніший інструмент, який підтримує роботу з усіма основними браузерами та забезпечує покращену швидкість тестування.
- Cypress: Підходить для інтеграційного та функціонального тестування сучасних веб-додатків.

4. Виконання тестів і аналіз результатів

4.1 Запуск тестів у локальному середовищі. Запустіть тести локально за допомогою команд у терміналі (наприклад, pytest).

4.2 Виконання тестів на CI/CD. Налаштуйте запуск тестів у процесах CI/CD за допомогою Jenkins, GitHub Actions, GitLab CI/CD або інших платформ. Приклад конфігурації для GitHub Actions надається у Додатку И п.9.

4.3 Аналіз результатів тестування

- Використовуйте інтеграцію з інструментами для аналізу звітів, такими як Allure або ExtentReports, щоб отримати зручні звіти про результати тестування.
- Налаштуйте систему сповіщень, щоб автоматично інформувати команду про результати тестів (наприклад, за допомогою електронної пошти або Slack).

5. Оптимізація і підтримка тестових сценаріїв

5.1 Регулярне оновлення тестів

- Оновлюйте тести у разі змін у функціоналі додатку.
- Використовуйте рефакторинг коду тестів для підвищення їх читаємості та підтримуваності.

5.2 Виявлення та усунення помилок у тестах

- Використовуйте дебаггер або логування для аналізу помилок, що виникають під час виконання тестів.

5.3 Використання зворотного зв’язку

- Проводьте періодичні перегляди тестових сценаріїв за участю команди розробників і тестувальників для покращення якості тестування.

Таким чином розробка та реалізація тестових сценаріїв є критично важливою частиною системи автоматичного тестування. Вона потребує ретельного планування, вибору відповідних інструментів і дотримання найкращих практик у написанні тестів. Використання сучасних фреймворків і автоматизація тестів забезпечують високий рівень покриття та якість програмного забезпечення.

3.5 Інтеграція автоматизованого тестування в CI/CD в системах автоматичного тестування

Інтеграція автоматизованого тестування в CI/CD (Continuous Integration/Continuous Deployment) є важливою складовою сучасного процесу розробки програмного забезпечення. Цей підхід дозволяє забезпечити якість і стабільність програмного продукту, автоматизуючи перевірку змін у коді і забезпечуючи швидке виявлення помилок. Ось як реалізується інтеграція автоматизованого тестування в CI/CD:

1. Основи CI/CD у системах автоматичного тестування

CI/CD — це методології, які автоматизують процеси інтеграції та розгортання програмного забезпечення. CI включає регулярне злиття змін у центральну гілку, з автоматичним запуском тестів для перевірки нових змін. CD

забезпечує автоматичне розгортання на середовище розробки, тестування чи навіть у продакшн.

2. Основні етапи інтеграції автоматизованого тестування в CI/CD

2.1 Налаштування CI/CD-платформи

Вибір інструментів для CI/CD залежить від середовища розробки та потреб проєкту. Найпопулярніші платформи включають:

- Jenkins: Потужна та гнучка платформа з підтримкою безлічі плагінів.
- GitHub Actions: Ідеально інтегрована з GitHub, дозволяє налаштувати CI/CD прямо в репозиторії.
- GitLab CI/CD: Інтегрована з GitLab, дозволяє налаштовувати та запускати пайплайни безпосередньо в репозиторії.
- CircleCI: Підходить для швидкої інтеграції та масштабування CI/CD процесів.

2.2 Налаштування середовища для автоматизованих тестів

- Контейнери Docker: Використання контейнерів для забезпечення стабільного середовища виконання тестів незалежно від середовища розробки.
- Віртуальні машини: Можливість використання VM для запуску тестів у середовищах, схожих на продакшн.
- Налаштування середовищ у хмарі: Використання хмарних платформ, таких як AWS, Azure чи Google Cloud, для запуску тестів.

2.3 Підготовка тестових сценаріїв для інтеграції. Тестові сценарії мають бути структуровані так, щоб їх можна було легко інтегрувати у пайплайн CI/CD:

- Розподіл тестів: Можливість запуску лише певної категорії тестів (наприклад, тільки юніт-тести або тільки інтеграційні).
- Підготовка тестових даних: Автоматизація створення тестових даних перед виконанням тестів.

2.4 Створення конфігураційних файлів для CI/CD. Конфігураційні файли для платформ CI/CD, наприклад, .github/workflows/main.yml для GitHub Actions або

Jenkinsfile для Jenkins. Приклад GitHub Actions для запуску тестів з Pytest надається у додатку И п.10:

3. Автоматизація тестування на кожному етапі CI/CD

3.1 Запуск юніт-тестів при кожному коміті. Ціль: Перевірка, чи не зламав новий код існуючу функціональність. Використання: Запуск за допомогою простих команд pytest, unittest, або nose.

3.2 Інтеграційне тестування при злитті гілок. Ціль: Перевірка інтеграції між різними модулями після злиття змін у гілку. Використання: Може включати запуск тестів, що перевіряють взаємодію між модулями за допомогою таких інструментів, як Selenium чи Playwright.

3.3 Функціональне тестування перед релізом. Ціль: Перевірка, чи відповідає кінцевий продукт вимогам користувачів. Використання: Запуск інтеграційних тестів і тестів типу E2E.

