ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка фреймворку автоматичного тестування вбудованих систем на основі pytest»

на здобуття освітнього ступеня магістра
зі спеціальності 123 Комп'ютерна інженерія
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні системи та мережі
(назва)

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

_______________ Дмитро ГАКМАН
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав:
здобувач вищої освіти Дмитро ГАКМАН
група КСДМ-61

Керівник: Артем АНТОНЕНКО
науковий ступінь, вчене звання к.т.н., доцент

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

Київ 2023

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

Навчально-науковий інститут інформаційних технологій

Кафедра Комп’ютерної інженерії

Ступінь вищої освіти Магістр

Спеціальність Комп’ютерна інженерія

Освітньо-професійна програма Комр’ютерні системи та мережі

ЗАТВЕРДЖУЮ
Завідувач кафедрою КІ

_______________ Лащевська Н. О.
«_____» ___________ 2023 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Гакману Дмитру Віталійовичу
(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Розробка фреймворку автоматичного тестування вбудованих систем на основі pytest

керівник кваліфікаційної роботи Артем АНТОНЕНКО к.т.н., доцент,
(Ім’я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «_____» ___________ 2023 р. №___

2. Строк подання кваліфікаційної роботи «_____» ___________ 2023 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, архітектури фреймворків, шаблони та принципи розробки, автоматизація тестування.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Дослідження вимог до системи роботи з вбудованими пристроями
Дослідження принципів побудови фреймворків
Розробка системи тестування

Розробка документації до проєкту

5. Перелік графічного матеріалу: *презентація*

6. Дата видачі завдання «_____» ___________ 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	
2	Вивчення матеріалів для аналізу наявних фреймворків	05.11-12.11.23	
3	Дослідження архітектур систем автоматизації	13.11-19.11.23	
4	Розробка програманої частини системи автоматизації	20.11-05.12.23	
5	Розробка документації та інструкцій до системи	05.12-10.12.23	
6	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	
7	Розробка демонстраційних матеріалів	21.12-29.12.23	

Здобувач вищої освіти ________________ Дмитро ГАКМАН

(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи ________________ Артем АНТОНЕНКО

(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 112 стор., 1 табл., 13 рис., 51 джерел.

Мета роботи – розробка комплексу для тестування вбудованих систем з урахуванням їхніх специфічних особливостей, включаючи теоретичну базу та практичну реалізацію фреймворку.

Об'єкт дослідження – процес розробки комплексу для тестування вбудованих систем, у тому числі вибір та визначення вимог до вбудованого пристрою для тестування, а також аналіз та вибір оптимальної архітектури фреймворку.

Предмет дослідження – фреймворк для тестування вбудованих систем, включаючи компоненти, такі як з'єднання з пристроєм, роботу з консольними командами, логіку тестування та роботу з тестовими даними.

Короткий зміст роботи: Робота включає в себе аналіз потреб тестування вбудованих систем, вибір та визначення вимог до вбудованого пристрою для тестування, а також вибір та розробку оптимальної архітектури фреймворку. Детально розглянуті компоненти фреймворку, розроблено кодову базу та документацію, яка є важливою для ефективного використання інструментарію.

Публікації:

- Гакман Д. В., Антоненко А. В. (2023). Особливості розробки та використання фреймворків для автоматизованого тестування.Таврійський науковий вісник. Серія: Технічні науки, (2), 21-32.
- Гакман Д. В. (2023). IV Всеукраїнська науково-технічна конференція «Telecommunication: problems and innovation». Збірник тез. – К.: ДУТ, 2023, 175 - 177.
- Гакман Д. В., Черевик В. М. (2022). Формування інформаційної інфраструктури з метою забезпечення освітнього процесу: досвід та перспективи. Науково-практична конференція «Проблеми комп'ютерної

інженерії». Збірник тез. – К.: ДУТ, 2022, 236-238.

КЛЮЧОВІ СЛОВА: ВБУДОВАНІ СИСТЕМИ, АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, ФРЕЙМВОРКИ ТЕСТУВАННЯ, АРХІТЕКТУРА ПЗ

ABSTRACT

The text part of the qualification work for the master's degree: 112 pages, 1 table, 13 figures, 51 sources.

The purpose of the work is to develop a complex for testing embedded systems, taking into account their specific features, including the theoretical basis and practical implementation of the framework.

The object of research is the process of developing a complex for testing embedded systems, including the selection and definition of requirements for an embedded device for testing, as well as the analysis and selection of the optimal architecture of the framework.

The subject of the study is the development of a framework for testing embedded systems, including components such as connecting to the device, working with console commands, testing logic, and working with test data.

Summary of work: The work includes analyzing the needs of testing embedded systems, selecting and defining the requirements for an embedded device for testing, and selecting and developing the optimal framework architecture. The components of the framework are considered in detail, the code base and documentation, which is important for the effective use of the toolkit, are developed.

Publications:

- Hakman D. V., Antonenko A. V. (2023). Features of the development and use of frameworks for automated testing. Series: Technical Sciences, (2), 21-32.
- Hakman D. V. (2023). IV All-Ukrainian scientific and technical conference "Telecommunication: problems and innovation". Collection of abstracts - K.: DUT, 2023, 175 - 177.
- Hakman D. V., Cherevyk V. M. (2022). Formation of information infrastructure to ensure the educational process: experience and prospects. Scientific and practical conference "Problems of computer engineering". Collection of abstracts - K.: DUT, 2022, 236-238.

KEYWORDS: EMBEDDED SYSTEMS, TEST AUTOMATION, TEST FRAMEWORKS, SOFTWARE ARCHITECTURE

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОДАННЯ ГОЛОВІ ДЕРЖАВНОЇ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ ЩОДО ЗАХИСТУ БАКАЛАВРСЬКОЇ РОБОТИ

Направляється здобувач Гакман Д. В. до захисту кваліфікаційної роботи

за спеціальністю 123 Комп'ютерна інженерія

освітньо-професійної програми Комп'ютерні системи та мережі

на тему: «Розробка фреймворку автоматичного тестування вбудованих систем на основі pytest».

Кваліфікаційна робота і рецензія додаються.

Директор ННІІТ ____________ _________________
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Висновок керівника кваліфікаційної роботи

Здобувач ____

____

____

____

____

____

Все це дозволяє оцінити виконану бакалаврську роботу здобувача Гакмана Д. В. на оцінку «___________________________________» та присвоїти йому кваліфікацію ____.

Керівник роботи ________ Антоненко А. В.

«_____»_________ 2022 року

Висновок кафедри про кваліфікаційну роботу

Кваліфікаційна робота розглянута. Здобувач Гакман Д. В. допускається до захисту даної роботи в Екзаменаційній комісії.

Завідувач кафедри комп'ютерної інженерії ___________ _________________
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

ВІДГУК РЕЦЕНЗЕНТА
на кваліфікаційну магістерську роботу

здобувача вищої освіти Гакмана Дмитра Віталійовича
на тему: «Розробка фреймворку автоматичного тестування вбудованих систем на основі pytest».

Актуальність.
Вбудовані системи стають все більш поширеними в різних галузях - від автомобільної промисловості до медичних пристроїв, від домашньої електроніки до промислового обладнання. Однак, через їх специфіку, тестування вбудованих систем вимагає особливого підходу. Багато загальних фреймворків для тестування не враховують цих особливостей, що може призвести до неточностей у тестуванні або пропуску потенційних проблем. Дана робота направлена на розробку фреймворку, який би враховував ці особливості. Крім того, було зроблено акцент на створенні більш універсального інструменту, який може працювати з різними системами. Це важливо, оскільки вбудовані системи можуть значно відрізнятися за своїми характеристиками та вимогами. Таким чином, кваліфікаційна робота має практичне значення і застосування в галузі вбудованих систем, а отже є актуальною

Позитивні сторони.
1. Автор роботи продемонстрував систематичний підхід до розробки фреймворку, включаючи детальний аналіз існуючих архітектур фреймворків та обґрунтований вибір Python та Pytest як основних інструментів.
2. Розроблено модульну структуру фреймворку, що сприяє його гнучкості та можливості розширення, що є критично важливим для сучасних систем.
3. Створено детальну документацію, що включає алгоритми роботи з фреймворком, детальні рекомендації та послідовні інструкції, що є важливим ресурсом для інженерної команди.

Недоліки.
1. При реалізації роботи можливо було виконати певні пункти щодо універсалізації модулів конекторів.
2. Наведено небагато готових тестових сценаріїв як прикладів використання фреймворку.

Відзначені зауваження не впливають на загальну позитивну оцінку бакалаврської роботи.

Висновок: кваліфікаційна магістерська робота заслуговує оцінку "__________", а здобувач – Гакман Дмитро Віталійович заслуговує присвоєння кваліфікації ________________

Рецензент: ___________ _____________________
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface

CSV — Comma Separated Values

JSON – JavaScript Object Notation.

URL – Uniform Resource Locator

YAML — Yet Another Markup Language

ВСТУП

Команда BCC Research Team раз на декілька років проводить детальне дослідження ринку вбудованих систем. Їх статті та аналізи демонструють постійний ріст ринку вбудованих систем. Останній їх матеріал було опубліковано в 2020 році. В ньому зазначено що глобальний ринок вбудованих систем має зрости з 207,3 мільярдів доларів США у 2020 році до 267,3 мільярдів доларів США до 2025 року зі зведеним річним темпом зростання (CAGR) 5,2% за період 2020-2025 років[1]. Як можна бачити, вбудовані системи невпинно розвиваються, а отже стають все складнішими.