3.4 Створення звітів і моніторинг результатів. Використання інструментів, таких як Allure, ExtentReports, або стандартні звіти Pytest, для створення зрозумілих звітів.Інтеграція CI/CD з Slack, Email або іншими системами для сповіщення про результати тестування.

4. Виклики та найкращі практики інтеграції автоматизованого тестування в CI/CD

4.1 Виклики. Витрати часу на налаштування: Початкова конфігурація та підготовка тестів можуть займати значний час.Витрати ресурсів: Запуск великої кількості тестів може вимагати значних обчислювальних ресурсів. Нестабільні тести: Іноді автоматизовані тести можуть бути нестабільними, що призводить до помилкових тривог.

4.2 Найкращі практики. Ізоляція тестів: Тести повинні бути ізольовані, щоб не залежати один від одного. Оптимізація тестів: Використання технік паралельного виконання тестів для зменшення часу виконання. Регулярне оновлення: Зміни в коді і нові функції повинні супроводжуватись оновленням

тестів. Створення тестової документації: Опис тестів і їх призначення полегшує підтримку та масштабування.

Таким чином інтеграція автоматизованого тестування в CI/CD є важливим кроком до забезпечення якості програмного продукту. Це дозволяє забезпечити швидке виявлення помилок, знижує ймовірність помилок у продакшн-середовищі і підтримує високий рівень стабільності програми. Налаштування інтеграції потребує уважного вибору інструментів і налаштування процесів, але це значно покращує процес розробки та випуску продукту.

ВИСНОВКИ

Автоматичне тестування є невід'ємною складовою сучасного процесу розробки програмного забезпечення. Воно забезпечує виявлення помилок на ранніх етапах, що знижує витрати на виправлення дефектів.

Python є однією з найзручніших мов для створення систем автоматичного тестування завдяки її синтаксичній простоті, великій кількості бібліотек (наприклад, unittest, pytest, selenium) та активній спільноті розробників. Впровадження системи автоматичного тестування дозволяє значно зменшити ручну працю, підвищити точність тестування та забезпечити масштабованість процесу. Системи автоматичного тестування, розроблені на Python, легко інтегруються у CICD-процеси, сприяючи швидкому випуску нових версій програмного забезпечення.

У межах виконання дипломної роботи "Система автоматизованого тестування на основі мови програмування Python" були досягнуті поставлені цілі та завдання. Проведено детальний аналіз сучасних підходів до автоматизованого тестування, вивчено існуючі інструменти, такі як pytest, unittest, Selenium, а також методології TDD та BDD. Розроблений прототип автоматизованої системи тестування, що демонструє ефективність Python для створення надійних і гнучких тестових сценаріїв.

Результати роботи підтвердили, що використання автоматизованого тестування дозволяє суттєво підвищити якість програмного забезпечення, скоротити час на виявлення помилок та інтегрувати тестування у процес безперервної розробки (CI/CD). Створений прототип успішно продемонстрував можливості інтеграції автоматичних тестів у конвеєри GitHub Actions, забезпечуючи своєчасне виявлення дефектів під час розробки.

Незважаючи на досягнуті результати, автоматизація тестування має свої виклики, зокрема потребу в підтримці тестових сценаріїв і виборі оптимальних інструментів для конкретного проєкту. Враховуючи ці аспекти, подальші

дослідження можуть бути спрямовані на впровадження додаткових інструментів для нефункціонального тестування, таких як тестування продуктивності, безпеки та мобільних додатків.

Робота показала перспективність використання Python для автоматизації тестування, особливо завдяки його простоті, масштабованості та широкій екосистемі бібліотек. Розроблений підхід може бути рекомендований для впровадження в реальні проєкти, спрямовані на підвищення якості та продуктивності програмного забезпечення.

Подальший розвиток систем автоматичного тестування можливий завдяки застосуванню штучного інтелекту та машинного навчання для прогнозування помилок і автоматичного генерування тестових сценаріїв.

ПЕРЕЛІК ПОСИЛАНЬ

1. Future of Software Testing:. URL: https://testlio.com/blog/test-automation-tools/ / (дата звернення: 29.11.2024).
2. What is Unit Testing?:. URL: https://aws.amazon.com/what-is/unit-testing / / (дата звернення: 29.11.2024).
3. What is integration testing (I&T)?:. URL: https://www.techtarget.com/searchsoftwarequality/definition/integration-testing / (дата звернення: 29.11.2024).
4. Functional Testing – Software Testing/ URL: https://www.geeksforgeeks.org/software-testing-functional-testing / / (дата звернення: 29.11.2024).
5. Types of Automated Testing Explained Within Test Strategy/ URL: https://testomat.io/blog/types-of-automated-testing-explained-within-test-strategy / (дата звернення: 29.11.2024)/
6. Міжнародна науково-практична конференція «Сучасні аспекти діджиталізації та інформатизації в програмній та комп'ютерній інженерії». Збірник тез. – К.: ДУТ,2023. /URL: https://duikt.edu.ua/uploads/n_11337_64054605.pdf (дата звернення: 29.11.2024).
7. Java Unit Testing: методики, поняття, практика/URL: https://javarush.com/ua/groups/posts/uk.2500.java-unit-testing-metodiki-ponjattja-praktika(дата звернення: 29.11.2024).
8. Теорія тестування від А до Я. //URL: https://ilarionhalushka.github.io/ua/testing-theory(дата звернення: 29.11.2024).
9. Автоматизація тестування веб-додатків: інструменти та стратегії//URL: https://redstone.agency/blog/avtomatyzatsiia-testuvannia-veb-dodatkiv-instrumenty-ta-stratehii/(дата звернення: 29.11.2024).