Дані системи відрізняються від більш традиційних, таких як комп'ютери, оскільки вони обмежені вбудованим обладнанням, призначені для виконання конкретної функції та мають високі вимоги до якості та надійності, через що такі пристрої тісно пов'язані з критичними системами. Критичні системи - це системи, в яких несправності можуть завдати шкоди живим організмам або навколишньому середовищу. Таким чином, важливим фактором у розробці таких систем є забезпечення їхньої коректності, що вони працюють відповідно до своєї специфікації.

Коректність системи можливо перевірити через процес тестування, де виявляються дефекти, які потім можна виправити і, відповідно покращити якість продукту. Однак тестування вбудованих систем має свій ступінь складності. По-перше, вбудована система обмежена своїм власним контекстом, і для проведення тестування таких систем потрібно знати їх функціональні можливості та способи їх активації. По-друге, з урахуванням широкого використання вбудованих систем у кожному сегменті інженерного ринку, в кожному з них прийняті свої комунікаційні протоколи.

У зв'язку з цією складністю тестування вбудованих систем потрібно створити рішення, яке автоматизує процес тестування вбудованих пристроїв. Зокрема, те, яке здатне перекладати операції вбудованих систем у тестові скрипти, які можна викликати та виконувати функціонал.

Автоматизація тестування означає автоматизацію ручних кроків, включаючи розробку та виконання тестових скриптів, перевірку вимог до тестування та використання тестових інструментів. В порівнянні з ручним, автоматичне тестування дозволяє зменшити час, необхідний для перевірок, та збільшити їх точність, оскільки прибирається людський фактор[2]. Ці пункти дозволяють продуктам виходити на ринок раніше, що є важливим для успіху виробника. Однак, тестування програмного забезпечення, може зменшити витрати на програмне забезпечення, що може забрати понад 50% від загальних зусиль у розробці програмного продукту[3].

У 2011 році інженерам Google вдалося розробити та випустити понад 100 функцій Google+ протягом 100 днів [4]. Виконано це було шляхом стратегічного поєднання ручного та автоматизованого тестування на різних етапах SDLC.

З того часу Google «оновив» свої базові налаштування до платформи Google Test Automation Platform (TAP) — найбільшої в світі системи автоматизації тестування програмного забезпечення, що підтримується такими принципами DevOps, як Continuous Integration (CI), Continuous Testing (CT) і Continuous Delivery (CD).

Щодня TAP інтегрує та тестує понад 13 тисяч проєктів коду, що вимагає 800 тисяч збірок і 150 мільйонів тестових запусків. Навіть у меншому масштабі автоматизація тестування забезпечує надзвичайну швидкість обробки та деякі додаткові переваги, такі як:

- Миттєвий відгук про нещодавно надісланий код
- Вища економічна ефективність порівняно з ручним тестуванням
- Швидка швидкість виконання, що покращує продуктивність команди
- Вища точність тесту, оскільки менше можливостей для людської помилки
- Краще тестове покриття, оскільки ви можете зробити більше швидше
- Повторне використання тестових випадків, мінімізація зайвої роботи
- Швидший час виходу нових продуктів на ринок

Але у сфері веб-додатків та десктопних додатків автоматизація тестування досягла високого рівня зрілості, і на ринку є дуже повні рішення на відміну від вбудованих систем.

Отже, потрібен аналіз різних інструментів, а потім процес оцінки, який обере інструменти що найбільше підходять під вимоги. Наприклад, його можливість спілкування з вбудованим пристроєм, здатність виконувати різні види автоматизованого тестування та легкість специфікації тестових випадків. Після вибору засобу автоматизації тестування, на якому будуватиметься система, буде розглянуто способи спілкування з вбудованим пристроєм.

1 ПОСТАНОВКА ЗАДАЧІ

Для того щоб успішно провести аналіз існуючих напрацювань спершу потрібно описати поставлену задачу. Задача даної роботи - розробка комплексу що складається з теоретичної та програмної частини, та вирішує такі проблеми як розробка фреймворку для тестування вбудованих систем, процес роботи з даним фреймворком та його інтеграції в роботу команди. Оскільки фреймворки що існують, орієнтовані на дуже широкий спектр проблем, вони не покривають специфічних задач проєктів. Саме тому прийнято рішення провести розробку системи автоматизації тестування для вбудованих систем. В свою чергу всі тестувальники-автоматизатори мають досвід з тими чи іншими наявними фреймворками, тому було вирішено обрати один із них як базис, адже це дозволить направити всі ресурси на вирішення задач що стосуються вбудованих систем, а не концентруватись на роботі над вже вирішеними проблемами.

Першим питанням що постає - це вибір пристрою, який дозволить розробити прикладну систему тестування, та при цьому якогомога менше загострювати увагу на конкретному пристрої. Це означає що девайс повинен застосовувати якомога більш універсальні технології, які часто використовуються в інших системах також. Оскільки ціль даного проєкту - система тестування вбудованих девайсів, потрібно розуміти що таке вбудований девайс та виділити його основні характеристики. Це дозволить детально проаналізувати та обрати пункти на які потрібно звертати увагу при виборі.

1.1 Визначення пристрою для тестування

Термін Embedded System можна визначити як спеціалізовану обчислювальну систему, яка безпосередньо взаємодіє з фізичними процесами та об’єктами для розв’язання конкретного завдання. До складу простої вбудованої системи входять [5]:

- мікропроцесорний модуль з пам’яттю;
- периферійна система (датчики, виконавчі елементи і
- контролери введення-виведення для зв’язку з об’єктом
- контролю/управління, пристрої людино-машинного інтерфейсу за
- необхідністю);ц
- система електроживлення;
- конструктив, що об’єднує (шасі, корпус);
- програмне забезпечення, що керує.

Вбудовані системи значно відрізняються від систем загального призначення, наприклад персональних комп’ютерів. Їх основними особливостями вважаються [5-7]:

- робота в реальному масштабі часу;
- багатозадачність;
- обмежені ресурси;
- мінімізація вартості;
- програмно-апаратний дуалізм;
- різноманітні, часто складні, умови експлуатації;
- автономність роботи (відсутність оператора, обмеження електроживлення);
- високі вимоги щодо надійності і безпеки функціонування, особливо коли ВС використовуються у відповідальних областях, де треба враховувати всі зовнішні фактори, які можуть вплинути на надійність (природні або людські);
- критичні застосування, пов’язані зі здоров’ям і життям людини.

Дані систем відносяться до категорії систем з більшою мірою програмною реалізацією (Software-Intensive або Software-Dominated Systems). Як видно з досліджень, проведених компаніями UBM Tech Electronics та EE Times Group у 2013 році [8], більш ніж 60% ресурсів витрачається на розробку програмного забезпечення. Саме слово “вбудоване” підкреслює особливі властивості такого програмного забезпечення та вказує на потребу в спеціальних технологіях його створення [9-12].

Отож, проаналізувавши характеристики вбудованих систем, тестовий девайс повинен:

- Бути системою з обмеженими ресурсами
- Надавати інтерфейс керування чи спілкування (до прикладу невелика операційна системи з існуючим командним рядком, та набором команд).
- Надавати можливість підключення до пристрою різноманітними способами що є найбільш частими у використанні на даний час.
- Мати якісну та відкриту документацію, адже без неї реалізація системи тестування неможлива.

Маршрутизатори є одними з найбільш підходящих пристроїв відповідно до описаних потреб, оскільки:

- Багато маршрутизаторів мають обмежені обчислювальні ресурси в порівнянні з сучасними серверами чи комп'ютерами. Вони призначені для розподілення мережевого трафіку та повинні робити це досить ефективно за наявності обмежених ресурсів.
- Більшість маршрутизаторів мають вбудовані операційні системи з командним рядком, такі як Cisco IOS, Juniper Junos або відкриту операційну систему, таку як OpenWRT. Ці ОС надають інтерфейс для керування та налаштування маршрутизатора, а також можливість виконувати команди через командний рядок.
- Маршрутизатори підтримують різні методи підключення, такі як Ethernet, Wi-Fi, під’єднання по SSH чи Serial, інші мережеві протоколи та інтерфейси для спілкування з іншими мережевими пристроями. Вони також можуть бути легко інтегровані в існуючі мережі.
- Виробники маршрутизаторів зазвичай надають докладну та доступну документацію для своїх пристроїв, включаючи технічні специфікації, керівництва з налаштування та інструкції з використання. Більшість

маршрутизаторів використовують відкриті операційні системи, що сприяє доступності документації та можливості змінювати їхнє ПЗ.

У випадку, коли пристрій задовольняє вищезазначеним критеріям, він стає ідеальним вибором для даного проєкту, адже сама система тестування повинна бути адаптивною та якомога більш універсальною. Тому в подальший вибір було зроблено на основі наявних ресурсів та пристроїв.

У ролі пристрою для проведення тестування було обрано маршрутизатор Mikrotik RB2011U1AS. Величезною перевагою пристрою і в загальному цієї компанії в порівнянні з іншими є чудова документація. Вони надають докладні керівництва з налаштування, документацію з команд і можливість отримати технічну підтримку. Крім того, RouterOS є відкритою операційною системою, що дозволяє спеціалістам глибоко налаштовувати і розширювати її функціонал за необхідності.

1.2 Вимоги до системи

Визначивши пункти вибору тестового девайсу, наступним етапом в розгляді є програмна частина фреймворку. Цей аспект обговорює побудову архітектури та структури фреймворку, вибір інструментів, імплементацію різноманітних блоків відповідно до створеного дизайну та їх подальше поєднання в єдину систему. Зазначено, що безперервна та успішна реалізація цих елементів стає можливою лише у випадку, коли існують чітко сформульовані вимоги до системи. З цією метою, потрібно виконати складання PRD (Product Requirements Document).

Правильно сформульовані вимоги є основними вказівками, які визначають, яким чином повинен функціонувати та виконувати свої завдання програмний продукт. Вони стають відправною точкою для всієї подальшої розробки. Першочергово, написання вимог дозволяє створити чітке і ясне розуміння того, що вимагається від програмного продукту відповідно до потреб користувачів та бізнес-цілей. Це дозволяє розробникам та іншим учасникам проєкту мати чітке бачення функціональності та параметрів продукту. Додатково, вимоги визначають межі та обмеження проєкту, які важливі для управління обсягом та бюджетом розробки. Вони визначають необхідні ресурси, технічні характеристики та терміни, що стосуються проєкту. Важливо враховувати, що неправильно сформульовані або недостатньо докладні вимоги можуть призвести до затримок, перевищення допустимих ресурсів і недоліків в якості продукту.

Написання вимог є інструментом для уникнення непорозумінь, якщо над проєктом працює декілька людей чи команд. Вони створюють спільну мову та референс-пункт, який може бути використаний для оцінки виконаної роботи та вирішення спірних питань.

Отже, написання вимог перед розробкою продукту відіграє центральну роль у забезпеченні успішності та якості розробки.

– Вся система повинна бути організована в репозиторії системи контролю версій Git.
– Підтримка комунікаційних інтерфейсів: Забезпечення можливості взаємодії з різними комунікаційними інтерфейсами, такими як USB, Bluetooth, Wi-Fi, SPI, I2C і т. д., залежно від специфікацій пристрою.
– Автоматизоване з'єднання і роз'єднання: Можливість автоматичного встановлення з'єднання з тестовим пристроєм і його відключення після завершення тесту.
– Керування життєвим циклом пристрою: Здатність керувати життєвим циклом пристрою, включаючи запуск, перезавантаження, завершення роботи тощо.
– Генерація входів: Можливість генерувати вхідні дані для пристрою, щоб перевірити його реакцію на різні сценарії.
– Збір і аналіз даних: Здатність збирати дані від пристрою та аналізувати їх для визначення стану пристрою і результатів тестування.
– Логування і звіти: Можливість створювати докладні логи і звіти про результати тестів для подальшого аналізу і документації.

- Система тестування повинна бути орієнтована на роботу з девайсами за наявності командного рядка будь-якого вигляду.
- Задання тестових параметрів: Можливість вказання параметрів тестування, таких як ідентифікатори пристроїв, конфігураційні файли тощо.
- Система повинна бути розбита на блоки які мінімально залежать один від одного. це дозволить адаптувати розроблену систему під широке коло задач з мінімальною кількістю змін.
- Можливість розширення: Фреймворк повинен бути легко розширюваним для додавання нових функцій та інтеграції з іншими інструментами.
- В системі повинно бути відведене місце для створення нових та збереження всіх існуючих тестових сценаріїв.
- Автоматизатор повинен мати змогу розгорнути дану систему на робочій машині під керуванням дистрибутиву Linux.
- Автоматизатор повинен мати змогу запускати наявні тестові сценарії вручну
- Автоматизатор повинен мати змогу розробляти нові тестові сценарії в системі.
- Все оточення для роботи з системою повинно бути уніфіковане. Це дозволить не витрачати ресурси на відлагодження системи на робочій машині кожного інженера.
- Всі паролі та інші приватні дані повинні міститися в змінних оточення операційної системи або конфігураційних файлах.
- Всі приватні дані не повинні публікуватись в репозиторії та, за потреби, вручну вноситись в систему інженерами, які мають доступ до цієї інформації.
- Кроки тестових сценаріїв не повинні залежати від способу підключення до тестового девайсу.
- Система повинна надавати можливість розробляти як host-based тести так і target-based тести.

- Система повинна надавати можливість розробляти як повністю автоматичні так і частково автоматичні тестові сценарії.
- В системі повинна бути присутня документація яка описує такі аспекти:
  - процес налаштування робочого оточення;
  - всі використані інструменти та їх опис;
  - різноманітні опції та варіації використання системи;
  - правила, яких повинні дотримуватись автоматизатори при розробці тест кейсів;
  - процес рев'ю новостворених тест кейсів іншими інженерами з команди;

1.3 Вибір інструментів

1.3.1 Мова програмування

Одним із аспектів розробки програмного продукту, який набуває великої вагомості після описання вимог, є вибір інструментів і, зокрема, мови програмування. Вибір правильної мови програмування є стратегічно важливою точкою у процесі розробки, оскільки має глибокий вплив на продуктивність автоматизаторів, якість коду, швидкодію тестових сценаріїв і навіть можливість вдосконалення фреймворку в майбутньому.

Обрана мова програмування визначає доступність інструментів розробки, бібліотек і додаткових фреймворків, які можна використовувати для досягнення задач, описаних у вимогах. Кожна мова має свої переваги і обмеження, і вибір повинен базуватися на конкретних вимогах проєкту, а також на знаннях та навичках розробників. Отже, цей важливий етап розробки вимагає ретельного аналізу і вибору оптимального рішення для досягнення успіху проєкту.

Кожна мова програмування була створена для вирішення певного типу задач, тому не всі існуючі мови можуть застосовуватись для тестування вбудованих пристроїв. Такими що найбільш часто використовуються на проєктах є:

Bash-скрипти:

- Переваги: Простий синтаксис, доступний за замовчуванням в Linux. Відмінно підходить для автоматизації командного рядка та швидкого написання простих сценаріїв.
- Недоліки: Обмежена можливість виразів та обробки помилок, не найкращий вибір для складних сценаріїв.

Python:

- Переваги: Зручна та читабельна мова програмування, багато бібліотек для тестування, хороша підтримка вбудованих операційних систем.
- Недоліки: Вимагає інтерпретатора Python на вбудованих системах у випадку target-based тестів, може бути повільним для деяких завдань.

Perl:

- Переваги: Має потужні засоби для обробки тексту та регулярних виразів, добре підходить для швидкого прототипування.
- Недоліки: Складний синтаксис, вимагає додаткових бібліотек для тестування, не так популярний, як Python.

Shell-скрипти (sh, dash, і т. д.):

- Переваги: Мінімальний інтерфейс, підтримується на більшості Unix-подібних систем, включаючи вбудовані.
- Недоліки: Обмежена функціональність, менш елегантний синтаксис порівняно з іншими мовами.

C/C++:

- Переваги: Висока продуктивність, близька до апаратного рівня, підтримка вбудованих систем.
- Недоліки: Складний синтаксис, тривалий час розробки, може бути важко відлагоджувати.

У таблиці 1.1 було звернуто увагу на кілька ключових аспектів, що визначають придатність мови для цієї мети.

Таблиця 1.1 Порівняння характеристик мов програмування

Мова	Кількість розробників	Легкість у використанні	Підтримка від творців/ком'юніті	Наявність бібліотек та фреймворків
Bash-скрипти	Низька	Висока	Громадська підтримка	Багато стандартних утиліт та бібліотек Linux.
Python	Висока	Висока	Активна спільнота користувачів, підтримка від Python Software Foundation	Багато бібліотек, таких як unittest, pytest, і Robot Framework.
Perl	Низька	Середня	Активна спільнота користувачів	Є багато бібліотек для тестування.
Shell-скрипти	Низька	Висока	Зазвичай громадська підтримка	Спеціалізовані бібліотеки менше розповсюджені.
C/C++	Висока	Середня	Зазвичай підтримка від розробників	Багато бібліотек для вбудованих систем.

Python виділяється як один з найбільш переконливих кандидатів з наступних причин:

– Python має значну кількість активних розробників, що забезпечує постійний розвиток мови і наявність нових бібліотек та інструментів для тестування.

- Python відомий своєю природною читабельністю коду та простим синтаксисом, що сприяє розробці та підтримці тестових сценаріїв. Це особливо важливо в контексті тестування, оскільки прозорий код полегшує розуміння інтенцій розробника.
- Python є вільним та відкритим програмним забезпеченням, що дозволяє використовувати його для комерційних і некомерційних цілей без обмежень. Це важливий аспект для вбудованих систем, оскільки можливість використовувати мову без витрат на ліцензії є значущим фактором.
- Python має активну спільноту користувачів, а також підтримку від Python Software Foundation, що гарантує надійність та стабільність мови.
- Python пропонує розширенк екосистему бібліотек та фреймворків для тестування, включаючи популярні бібліотеки, такі як unittest, pytest, і Robot Framework, що спрощують процес створення та виконання тестових сценаріїв.

З урахуванням цих факторів, дана мова видається обґрунтованим вибором для реалізації фреймворку для тестування вбудованих пристроїв. Вона поєднує в собі легкість використання, потужність і розширюваність, що робить її важливим інструментом для забезпечення надійності та функціональності системи тестування.

1.3.2 Базовий фреймворк

Оскільки існують загальні фреймворки, які вже мають певний функціонал, щоб не розв'язувати задачі які вже вирішені, для реалізації системи тестування вбудованих систем, як базис, буде використано один із існуючих фреймворків. Оптимально підібраний тестовий фреймворк визначає методики, інструменти та підходи, які будуть використовуватися для перевірки функціональності програми і може суттєво впливати на результати тестування та швидкість виявлення помилок.

Для досягнення успіху, необхідно не лише визначити оптимальний тестовий фреймворк, але й розробити стратегію тестування, підібрати відповідні інструменти і визначити методики проведення тестів. В цьому контексті, вибір тестового фреймворку визначає основу для всього процесу тестування і вимагає обґрунтованого підходу, щоб гарантувати високу якість та надійність програмного продукту.