10. Статичне та динамічне тестування: основні відмінності та практичні приклади //URL: https://mate.academy/dev-blog/qa/static-dynamic-testing/(дата звернення: 29.11.2024).

11. Python-IIT-Kanpur-Winter-Programme-Questions//URL: https://github.com/Sahilll94/Python-IIT-Kanpur-Winter-Programme-Questions/blob/main/To%20fine%20factorial%20of%20a%20(дата звернення: 29.11.2024).

12. Programming Exercise #10_1: Create a Python source file that will include the following two line//URL:https://www.chegg.com/homework-help/questions-and-answers/1-programming-exercise-101-create-python-source-file-named-ex101py-include-following-two-l-q71111387(дата звернення: 29.11.2024).

13. Тестування програмного забезпечення: типи, види та застосування. //URL: https://foxminded.ua/testuvannia-prohramnoho-zabezpechennia/(дата звернення: 29.11.2024).

14. Top 8 Python Testing Frameworks in 2024// URL: https://www.browserstack.com/guide/top-python-testing-frameworks/ (дата звернення: 29.11.2024).

15. The framework used in automation software testing// URL: https://www.javatpoint.com/automation-testing (дата звернення: 29.11.2024).

16. Python Cross-Browser Testing// URL: https://www.bloggersranking.com/python-cross-browser-testing/ (дата звернення: 29.11.2024).

17. PYTHON UNIT TESTING By Ridhi Ardeshna / May 22, 2024// URL: https://coderspacket.com/posts/python-unit-testing /(дата звернення: 29.11.2024).

18. How do I run multiple Python test cases in a loop? // URL: https://pythonhint.com/post/1379503304336138/how-do-i-run-multiple-python-test-cases-in-a-loop/(дата звернення: 29.11.2024).

19. Автоматизація тестування з Selenium: як підвищити якість коду// URL: https://itproger.com/ua/news/avtomatizatsiya-testirovaniya-s-selenium-kak-povisit-kachestvo-vashego-ko#google_vignette /(дата звернення: 29.11.2024).

20. CI-CD-Pipeline // URL: https://github.com/mdalmaruf/CI-CD-Pipeline/(дата звернення: 29.11.2024).

21. Automating Every Aspect of Your Python Project// URL: https://dev.to/martinheinz/automating-every-aspect-of-your-python-project-f5f/(дата звернення: 29.11.2024).

22. How to Use GitHub Actions to Automate Open-Source Projects// URL: https://www.freecodecamp.org/news/automate-open-source-projects-with-github-actions//(дата звернення: 29.11.2024).

23. Python:813 (in "parametrize") raises "TypeError: object of type 'int' has no len()" in some cases #638// URL: https://github.com/pytest-dev/pytest/issues/638/(дата звернення: 29.11.2024).

24. Unable to use Selenium Webdriver. Getting two exceptions// URL: https://stackoverflow.com/questions/76461596/unable-to-use-selenium-webdriver-getting-two-exceptions /(дата звернення: 29.11.2024).

25. Unit Testing in Python, July 17, 2024// URL: https://www.erikrasin.io/blog/unit-testing/(дата звернення: 29.11.2024).

26. Unit Testing – Software Testing/URL: https://www.geeksforgeeks.org/unit-testing-software-testing / (дата звернення: 29.11.2024).

27. Unit Test //URL: https://martinfowler.com/bliki/UnitTest.html (дата звернення: 29.11.2024).

28. Mock Testing / URL: https://devopedia.org/mock-testing / (дата звернення: 29.11.2024).

29. A Guide to Testing in Python: `unittest` and `pytest`/ URL: https://www.pyquantnews.com/free-python-resources/a-guide-to-testing-in-python-unittest-and-pytest (дата звернення: 29.11.2024).

30. Python Unittest Tutorial | Unit Testing in Python using unittest Framework`/ URL: https://codeease.net/programming/python/Unit-test-utilities /(дата звернення: 29.11.2024)

31. Python Unittest Tutorial | Unit Testing in Python using unittest Framework Last Updated : 22 Apr, 2024 `/ URL: https://www.geeksforgeeks.org/unit-testing-python-unittest //(дата звернення: 29.11.2024).

32. How to write test cases correctly`/ URL: https://labex.io/tutorials/python-how-to-write-test-cases-correctly-418595 /(дата звернення: 29.11.2024).

33. How to Set Up GitHub Actions for Continuous Integration/ URL: https://dev.to/angelotheman/how-to-set-up-github-actions-for-continuous-integration-2054 /(дата звернення: 29.11.2024).

34. Automated API Testing Best Practices Including AI Testing/ URL: https://www.qualiti.ai/ai-testing/automated-api-testing /(дата звернення: 29.11.2024).