Відштовхуючись від вже обраної мови програмування Python, визначимо наявні варіанти фреймворків, їхні переваги та недоліки.

PyTest - одни із фреймворків для автоматизованого тестування ПЗ. Він здобув популярність завдяки своїй простоті використання, розширюваності та можливостям інтеграції з іншими інструментами.

Основні переваги

– Простота використання: PyTest дозволяє легко створювати тести за допомогою інтуїтивного синтаксису. Можливість створювати тести у вигляді звичайних функцій Python робить процес написання тестів швидким та зрозумілим.
– Автоматичне виявлення тестів: PyTest може автоматично виявляти тести в вашому проєкті, що спрощує процес створення тестових комплектів.
– Розширення та Плагіни: PyTest має багато розширень та плагінів, які дозволяють розширити його функціональність. Наприклад, існують плагіни для підтримки параметризації тестів, звітів, відлагодження та багато інших.
– Широкий вибір перевірок: PyTest надає багато вбудованих асертів для порівняння результатів тестів, що полегшує написання та читання тестів.
– Інтеграція з іншими інструментами: PyTest легко інтегрується з іншими інструментами, такими як Jenkins, Travis CI, та іншими системами для забезпечення CI/CD. Це дозволяє автоматизувати процес тестування та впровадження програмного забезпечення.

Недоліки:

– Швидкість виконання: Хоча PyTest досить швидкий, він все одно повільніший у порівнянні з деякими іншими тестовими фреймворками, такими як Nose. Це може бути помітним для дуже великих проєктів.

- Специфічні питання інтеграції: При інтеграції з деякими іншими інструментами або бібліотеками, можуть виникати питання або конфлікти, що вимагають додаткового налаштування.
- З погляду зручності використання та розширення функціональності, PyTest є потужним інструментом для автоматизованого тестування в програмуванні на Python.

Unittest (або ще відомий як PyUnit) - це вбудований фреймворк зі стандартної бібліотеки Python. Він надає засоби для створення, виконання та організації тестів.

Основні переваги

- Стандартна бібліотека: unittest є частиною стандартної бібліотеки Python, що означає, що ви не потребуєте встановлювати жодні додаткові бібліотеки для його використання. Він включений в стандартний комплект Python, що полегшує роботу з ним.
- Підтримка тестових комплектів: unittest дозволяє організовувати тести в тестові комплекти, що полегшує групування тестів і виконання їх у пакетному режимі.
- Можливість створювати тести як класи: Ви можете створювати тести як класи, що надає можливість використовувати спадкування та інші ООП-принципи для організації тестового коду.
- Багато функціональних можливостей: unittest надає багато можливостей для налаштування тестових умов, перевірки та інших аспектів тестування.

Недоліки**:**

- Високий обсяг коду: unittest вимагає більше коду для написання тестів порівняно з іншими тестовими фреймворками, такими як PyTest. Це може призвести до ускладнення розробки та підтримки автоматичних тестів.
- Складна структура тестового коду: У деяких випадках, unittest може

змусити створювати складну ієрархію класів для організації тестів, що може бути заплутаним для початківців та вимагатиме більше ресурсу на створення тестового сценарію.

- Менша популярність у порівнянні з іншими фреймворками: У порівнянні з PyTest, unittest може мати меншу популярність та менше розширень та плагінів для збільшення його функціональності.
- Обмежене вбудоване логування та звіти: unittest не надає великої кількості можливостей для генерації звітів та логування порівняно з іншими фреймворками.

Unittest може бути корисним для проєктів, де потрібна вбудована підтримка тестування в мові Python, або в тих випадках, коли вам зручно використовувати стандартний фреймворк. Проте, він має свої обмеження та вимагає більше зусиль для налаштування та використання порівняно з деякими іншими фреймворками.

Nose - це фреймворк популярність якого з часом знизилася через появу більш сучасних фреймворків. Він все ще має свої особливості та може бути корисним для певних проєктів та все ж це має певний вплив на його розробку та підтримку.

Основні переваги:

- Автоматичне виявлення тестів: Nose здатний автоматично виявляти тести в вашому проєкті, що полегшує процес написання та організації тестових комплектів.
- Підтримка паралельного виконання: Nose підтримує паралельне виконання тестів, що може значно прискорити процес тестування в великих проєктах.
- Багато плагінів та розширень: Nose має значну кількість плагінів, що дозволяють збільшити його функціональність. Це включає плагіни для генерації звітів, інтеграції з іншими інструментами та інше.
- Простий синтаксис для параметризованих тестів: Nose надає зручний синтаксис для створення параметризованих тестів, що дозволяє вам легко тестувати різні комбінації вхідних даних.

Недоліки:

- Можливі проблеми із сумісністю: Nose може виявляти проблеми із сумісністю та стабільністю з новими версіями Python, оскільки підтримка та розвиток фреймворку з часом зменшувалася.
- Питання щодо підтримки: Оскільки Nose втратив популярність, він може мати менше активну спільноту користувачів та менше оновлень та підтримки порівняно з іншими фреймворками.
- Обмежена функціональність порівняно з іншими фреймворками: Насправді, Nose має менше функціональності та можливостей порівняно з іншими сучасними тестовими фреймворками, такими як PyTest.

Хоча Nose має свої переваги, він може бути менш привабливим вибором порівняно з більш сучасними тестовими фреймворками, такими як PyTest чи unittest. Проте, він все ще може бути корисним для проєктів, які вже використовують його або де вимагається специфічна функціональність.

Robot Framework - це відкритий фреймворк що використовує свій власний синтаксис. Цей фреймворк відзначається своєю унікальною специфікою, оскільки дозволяє писати тести в форматі ключових слів (Keyword-Driven Testing).

Основні переваги:

- Формат ключових слів: Robot Framework дозволяє описувати тести в форматі ключових слів, що робить тестування більш доступним для людей без глибоких технічних знань. Це особливо корисно в сферах, де фахівці з тестування мають обмежені навички програмування.
- Широкий спектр бібліотек і плагінів: Robot Framework має велику кількість бібліотек та плагінів, що покривають різні аспекти тестування та автоматизації. Це дозволяє розширити його функціональність та адаптувати під конкретні потреби проєкту.
- Легке створення звітів і логування: Robot Framework надає зручні інструменти для створення звітів та логування результатів тестів. Це допомагає відстежувати стан проєкту та результати тестування.

– Підтримка планування та виконання тестів: Robot Framework дозволяє планувати та виконувати тести в автоматизованих режимах, що робить його корисним для CI/CD та інших автоматизованих процесів.

Недоліки:

– Навички використання специфічного синтаксису: Використання специфічного синтаксису Robot Framework вимагає навичок та часу на оволодіння. Для деяких користувачів це може бути бар'єром у використанні.
– Швидкість виконання: Хоча Robot Framework надає високорівневий інтерфейс для тестування, він може бути повільним у порівнянні з іншими більш низькорівневими фреймворками.
– Менше популярний: У порівнянні з PyTest чи unittest, Robot Framework може мати менше популярність та меншу активну спільноту користувачів.

Robot Framework - це потужний фреймворк для автоматизованого тестування та автоматизації, особливо в сферах, де ключовими фігурами є тестувальники, які не мають глибоких знань програмування. Його унікальний формат ключових слів робить його цікавим вибором для певних проєктів.

Згідно із зазначеними вимогами, фреймворк повинен володіти гнучкістю, розширюваністю, широкою підтримкою плагінів і високоякісною документацією. Тому після аналізу можливих альтернатив було встановлено, що PyTest являється найбільш оптимальним вибором базового фреймворку для тестування вбудованих систем з наступних причин:

– PyTest володіє простим і зрозумілим синтаксисом, що сприяє легкій і швидкій інтеграції з вбудованими системами. Ця простота дозволяє інженерам швидко створювати і адаптувати тести під конкретні потреби пристрою.
– Фреймворк має значну кількість розширень та плагінів, які дозволяють легко збільшити його функціональність.
– PyTest надає ретельно структуровану та докладну документацію, що є надзвичайно важливим для інженерів, які працюють з вбудованими

системами. Ця документація допомагає швидко ознайомитися з фреймворком і використовувати його ефективно в конкретному контексті.

- Можливість відлагоджування (debugging) важлива для вбудованого програмування. PyTest надає зручні інструменти для відлагодження, що дозволяють інженерам знайти та виправити помилки у вбудованих програмах та тестах.
- Фреймворк має активну спільноту користувачів та розробників, що забезпечує актуальну та надійну підтримку. Ця підтримка дозволяє швидко вирішувати проблеми та отримувати рекомендації щодо вдосконалення процесу тестування вбудованих пристроїв.

2 РОЗРОБКА ФРЕЙМВОРКУ

Шаблони, принципи та методи автоматизації тестування є критично важливими при розробці тестових сценаріїв. Але набагато менше уваги приділяється розробці та реалізації архітектури та інфраструктури автоматизації тестування. І, хоча, автоматизація QA є складним процесом, вона також є систематичним, тому щоб обрати архітектуру і розробити систему тестування, потрібно виконати певний набір кроків.

2.1 Визначення оптимальної архітектури системи

Забезпечення якості – це багатогранна діяльність. Програмне забезпечення можна досліджувати під різними кутами, щоб переконатися, що воно працює так, як очікується, і не має критичних помилок чи вразливостей. З цієї причини існує багато різних типів тестів, які розбиваються на дві основні категорії, функціональне та нефункціональне.

Функціональне тестування є ключовим етапом в процесі перевірки програмного забезпечення на відповідність заявленим вимогам та очікуваному функціоналу. Цей вид тестування спрямований на перевірку роботи програми з урахуванням конкретних функцій та операцій, які вона повинна виконувати згідно з вимогами специфікації. Функціональне тестування орієнтоване на валідацію правильності функцій програми, включаючи введення даних, обробку, зберігання та виведення інформації. Це включає тестування коректності алгоритмів, перевірку точності результатів, а також взаємодії з іншими компонентами системи. Мета функціонального тестування полягає в ідентифікації помилок, неузгоджень або відхилень у роботі програми, щоб розробники могли виправити їх перед впровадженням. Підтипами функціональних тестів є

– Модульні тести

- Компонентні тести
- Димові випробування
- Регресійні тести
- Тести API
- Тести інтерфейсу користувача

Нефункціональне тестування, навпаки, спрямоване на оцінку якості та характеристик програмного забезпечення, які не пов'язані з його функціональністю. Цей вид тестування ставить за мету перевірку аспектів, таких як продуктивність, безпека, масштабованість, сумісність, надійність та інші нефункціональні параметри програми. Такі тести включають перевірку реакції програми на навантаження або стресові ситуації, перевірку захищеності від потенційних загроз, оцінку швидкодії та ефективності програми в різних умовах використання. Нефункціональне тестування важливо для забезпечення не лише працездатності програми, а й її придатності для використання в реальних умовах.

- Тести продуктивності
- Навантажувальні тести
- Тестування на проникнення (тести безпеки)
- Стрес-тести
- Об'ємні тести
- Тести на відновлення

Так, їх всіх можна включити до системи AQA, але це буде нераціональний підхід до вирішення проблеми. Хороша система автоматизованого тестування повинна зосереджуватися на певному наборі проблем і бути розроблена відповідно до типу продукту. Перш ніж почати розробляти архітектуру, потрібно чітко визначити пріоритети забезпечення якості, виписати основні вимоги до тестування, вибрати серію необхідних тестів та розглянути різні стратегії їх автоматизації.

Різноманітні типи тестування мають свої важливість та покликання на різних етапах процесу розробки програмного забезпечення. Деякі з них важливо виконати раніше через їх спрямованість на базовий функціонал, а інші потребують більшого розгорнутого середовища чи відомостей про систему.

На ранній стадії розробки найважливішим є функціональне тестування, оскільки воно перевіряє правильність роботи окремих модулів, компонентів та системи в цілому. Модульні тести та компонентні тести дозволяють виявити помилки на ранніх етапах, коли вони ще не розрослися та легше виправляються. Регресійні тести також важливі на цьому етапі, оскільки вони перевіряють, чи не вплинули останні зміни на вже наявний функціонал.

Пізніше, коли базовий функціонал вже протестований, варто звернутися до нефункціонального тестування. Тести продуктивності, навантажувальні тести та стрес-тести важливі для визначення працездатності системи під час великої кількості операцій чи в умовах великої навантаженості. Тестування на проникнення виявляє потенційні уразливості системи перед їх використанням з боку зловмисників. Об'ємні тести та тести на відновлення важливі для визначення реакції системи на об'ємні дані та відновлення роботи після аварійних ситуацій.

Ця послідовність тестування забезпечує покриття базових функцій та стабільність системи на ранніх етапах розробки, а також гарантує її продуктивність, безпеку та відновлюваність пізніше, коли система вже має базовий функціонал.

На основі інформації наведеної вище, у розробці фреймворку для тестування, приділення уваги функціональним тестам є обґрунтованим, оскільки це дає змогу забезпечити основні функціональність та якість програмного забезпечення на ранніх етапах його розробки, що дозволяє знизити витрати на виправлення помилок у подальшому та поліпшує загальну якість та стабільність продукту. Такий підхід до розробки фреймворку сприяє покращенню відповідності вимогам швидкого виходу на ринок та сприяє задоволенню користувачів.

Наступним пунктом що потрібно розглянути при виборі правильного підходу до розробки фреймворку - це різноманітність систем автоматизації тестування. Кожна з цих систем має власну логіку, що впливає на ефективність та точність тестів програмного забезпечення.

Існуючі системи автоматизації (рис. 2.1):

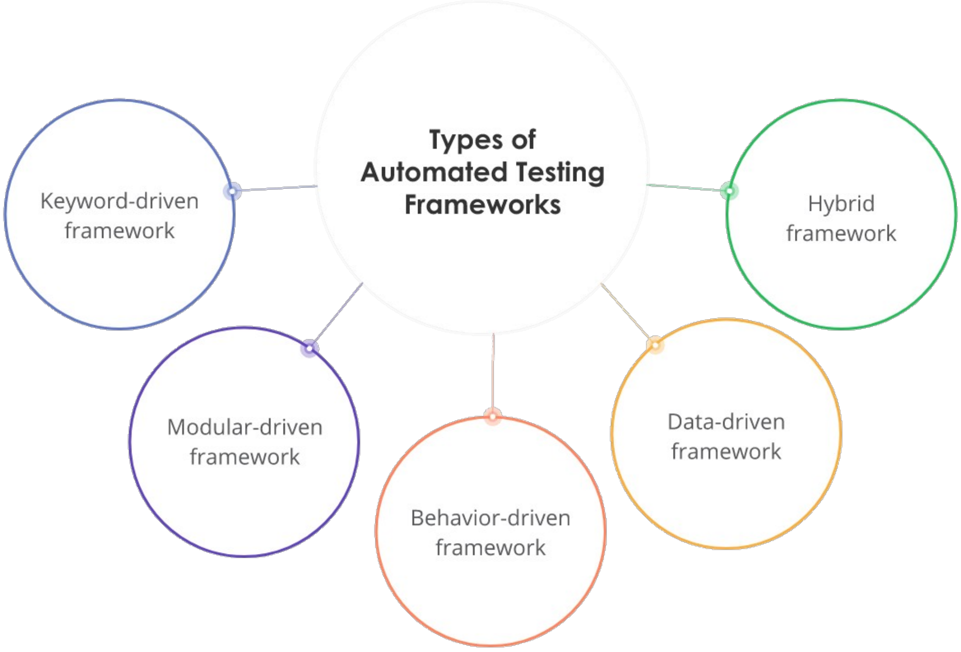


Рисунок 2.1 Системи автоматизації

Keyword-driven framework. Це популярний підхід до автоматизованого функціонального тестування. Логіка тестування розбивається на ключові слова та функції. ці ключові слова зберігаються в окремому сховищі об'єктів та викликаються за потреби провести тест. Цей метод автоматизації тестування програмного забезпечення базується на використанні ключових слів або ключових фраз для опису тестових сценаріїв. Ключові слова представляють собою абстрактні команди чи дії, які визначаються для виконання певних дій або перевірки певних умов в програмному продукті. Цей підхід використовує окремі таблиці або файли, де кожен рядок представляє собою конкретний тестовий сценарій або набір дій, в яких використовуються ці ключові слова. Це спрощує процес створення, управління та редагування тестів, оскільки зміни у тесті можуть бути внесені безпосередньо в таблицю ключових слів, зберігаючи структуру тесту не змінюючись. Крім того, цей підхід дозволяє розбити тестові скрипти на менші частини для більшої перевикористовуваності та підтримки.

Modular-driven framework. Модульно-орієнтований фреймворк для автоматизації тестування базується на розбитті програмного продукту на окремі модулі або компоненти. Кожен модуль відповідає за тестування певної функціональності чи частини програми. Це дозволяє автоматизаторам створювати набори тестових скриптів, які тестують окремі модулі незалежно один від одного. Модульні тести можуть бути легко перевикористані, оскільки вони не залежать від інших частин програми. Цей підхід сприяє більшій стабільності та підтримці коду, адже зміни у функціональності можуть вноситися, зберігаючи цілісність тестувальних скриптів.

Behavior-driven framework. Методика behavior-driven development (BDD) включає в себе використання узгодженої мови для спілкування між учасниками проєкту, такими як розробники, тестувальники та замовники, для уточнення очікувань від програмного забезпечення. Фреймворк BDD зосереджується на описі поведінки системи через сценарії та умови, що дозволяє створювати тести, орієнтовані на бізнес-логіку програми. Ключовою частиною BDD є використання "Given-When-Then" (Дано-Коли-Тоді) синтаксису для опису очікуваної поведінки системи у відповідь на певні вхідні умови чи дії.

Data-driven framework. Підхід до автоматизації тестування, який розглядає дані як основний елемент тестів. Використовуючи цей метод, тестові сценарії відокремлюються від вхідних даних, що дозволяє легко маніпулювати тестами шляхом зміни вхідних даних без модифікації самого тесту. Це підвищує повторюваність тестів та дозволяє проводити більше варіантів тестування з мінімальними змінами у коді.

Hybrid framework — будь-яка комбінація вищевказаних фреймворків AQA. Гібридний фреймворк поєднує в собі різні підходи до автоматизації тестування, такі як ключовий, модульний, даних-driven та інші. Це дозволяє розробникам тестів використовувати найкращі практики кожного методу, щоб створити більш гнучкий та ефективний фреймворк для тестування програмного забезпечення, враховуючи конкретні вимоги та потреби проєкту.

Набуття розуміння різноманітних типів тестування, що мають бути впроваджені в конкретний фреймворк або систему, є важливим етапом для належної організації та виконання тестувальних процедур. Це дозволяє не лише чітко визначити обсяг і види тестування, які необхідно впровадити, а й надає можливість належно управляти тестовими активностями та ресурсами. Врахування типів тестування у вихідних документах або специфікаціях фреймворку допомагає визначити правильний набір тестів для покриття різноманітних аспектів програмного забезпечення, зокрема, функціональних та нефункціональних вимог, що забезпечує якість продукту. Засновуючись на виборі інструментів, в даній системі буде реалізовано підтримку гібридного фреймворку, що може складатися з KD, MD та DD, оскільки BD вимагає вивчення ще додаткового синтаксису що ускладнить систему та збільшить час потрібний на розробку тестових сценаріїв.