35. How to Setup a CI/CD Pipeline with GitHub Actions and AWS/ URL: https://www.freecodecamp.org/news/how-to-setup-a-ci-cd-pipeline-with-github-actions-and-aws //(дата звернення: 29.11.2024)

36. Decoding Testing Paradigms: TDD vs. BDD for Software Excellence/ URL: https://katalon.com/resources-center/blog/tdd-vs-bdd (дата звернення: 29.11.2024).

37. What is Automation Testing: Benefits, Strategy and Best Practices/URL: https://www.lambdatest.com/learning-hub/automation-testing (дата звернення: 29.11.2024).

38. Benefits of Automated Testing/ URL: https://testinium.com/blog/top-10-benefits-of-automated-testing-2/ (дата звернення: 29.11.2024).

39. Top 10 Automation Testing Tools for 2024 /URL: https://www.netsolutions.com/insights/top-10-automation-testing-tools/ (дата звернення: 29.11.2024).

40. Test Management Tools/ URL: https://testproeg.com/top-test-management-tools / (дата звернення: 29.11.2024).

41. The Crucial Role of Test Environment Setup in Software Testing/ URL: https://www.frugaltesting.com/blog/the-crucial-role-of-test-environment-setup-in-software-testing / (дата звернення: 29.11.2024).

Додаток А.

Приклад автоматичного тестування студентського коду за допомогою pytest:

Завдання: Написати функцію для обчислення факторіалу числа.

```
python
Copy code
# Student's solution
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Тест:

```
python
Copy code
import pytest

# Test cases
@pytest.mark.parametrize("input_value, expected", [
    (0, 1),
    (1, 1),
    (5, 120),
    (10, 3628800)
])
def test_factorial(input_value, expected):
    from student_solution import factorial
    assert factorial(input_value) == expected
```

Результат: При виконанні тесту будуть перевірені всі вхідні дані. У разі помилки, система повідомить про невідповідність.

Додаток Б.

Приклади Pipeline, Workflow,

1. Приклад Pipeline

```
groovy
Copy code
pipeline {
    agent any
    stages {
        stage('Clone Repository') {
            steps {
                git 'https://github.com/username/repository.git'
            }
        }
        stage('Build') {
            steps {
                sh 'make build'
            }
        }
        stage('Test') {
            steps {
                sh 'pytest tests/'
            }
        }
        stage('Deploy') {
            steps {
                sh './deploy.sh'
            }
        }
    }
}
```

2. Приклад Workflow:

```
yaml
name: CI Pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3
```

```
- name: Set up Python
  uses: actions/setup-python@v4
  with:
    python-version: 3.9

- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install -r requirements.txt

- name: Run tests
  run: pytest tests/
```

Додаток В.

Фреймворки та бібліотеки Python для автоматизації тестування

Pytest

- **Призначення:** модульне тестування, інтеграційне тестування, тестування API.
- **Переваги:** простота, параметризація тестів, потужна система плагінів.
- **Особливості:** підтримує асинхронне тестування та інтеграцію з іншими інструментами.

Приклад:

```
python
import pytest

def add(a, b):
    return a + b

@pytest.mark.parametrize("a, b, expected", [(1, 2, 3), (-1, 1, 0)])
def test_add(a, b, expected):
    assert add(a, b) == expected
```

Unittest

- **Призначення:** стандартний модуль для модульного тестування.
- **Переваги:** вбудований у Python, підходить для невеликих проєктів.
- **Недоліки:** менш гнучкий у порівнянні з Pytest.

Приклад:

```
python
import unittest

class TestMathOperations(unittest.TestCase):
    def test_add(self):
        self.assertEqual(1 + 1, 2)
```

Selenium

- **Призначення:** автоматизація тестування веб-додатків.
- **Переваги:** кросбраузерна підтримка.
- **Недоліки:** потребує значних ресурсів.

Приклад:

```
python
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://example.com")
assert "Example Domain" in driver.title
driver.quit()
```

Locust

- **Призначення:** навантажувальне тестування веб-додатків.
- **Переваги:** легкий у використанні, добре масштабується.
- **Особливості:** підтримка розподіленого тестування.

Приклад:

```
python
from locust import HttpUser, task

class WebsiteTestUser(HttpUser):
    @task
    def load_homepage(self):
        self.client.get("/")
```

Python для тестування API

Python є ідеальним вибором для тестування API завдяки бібліотекам **requests** і **pytest**:

- **Requests:** проста у використанні бібліотека для роботи з HTTP-запитами.
- **Pytest:** використовується для створення сценаріїв тестування API.

Приклад тестування API:

```
python
import requests

def test_api_status():
    response =
requests.get("https://jsonplaceholder.typicode.com/posts")
    assert response.status_code == 200
```

Python для тестування баз даних

Python підтримує багато драйверів для роботи з базами даних:

- **SQLite:** вбудована підтримка.
- **SQLAlchemy:** ORM для інтерактивної роботи з базами даних.

Приклад:

```
python
import sqlite3

def test_database():
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE test (id INTEGER PRIMARY KEY, name
TEXT)")
    cursor.execute("INSERT INTO test (name) VALUES ('Alice')")
    conn.commit()
    cursor.execute("SELECT name FROM test WHERE id = 1")
    result = cursor.fetchone()
    assert result[0] == "Alice"
    conn.close()
```

Додаток Г.

Інструменти для юніт-тестування на Python

2.1 Unittest

Приклад:

```
python
import unittest

def add(a, b):
    return a + b

class TestMathOperations(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)

if __name__ == '__main__':
    unittest.main()
```

2.2 Pytest

Приклад:

```
python
import pytest

def add(a, b):
    return a + b

@pytest.mark.parametrize("a, b, expected", [(1, 2, 3), (-1, 1, 0),
(0, 0, 0)])
def test_add(a, b, expected):
    assert add(a, b) == expected
```

Запуск тестів:

```
bash
Copy code
pytest test_file.py
```

2.3 Приклад використання `unittest.mock`:

```
python
from unittest.mock import MagicMock

def fetch_data_from_api(api_client):
    return api_client.get_data()
```

```
def test_fetch_data_from_api():
    mock_client = MagicMock()
    mock_client.get_data.return_value = {"key": "value"}
    result = fetch_data_from_api(mock_client)
    assert result == {"key": "value"}
```

2.4 Інтеграція юніт-тестування з CI/CD

```
yaml
name: Run Unit Tests

on:
  push:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: 3.9

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest

      - name: Run tests
        run: pytest tests/
```

Інструменти для інтеграційного тестування на Python

2.1 Pytest:

Приклад:

```
python
import pytest
import requests

@pytest.fixture
def api_base_url():
    return "https://jsonplaceholder.typicode.com"

def test_get_posts(api_base_url):
    response = requests.get(f"{api_base_url}/posts")
    assert response.status_code == 200
    assert isinstance(response.json(), list)
```

2.2 Unittest

Приклад:

```
python
import unittest
from unittest.mock import patch

def fetch_user_data(api_client, user_id):
    return api_client.get_user(user_id)

class TestIntegration(unittest.TestCase):
    @patch("api_client.ApiClient.get_user")
    def test_fetch_user_data(self, mock_get_user):
        mock_get_user.return_value = {"id": 1, "name": "John Doe"}
        result = fetch_user_data(mock_get_user, 1)
        self.assertEqual(result, {"id": 1, "name": "John Doe"})
```

2.4 Docker

Приклад використання Docker Compose для тестування мікросервісів:

```
yaml
version: '3.8'

services:
  app:
    build: .
    ports:
      - "5000:5000"
  db:
    image: postgres:14
```

```
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
```

3.1 Інструменти на основі SQL.

Приклад:

```
python
import pytest
from sqlalchemy import create_engine

@pytest.fixture
def db_connection():
    engine = create_engine("sqlite:///:memory:")
    connection = engine.connect()
    yield connection
    connection.close()

def test_insert_data(db_connection):
    result = db_connection.execute("CREATE TABLE test (id INTEGER
PRIMARY KEY, name TEXT)")
    assert result is not None
```

4. Mocking у інтеграційному тестуванні

Приклад: використання responses для mocking HTTP-запитів:

```
python
import responses
import requests

@responses.activate
def test_mocked_api():
    responses.add(
        responses.GET,
        "https://example.com/api",
        json={"message": "success"},
        status=200
    )
    response = requests.get("https://example.com/api")
    assert response.json() == {"message": "success"}
```

5. Інтеграція з CI/CD

Приклад інтеграції в GitHub Actions:

```
yaml
name: Integration Tests

on:
  push:
```

```
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest
    services:
      db:
        image: postgres:14
        env:
          POSTGRES_USER: user
          POSTGRES_PASSWORD: password
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: 3.9
      - name: Install dependencies
        run: pip install pytest requests
      - name: Run integration tests
        run: pytest integration_tests/
```

Додаток Д

Інструменти для функціонального тестування на Python

2.1 Selenium.

Приклад використання Selenium:

```
python
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://example.com")

# Перевірка наявності елементу на сторінці
assert "Example Domain" in driver.title

element = driver.find_element(By.XPATH, "//h1")
assert element.text == "Example Domain"

driver.quit()
```

2.2 Pytest

Приклад використання Pytest для тестування API:

```
python
Copy code
import pytest
import requests

def test_get_user():
    response =
requests.get("https://jsonplaceholder.typicode.com/users/1")
    assert response.status_code == 200
    assert response.json()['name'] == "Leanne Waters"
```

Додаток Е

1. Приклад TDD з Pytest

```
python
# Спочатку пишемо тест, який не проходить
def test_addition():
    assert add(2, 3) == 5  # Очікуваний результат

# Потім пишемо мінімальний код для проходження тесту
def add(a, b):
    return a + b
```

2. Приклад сценарію BDD:

- Given — стан перед виконанням дії.
- When — дія, яка виконується.
- Then — очікуваний результат.