Після встановлення типів тестування стає можливим розгляд різноманітних архітектур, оскільки це дає змогу визначити, які аспекти системи вимагають особливої уваги під час тестування. Це також дозволяє врахувати особливості архітектури програмного продукту при плануванні тестових сценаріїв та обрати найбільш ефективні стратегії тестування для конкретної архітектури. Такий підхід сприяє забезпеченню більш повного охоплення тестування та впевненості у якості програмного продукту в рамках відповідної архітектури.

Архітектура, у контексті програмного забезпечення, представляє собою важливий концептуальний фундамент, що визначає структуру, взаємозв'язки та внутрішні компоненти програми або фреймворку. Вона відображає ментальну модель розробників і визначає як система повинна бути побудована для досягнення поставлених цілей. Зазвичай, архітектура визначається через фізичний розподіл компонентів програмного забезпечення, хоча вона може також включати абстрактні концепції та принципи організації програмних компонентів. Основною метою розробки архітектури є сприяння підвищенню підтримуваності та зменшенню складності програмного продукту.

Підвищення підтримуваності в архітектурі програмного забезпечення реалізується шляхом створення структури, яка враховує ймовірність частої зміни деяких компонентів. Це означає ретельне ідентифікування та виділення частин системи, які ймовірно будуть піддаватися частим змінам, що дозволяє окремим частинам програми адаптуватися до нових вимог швидше та безпечніше. Диференціація між рідко змінюваними та часто модифікованими компонентами дозволяє забезпечити більшу гнучкість та швидкість реагування на зміни, збільшуючи загальну підтримуваність програмного забезпечення.

Зменшення складності програмного коду в архітектурі досягається через дрібну сегментацію програми на окремі елементарні частини або компоненти. Цей підхід спрощує структуру програмного коду та робить його більш зрозумілим та легшим для обслуговування. Спрощена структура полегшує розуміння коду для обслуговуючого персоналу, сприяючи виконанню завдань з підтримки та розвитку системи. Такий підхід сприяє зменшенню вартості підтримки програмного продукту та підвищує його стабільність та робочу ефективність.

Найбільш відомі типи архітектури фреймворку автоматизації тестування:

- No architecture
- Layered
- Layered pluggable
- Hexagonal

Підхід "No architecture" (без архітектури) у контексті створення фреймворку для автоматизації тестування відображається в відсутності конкретної архітектурної структури, що визначає організацію та взаємозв'язки між компонентами програми. У такому підході розробка коду відбувається без чіткого планування або строго розподілу на компоненти. Замість цього, розробники можуть спрямовувати свої зусилля на написання коду в межах визначених завдань без попередньої організації або структуризації програми.

Деякі ключові аспекти цієї архітектури включають:

- Безструктурність. Немає чіткого розділення на шари чи компоненти.

Розробка ведеться в стилі "на льоту" без чіткого плану.

- Цей підхід може бути простим, особливо для невеликих проєктів або тестів, де немає потреби в складній архітектурі.
- Невеликий обсяг коду. Зазвичай використовується в ситуаціях, коли функціонал автоматизації тестування не потребує великої інфраструктури.
- Відсутність структури може зробити проєкт важким для підтримки і розширення зі зростанням обсягу тестів чи складності проєкту.
- Відсутність чітких правил може призводити до хаотичного коду, що ускладнює розуміння та модифікацію.

Використання цієї архітектури обмежується сценаріями, де простота та високий ступінь експериментування є більш пріоритетними, ніж чітка організація та масштабованість. Вона може мати певні переваги у випадках, коли робиться акцент на простоту та швидкість розробки, або коли проєкт є малим за обсягом та не потребує масштабування. Однак, у більших проєктах або при роботі в команді, відсутність організованої архітектурної структури може стати перешкодою для розвитку та підтримки програмного забезпечення. Тому вибір підходу без архітектури варто розглядати з урахуванням специфіки проєкту та його майбутніх потреб.

Архітектурний підхід Layered (шарований) у фреймворку для тестування встановлює структуру коду програмного забезпечення у вигляді різних рівнів або шарів, де кожен шар має відповідальність за конкретний аспект функціоналу і представляє собою логічну групу компонентів або модулів, які спеціалізуються на виконанні певних завдань чи операцій.

Основні риси цієї архітектури включають:

- Ієрархічна структуру: Фреймворк розділений на логічні шари, зазвичай три: інтерфейсний (presentation), бізнес-логічний (business logic), і доступ до даних (data access). Кожен з цих шарів відповідає за конкретні обов'язки.
- Розділення відповідальностей: Кожен шар виконує певну функцію та несе відповідальність за свою область. Наприклад, інтерфейсний шар відповідає

за взаємодію з користувачем, бізнес-логічний - за обчислення, а доступ до даних - за роботу з базою даних.

- Зручність тестування: Шарована архітектура сприяє зручності тестування, оскільки окремі шари можна тестувати ізольовано. Це дозволяє виявляти та виправляти помилки в окремих компонентах без впливу на решту системи.
- Модульність: Кожен шар є модульним, що полегшує розширення та модифікацію системи. Модульність дозволяє підтримувати код в хорошому стані і спрощує обслуговування.
- Сховище залежностей: Кожен шар може залежати від інших шарів, але зазвичай найнижчий шар (доступ до даних) є найменш залежним.
- Підтримка високого рівня абстракції: Кожен шар може служити абстракцією для інших шарів, забезпечуючи високий рівень абстракції та розділення концепцій.

Основна мета шарованої архітектури полягає у розділенні функціоналу програми на взаємодіючі рівні. Цей підхід дозволяє створити модульну структуру, що сприяє підвищенню перевикористовуваності компонентів та полегшує тестування окремих шарів незалежно один від одного. Крім того, шарована архітектура сприяє збереженню високого рівня зрозумілості, адже окремі шари можуть бути розроблені та модифіковані незалежно, що спрощує процес розвитку та підтримки програмного забезпечення. Використання шарованої архітектури дозволяє створювати масштабовані, розширювані та легко тестовані системи.

Архітектура Layered pluggable є розширенням базового підходу Layered до створення фреймворку для тестування. Цей тип архітектури спрямований на створення системи, яка дозволяє легко додавати новий функціонал або вносити зміни до існуючого без значного впливу на інші шари програмного забезпечення.

Деякі ключові аспекти цієї архітектури включають:

- Аналогічно до базової шарованої архітектури, інтерфейс, бізнес-логіка і доступ до даних розділені на відокремлені шари.
- Ключовою особливістю є можливість додавати новий функціонал через

"плагіни" або "модулі" без прямого впливу на існуючий код.

- Кожен плагін може бути розроблений та тестований ізольовано від інших, що полегшує управління кодом та використання різних розробників для створення різноманітного функціоналу.
- Додаючи нові шари або плагіни, можна легко змінювати функціонал системи, не переписуючи весь код. Це забезпечує високий рівень гнучкості та модульності.
- Зручність для розробників. Розробники можуть спрямовувати свої зусилля на конкретні компоненти чи функціонал, що дозволяє їм працювати ефективно та швидко вносити зміни.
- Ця архітектура дозволяє легко підтримувати різноманітні вимоги та варіації функціоналу для різних клієнтів чи сценаріїв використання.

Даний підхід сприяє великій гнучкості та розширюваності фреймворку, оскільки новий функціонал може бути впроваджений без значних змін у основних компонентах системи. Крім того, використання плагінів у Layered pluggable дозволяє розширювати функціональність системи без необхідності модифікації вихідного коду, що сприяє збереженню стабільності та простоти управління програмними компонентами. Використання цієї архітектури дозволяє створювати динамічні та розширювані системи, які легко адаптуються до змін в вимогах і дозволяють розробникам ефективно працювати над окремими компонентами функціоналу.

Архітектурний підхід Hexagonal (також відомий як Ports and Adapters) у контексті розробки програмного забезпечення пропонує модель, що базується на концепції внутрішньої системи з чітко визначеними портами і зовнішньої моделі з адаптерами. Цей підхід виник як спроба створити системи, які були б менш залежними від конкретних технологій, фреймворків чи зовнішніх інтерфейсів, а замість цього надавали б можливість ефективно масштабуватися та змінюватися.

Основні особливості цієї архітектури:

- Система розділяється на внутрішню модель (бізнес-логіка, яка не залежить від зовнішнього середовища) та зовнішню модель (залежну від зовнішніх компонентів, таких як база даних, інтерфейс користувача).

- Порти визначають інтерфейси для взаємодії між внутрішньою та зовнішньою моделями. Адаптери реалізують ці порти та забезпечують конкретну реалізацію взаємодії з зовнішніми компонентами.
- Бізнес-логіка не залежить від конкретних технологій інфраструктури. Це полегшує тестування та забезпечує гнучкість.
- Бізнес-логіка може бути тестована ізольовано від зовнішніх компонентів, що полегшує тестування та забезпечує надійність.
- Зовнішні адаптери можуть бути легко замінені без зміни внутрішньої бізнес-логіки, що дозволяє ефективно змінювати технології чи компоненти.
- Система може легко підтримувати різні типи інтерфейсів (наприклад, веб-інтерфейс, консольний інтерфейс) через використання різних адаптерів.