```
gherkin
Feature: Login functionality

  Scenario: Successful login with valid credentials
    Given a user is on the login page
    When the user enters valid credentials and submits the form
    Then the user should be redirected to the dashboard
```

3. Приклад використання Behave:

```
gherkin
Feature: Search functionality

  Scenario: Searching for a product
    Given the user is on the search page
    When the user enters "laptop" in the search bar and submits
    Then the search results should display items related to "laptop"
```

4. Приклад Python-реалізація для Behave:

```
python
from behave import given, when, then

@given('the user is on the search page')
def step_given_user_on_search_page(context):
    pass  # Код для підготовки сторінки

@when('the user enters "{query}" in the search bar and submits')
def step_when_user_enters_query(context, query):
    context.query = query  # Код для введення запиту
```

```
@then('the search results should display items related to
"{query}"')
def step_then_search_results(context, query):
    assert "laptop" in context.query  # Перевірка результатів
```

Додаток Ж

Методика практичної реалізація системи автоматичного тестування

1.Приклад встановлення Pytest:

```
bash
Copy code
pip install pytest
```

2. Приклад юніт-тесту з Pytest:

```
python
# Функція, яку тестуємо
def add(a, b):
    return a + b

# Тестова функція
def test_add():
    assert add(2, 3) == 5
```

3. Приклад використання Selenium для тестування веб-сайту:

```
python
from selenium import webdriver

driver = webdriver.Chrome()  # Запуск браузера Chrome
driver.get("https://example.com")
assert "Example Domain" in driver.title
driver.quit()
```

4. Приклад GitHub Actions для запуску Pytest:

```
yaml
name: Python CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: '3.9'
```

```
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run tests
      run: |
        pytest
```

5. Приклад використання Pytest для генерації звіту:

```
bash
Copy code
pytest --html=report.html
```

1. Приклад тесту з Pytest:

```
python
Copy code
def test_addition():
    assert 1 + 1 == 2
```

2. Приклад тесту з Selenium:

```
python
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://example.com")
assert "Example Domain" in driver.title
driver.quit()
```

3. Приклад тесту API за допомогою Requests:

```
python
Copy code
import requests

response = requests.get('https://api.example.com/data')
assert response.status_code == 200
```

4. Приклад використання Pytest для генерації звіту:

```
bash
Copy code
pytest --html=report.html
```

5. Приклад конфігурації GitHub Actions для запуску Pytest:

```
yaml
name: Python CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v3
      with:
```

```
        python-version: '3.9'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run tests
      run: |
        pytest
```

6. Приклад тестового сценарію:

```
plaintext
Назва тесту: Перевірка правильності логіну користувача

Передумови: Користувач зареєстрований у системі.
Кроки виконання:
1. Перейти на сторінку входу.
2. Ввести правильний логін і пароль.
3. Натиснути кнопку "Увійти".
Очікуваний результат: Користувач успішно потрапляє на домашню
сторінку.
```

7. Приклад сценарію BDD за допомогою Gherkin:

```
gherkin
Feature: Логін користувача
  As a registered user
  I want to log in to the system
  So that I can access my account

  Scenario: Успішний логін
    Given користувач знаходиться на сторінці входу
    When він вводить правильний логін і пароль
    And натискає кнопку "Увійти"
    Then він потрапляє на домашню сторінку
```

8. Приклад тесту з використанням Pytest:

```
python
import pytest
from selenium import webdriver

@pytest.fixture
def driver():
    driver = webdriver.Chrome()
    driver.get("https://example.com/login")
    yield driver
    driver.quit()
```

```
def test_login(driver):
    driver.find_element("name", "username").send_keys("testuser")
    driver.find_element("name", "password").send_keys("password123")
    driver.find_element("id", "loginButton").click()
    assert "Home Page" in driver.title
```

9. Приклад конфігурації для GitHub Actions:

```
yaml
name: Run Selenium Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: '3.9'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run Selenium tests
      run: |
        pytest --tb=short --disable-warnings
```

10. Приклад GitHub Actions для запуску тестів з Pytest:

```
yaml
name: Python CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: '3.9'
```

```
- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install -r requirements.txt