Отже, архітектурний підхід "Hexagonal" або "Ports and Adapters" дозволяє створювати системи з високим рівнем залежності та можливістю змін, що сприяє підтримці та розвитку програмного забезпечення на різних етапах його життєвого циклу. Даний підхід позитивно впливає на розробку систем, які мають високий степінь модульності, роблячи їх більш легкими для розуміння, тестування та розвитку.

Проаналізувавши наявні типи архітектур, було визначено що архітектурні підходи Layered та Layered pluggable є сприятливими для розробки фреймворку для тестування вбудованих систем оскільки Layered архітектура дозволяє розділити систему на логічні шари, такі як інтерфейс, бізнес-логіка та доступ до даних, що полегшує керування складністю системи. Layered pluggable архітектура розширює Layered концепцію, дозволяючи додавати новий функціонал без значного впливу на існуючі шари. Це забезпечує більшу гнучкість у розробці та підтримці фреймворку, що може бути важливим для вбудованих систем, які можуть вимагати швидких змін і підтримку різних конфігурацій. Але це підвищить складність системи, тому було прийнято рішення щодо реалізації саме layered архітектури, яка за потреби може бути розширена до layered pluggable.

2.2 Детальний опис шаблону

Компоненти в рамках шаблону багаторівневої архітектури організовані в горизонтальні рівні, кожен з яких виконує певну роль у програмі (наприклад, логіка презентації або бізнес-логіка). Хоча шаблон багаторівневої архітектури не визначає кількість і типи рівнів, які повинні існувати в шаблоні, більшість багатошарових архітектур складаються з чотирьох стандартних рівнів: презентації, бізнесу, збереження та бази даних (рис. 2.2). У деяких випадках бізнес-рівень і рівень збереження об'єднуються в один бізнес-рівень, особливо коли логіка збереження вбудована в компоненти бізнес-рівня. Таким чином, менші програми можуть мати лише три рівні, тоді як більші та складніші бізнес-програми можуть містити п'ять або більше рівнів.

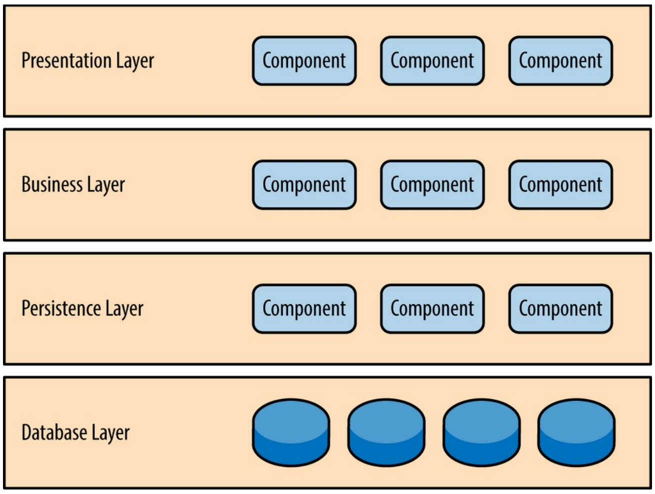


Рисунок 2.2

Кожен рівень шаблону багаторівневої архітектури має певну роль і відповідальність у програмі. Наприклад, презентаційний рівень буде відповідати за обробку всього інтерфейсу користувача та логіки зв'язку браузера, тоді як бізнес-рівень буде відповідати за виконання конкретних бізнес-правил, пов’язаних із запитом. Кожен рівень в архітектурі утворює абстракцію навколо роботи, яку необхідно виконати, щоб задовольнити певний бізнес-запит.

Наприклад, презентаційному рівню не потрібно знати або турбуватися про те, як отримати дані клієнтів; потрібно лише відобразити цю інформацію на екрані в певному форматі. Так само бізнес-рівню не потрібно турбуватися про те, як відформатувати дані клієнтів для відображення на екрані або навіть про те, звідки вони надходять; йому потрібно лише отримати дані з рівня збереження, виконати бізнес-логіку з даними (наприклад, обчислити значення або агрегувати дані) і передати цю інформацію на рівень презентації.

Однією з потужних особливостей layered архітектури є розподіл завдань між компонентами. Компоненти в межах певного шару мають справу лише з логікою, яка відноситься до цього шару. Наприклад, компоненти в шарі презентації мають справу лише з логікою презентації, тоді як компоненти, що знаходяться в бізнес-шарі, мають справу лише з бізнес-логікою. Такий тип класифікації компонентів полегшує створення ефективних моделей ролей та відповідальності у вашій архітектурі, а також спрощує розробку, тестування, управління та підтримку додатків, що використовують цей патерн архітектури, завдяки чітко визначеним інтерфейсам компонентів та обмеженому обсягу компонентів.

На рисунку 2.3, що кожен з шарів архітектури позначений як закритий. Це важливе поняття у шаблоні багаторівневої архітектури. Закритий рівень означає, що коли запит рухається з рівня на рівень, він повинен пройти через рівень, що знаходиться безпосередньо під ним, щоб потрапити на наступний. Наприклад, запит, що походить з рівня представлення, повинен спочатку пройти через бізнес-рівень, а потім через рівень збереження, перш ніж потрапити на рівень бази даних.

То чому б не дозволити шару представлення прямий доступ до шару збереження або до шару бази даних? Зрештою, прямий доступ до бази даних з рівня представлення набагато швидший, ніж проходження через купу непотрібних рівнів, щоб отримати або зберегти інформацію з бази даних. Відповідь на це питання лежить у площині ключової концепції, відомої як рівні ізоляції.

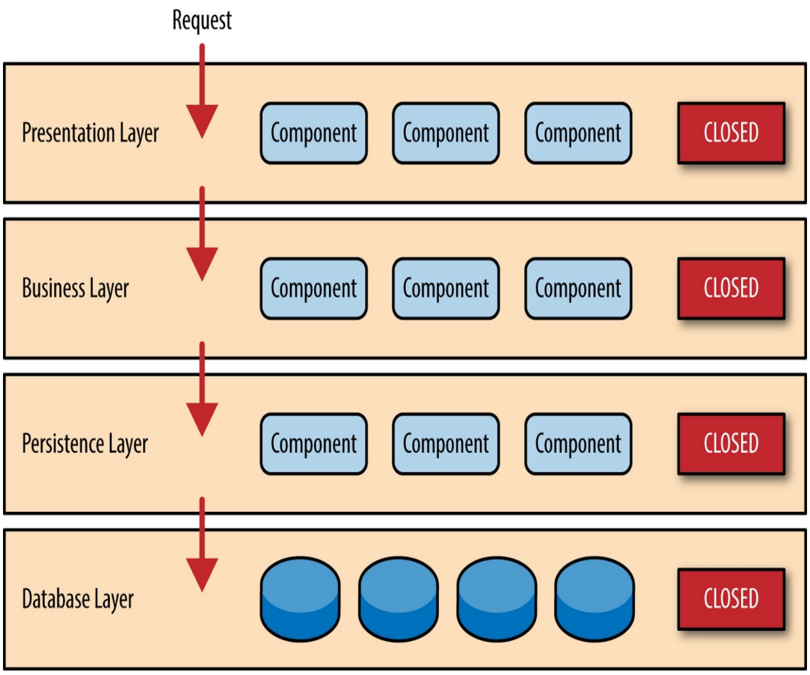


Рисунок 2.3 Рух запиту через рівні

Концепція рівнів ізоляції означає, що зміни, зроблені в одному шарі архітектури, як правило, не впливають або мінімально впливають на компоненти в інших шарах. Якщо дозволити шару представлення прямий доступ до шару збереження, то зміни, внесені до SQL у шарі збереження, вплинуть як на бізнес-шар, так і на шар представлення, що призведе до створення дуже тісно пов'язаного додатку з великою кількістю взаємозалежностей між компонентами. Такий тип архітектури потім стає дуже важко і дорого змінювати.

Концепція ізольованих шарів також означає, що кожен шар є незалежним від інших, таким чином, маючи мало або зовсім не маючи знань про внутрішню роботу інших частин архітектури. Якщо припустити, що контракти (наприклад, модель), які використовуються між презентаційним і бізнес-шаром, залишаються незмінними, то бізнес-шар не зазнає впливу рефакторингу і залишається повністю незалежним від типу фреймворку користувацького інтерфейсу, який використовується презентаційним шаром.

Хоча закриті шари сприяють створенню рівнів ізоляції і, отже, допомагають ізолювати зміни в архітектурі, бувають випадки, коли є сенс зробити певні шари відкритими. Наприклад, є потреба додати до архітектури рівень спільних служб, що містить спільні сервісні компоненти, до яких мають доступ компоненти бізнес-рівня (наприклад, класи утиліт для роботи з даними та рядками або класи аудиту та реєстрації). Створення рівня служб зазвичай є гарною ідеєю у цьому випадку, оскільки архітектурно він обмежує доступ до спільних служб бізнес-рівнем (а не рівнем представлення). Без окремого шару архітектурно немає нічого, що обмежувало б рівень представлення від доступу до цих спільних служб, що ускладнює керування цим обмеженням доступу.

У цьому прикладі новий рівень служб, найімовірніше, розташовуватиметься нижче бізнес-рівня, щоб показати, що компоненти цього рівня служб недоступні з рівня представлення. Однак, це створює проблему, оскільки бізнес-рівень тепер повинен пройти через рівень служб, щоб дістатися до рівня збереження, що не має жодного сенсу. Це давня проблема багаторівневої архітектури, і вона вирішується шляхом створення відкритих шарів в архітектурі.