- name: Run tests
  run: |
    pytest --tb=short --disable-warnings
```

Копії обов'язкових креслень (Презентація)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

ДИПЛОМНА РОБОТА
на ступінь вищої освіти магістр
із спеціальності 122 Комп'ютерні науки

Система автоматичного тестування на основі мови програмування Python

Виконав: студент 6 курсу, групи КНДМ-63
Гутнік Олексій Михайлович
Керівник: д.т.н., професор Катков Ю.І.

Київ - 2024

ЗАГАЛЬНА ХАРАКТЕРИСТИКА ДИПЛОМНОЇ РОБОТИ

Тема	Система автоматичного тестування на основі мови програмування Python
Мета дослідження	Мета роботи: Підвищення ефективності автоматизованого тестування шляхом вибору оптимальних інструментів і технологій Python інтегрованих в сучасні CI/CD процеси..
Наукове завдання	Розробка ефективної системи автоматизованого тестування програмного забезпечення з використанням Python, інтегрованої в сучасні CI/CD процеси.
Об'єкт дослідження	Процес тестування програмного забезпечення, що дозволяє забезпечити ефективність, якість і швидкість перевірки працездатності програмних систем із використанням екосистеми Python.
Предмет дослідження	Автоматизована система тестування програмного забезпечення з використанням мови програмування Python

ЗАВДАННЯ ДИПЛОМНОЇ РОБОТИ

Розглядається проблема застосування автоматизованого тестування для різних типів програмних продуктів.

Треба:.

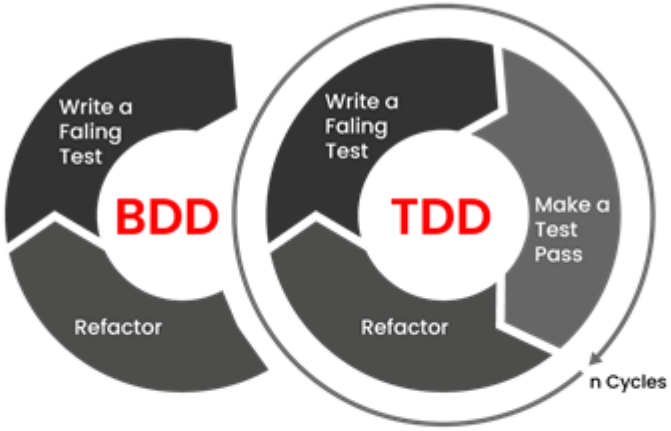
1. Провести аналіз передумов щодо впровадження автоматизованого тестування в процес розробки програмного забезпечення.
2. Дослідити критичні аспекти під час впровадження автоматизованого тестування, включаючи виклики та можливості.
3. Розробити практичні рекомендації щодо впровадження автоматизованого тестування у різних типах програмних продуктів.

Аналіз підходів та інструментів автоматизованого тестування

Вид тестування	Опис
Юніт-тестування	Перевірка окремих модулів або функцій програми на коректність.
Інтеграційне тестування	Аналіз взаємодії між модулями та компонентами, щоб виявити проблеми на рівні інтеграції.
Функціональне тестування	Оцінка відповідності системи вимогам, перевірка функціональності програми.
Регресійне тестування	Гарантія, що зміни в коді не вплинули на існуючу функціональність.
Навантажувальне тестування	Визначення продуктивності системи під значним навантаженням, перевірка стійкості.

Аналіз підходів та інструментів автоматизованого тестування

5



Методологія	Опис
TDD (Test-Driven Development)	Розробка програмного забезпечення через написання тестів перед реалізацією функціональності.
BDD (Behavior-Driven Development)	Опис поведінки системи зрозумілою мовою, щоб залучити усіх учасників проекту до процесу тестування.

Огляд існуючих систем автоматизованого тестування

6

Системи автоматизованого тестування надають розробникам та тестувальникам інструменти для підвищення якості програмного забезпечення через автоматизацію тестування. Ось три основні фреймворки, які широко використовуються у практиці:

Фреймворк	Призначення	Простота використання	Підтримка параметризації	Інтеграція з іншими інструментами
Pytest	Модульне, API	Висока	Так	Висока
Unittest	Модульне	Середня	Обмежена	Середня
Selenium	Функціональне (UI)	Низька (складне налаштування)	Немає	Висока

Ці фреймворки дозволяють ефективно тестувати різні аспекти програмного забезпечення, забезпечуючи високу якість продукту та скорочуючи час на тестування. Вибір конкретного фреймворка залежить від потреб проекту та специфіки тестування.

Інструменти для безперервної інтеграції: Jenkins, GitHub Actions

7

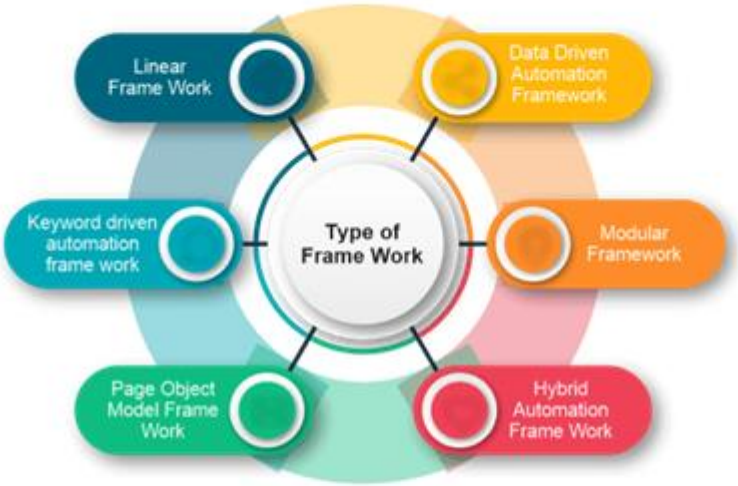
Параметр	Jenkins	GitHub Actions
Тип	Автономний CI/CD сервер	Інтегрований в GitHub
Налаштування	Складне, потребує окремого сервера	Просте, налаштовується через YAML
Плагіни/інтеграції	1800+ плагінів	Багато готових Action-скриптів
Гнучкість	Висока, підтримує кастомні сценарії	Середня, але покриває більшість потреб
Вартість	Безкоштовний, але потребує сервера	Безкоштовно для публічних репозиторіїв
Швидкість розгортання	Повільне через складність налаштування	Швидке, працює з репозиторіями GitHub

Таким чином

- Jenkins підходить для великих проєктів, де потрібна гнучкість і складні сценарії CI/CD.
- GitHub Actions є ідеальним вибором для невеликих або середніх проєктів, особливо якщо вони вже використовують GitHub.

Python для автоматизованого тестування

8



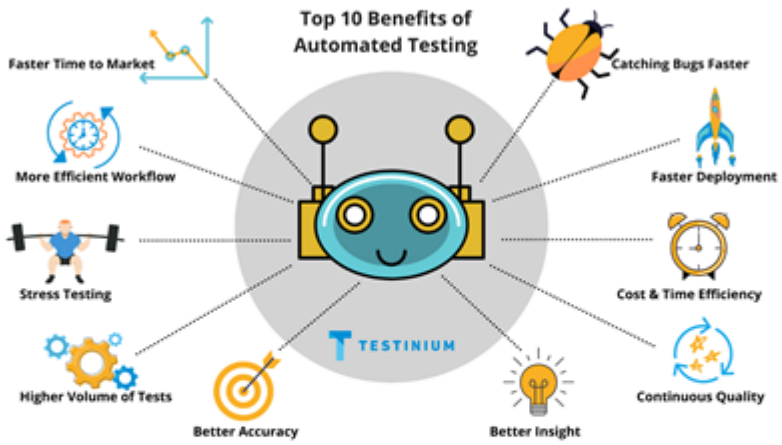
Вибір оптимальних інструментів і технологій для автоматизації тестування на Python залежить від багатьох факторів, таких як тип тестування, масштаб проєкту, доступні ресурси, інтеграційні можливості та вимоги до продуктивності.

Огляд сучасних технологій Python для тестування

Тип тестування	Рекомендовані інструменти	Додаткові примітки
Модульне тестування	Pytest, Unittest	Pytest має більше функцій для масштабних проєктів.
Інтеграційне тестування	Pytest + плагіни	Використовуйте з базами даних чи API.
Тестування UI	Selenium, Playwright	Playwright швидший і простіший для сучасних веб-додатків.
Навантажувальне тестування	Locust	Підходить для великих навантажень.
API-тестування	Requests, Pytest	Використовуйте з параметризацією для багатьох сценаріїв.

Таким чином Python забезпечує багатий набір інструментів для автоматизації тестування, який можна адаптувати до будь-яких вимог. Сучасний підхід до вибору технологій включає аналіз вимог, оцінку доступних рішень та тестування їх ефективності в реальних умовах.

Переваги автоматизованого тестування



Таким чином практична реалізація системи автоматичного тестування передбачає вибір відповідних інструментів, написання та інтеграцію тестів, а також налаштування процесів CI/CD для регулярного запуску тестів. Використання звітів і аналітики дозволяє ефективно моніторити та підтримувати якість програмного забезпечення.

Виклики та проблеми автоматизованого тестування

11

1.Складність налаштування: Інтеграція тестових фреймворків з існуючими системами може вимагати значних зусиль.

2.Високі початкові витрати: Розробка та впровадження автоматизованого тестування потребують часу і ресурсів.

3.Проблеми з підтримкою тестів: Тести можуть стати застарілими через зміни в коді, що потребує регулярного оновлення.

4.Обмежена охоплюваність: Не всі аспекти системи можуть бути автоматизовані, що може призвести до пропуску важливих тестів.

5.Залежності та конфлікти: Тести можуть мати залежності від зовнішніх систем, що ускладнює їх запуск та управління.

Ці виклики вимагають уважного планування та стратегії для ефективного впровадження автоматизованого тестування.

12

ВИСНОВКИ

•Необхідність автоматизованого тестування: Автоматизація тестування є критично важливою для забезпечення високої якості програмного забезпечення в умовах швидкої розробки.

•Ефективність та швидкість: Використання автоматизованих тестів дозволяє значно скоротити час на виявлення дефектів та підвищити швидкість випуску нових версій програмного забезпечення.

•Інструменти та методології: Вибір відповідних інструментів (таких як Python, pytest, Selenium) і методологій (TDD, BDD) є ключовим для успішної реалізації автоматизованого тестування.

•Виклики: Потрібно враховувати виклики, такі як складність налаштування та потреба у підтримці тестів, що вимагає постійної уваги з боку команди розробників.

•Перспективи розвитку: Автоматизоване тестування продовжує розвиватися, і впровадження нових технологій (наприклад, для тестування мобільних додатків) відкриває нові можливості для покращення якості програмного забезпечення.

Апробація

Стаття:

Гутнік О.М. Сучасні підходи до вибору оптимальних інструментів і технологій Python для автоматизованого тестування// Наукові записки Державного університету інформаційно-комунікаційних технологій №2, 2024, Подано до друку. https://journals.dut.edu.ua/index.php/sciencenotes/issue/view/186

В тезисах:

Гутнік О.М. ПРОБЛЕМИ РОЗВІТКУ СИСТЕМ АВТОМАТИЧНОГО ТЕСТУВАННЯ НА ОСНОВІ МОВИ ПРОГРАМУВАННЯ PYTHON // Науково-технічна конференція «Сучасні досягнення компанії Hewlett Packard Enterprise в галузі іт та нові можливості їх вивчення і застосування». Державний університет інформаційно-комунікаційних технологій. Збірник тез 14 грудня 2024. – К.: ДУІКТ, Подано до друку. https://duikt.edu.ua/ua/2661-konferencii-2024-roku-konferencii-ta-seminari