Як показано на рисунку 2.4, рівень сервісів в даному випадку позначений як відкритий, що означає, що запитам дозволено обходити цей відкритий рівень і переходити безпосередньо до рівня, розташованого під ним. У наступному прикладі, оскільки рівень сервісів є відкритим, бізнес-шару тепер дозволено оминати його і переходити безпосередньо до рівня персистентності, що є цілком логічним.

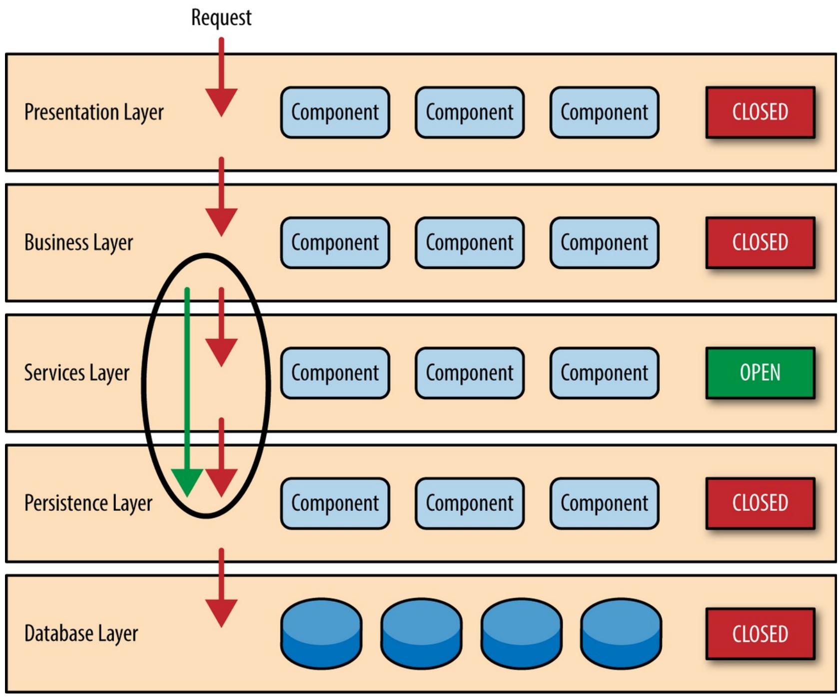


Рисунок 2.4 Відкритий рівень сервісів

Використання концепції відкритих і закритих рівнів допомагає визначити взаємозв'язок між рівнями архітектури і потоками запитів, а також надає проєктувальникам і розробникам необхідну інформацію для розуміння різних обмежень доступу до рівнів в архітектурі. Якщо не задокументувати або не повідомити належним чином, які рівні архітектури є відкритими, а які закритими (і чому), то це зазвичай призводить до створення тісно пов'язаних і крихких архітектур, які дуже складно тестувати, підтримувати і розгортати.

Щоб проілюструвати, як працює багаторівнева архітектура, розглянемо запит від бізнес-користувача на отримання інформації про клієнтів для конкретної особи, як показано на рисунку 2.5. Чорні стрілки показують запит, який надходить до бази даних для отримання даних про клієнта, а червоні стрілки показують відповідь, яка надходить назад на екран для відображення даних. У цьому прикладі інформація про клієнта складається як з даних клієнта, так і з даних замовлень (замовлень, розміщених клієнтом).

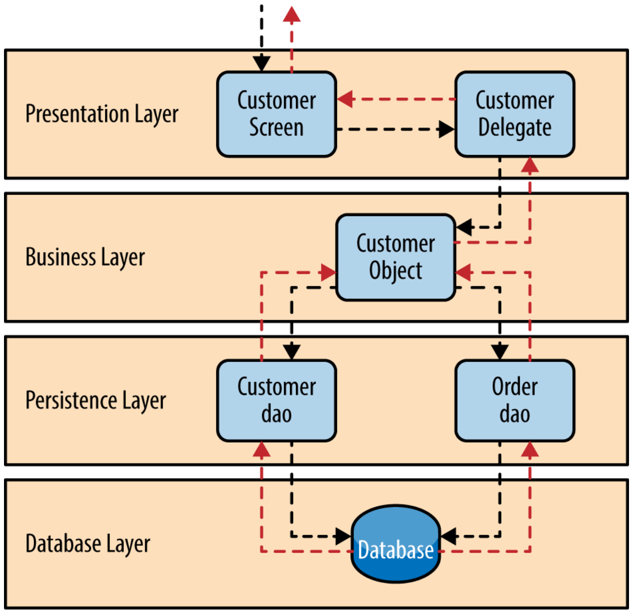


Рисунок 2.5 Запит бізнес-користувача

Екран клієнта відповідає за прийняття запиту і відображення інформації про клієнта. Він не знає, де знаходяться дані, як їх отримати або скільки таблиць бази даних потрібно запитати, щоб отримати ці дані. Як тільки екран клієнта отримує запит на отримання інформації про клієнта для конкретної особи, він перенаправляє цей запит до модуля делегата клієнта. Цей модуль відповідає за те, які модулі бізнес-рівня можуть обробити цей запит, а також за те, як дістатися до цього модуля і які дані йому потрібні (договір).

Об'єкт клієнта в бізнес-рівні відповідає за агрегування всієї інформації, необхідної для бізнес-запиту (в даному випадку для отримання інформації про клієнта). Цей модуль звертається до модуля customer dao (об'єкт доступу до даних) на рівні персистентності, щоб отримати дані про клієнта, а також до модуля order dao, щоб отримати інформацію про замовлення. Ці модулі, в свою чергу, виконують SQL-запити для отримання відповідних даних і передають їх назад до об'єкта клієнта в бізнес-рівні. Після того, як об'єкт клієнта отримує дані, він агрегує їх і передає цю інформацію назад до делегата клієнта, який потім передає ці дані на екран клієнта, щоб представити їх користувачеві.

Однак, при виборі цього патерну є кілька моментів, які слід враховувати з точки зору архітектури.

Перше, на що слід звернути увагу - це те, що відомо як антипатерн "архітектурна вирва". Цей патерн описує ситуацію, коли запити проходять через декілька рівнів архітектури як просту наскрізну обробку з невеликою кількістю логіки, що виконується на кожному рівні, або взагалі без неї. Наприклад, припустимо, що рівень представлення відповідає на запит користувача про отримання даних про клієнта. Рівень представлення передає запит на бізнес-рівень, який просто передає запит на рівень персистентності, який потім робить простий SQL-запит до рівня бази даних, щоб отримати дані клієнта. Потім дані передаються назад у стек без додаткової обробки або логіки для агрегування, обчислення або перетворення даних.

Кожна багаторівнева архітектура матиме принаймні кілька сценаріїв, які підпадають під антипатерн "воронка архітектури". Ключовим моментом, однак, є аналіз відсотка запитів, які потрапляють в цю категорію. Правило 80-20 зазвичай є гарною практикою, якої слід дотримуватися, щоб визначити, чи присутній антипатерн "архітектурна воронка" чи ні. Типовою є ситуація, коли близько 20 відсотків запитів є простою наскрізною обробкою, а 80 відсотків запитів мають певну бізнес-логіку, пов'язану із запитом. Однак, якщо виявиться, що це співвідношення зворотне і більшість запитів є простою наскрізною обробкою, можна розглянути можливість зробити деякі шари архітектури відкритими, маючи на увазі, що буде важче контролювати зміни через відсутність ізоляції шарів.

Ще одна особливість шаблону багаторівневої архітектури полягає в тому, що він має тенденцію до створення монолітних додатків, навіть якщо відбудеться розділення рівня презентації та бізнес-рівеня на окремі одиниці. Хоча це може не викликати занепокоєння для деяких додатків, це створює деякі потенційні проблеми з точки зору розгортання, загальної стійкості та надійності, продуктивності та масштабованості.

З точки зору багаторівневої архітектури, рівні каркасу автоматизації будуть розділені таким чином як представлено на рисунку 2.6:

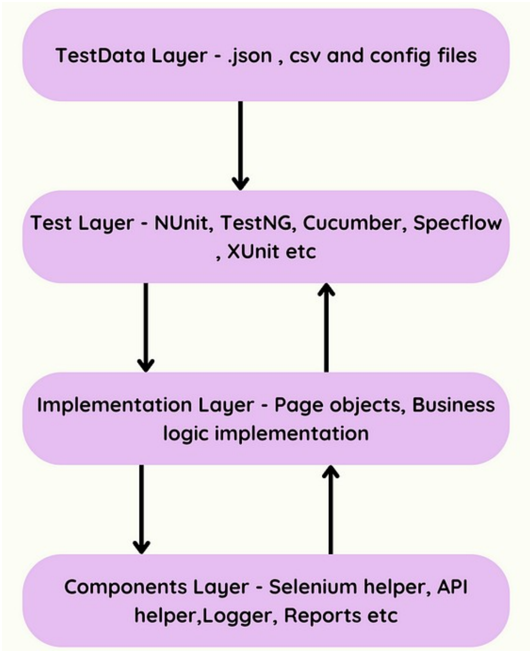


Рисунок 2.6 Каркас рівнів системи авоматизації

- Презентаційний рівень. Це найвищий рівень в багаторівневій архітектурі. тому в автоматизації тестування тестовий рівень буде нашим презентаційним рівнем, який діятиме як інтерфейс для користувачів.
- Бізнес-рівень. Доменну логіка розміщується на бізнес-рівні. Він також діє як міст між рівнями презентації та стійкості. Таким чином, в автоматизації тестування розробляється бізнес-логіку в класах сторінок, отже, об’єкти сторінок підпадають під цей бізнес-рівень.
- Рівень збереження — рівень збереження відповідає за зберігання даних і файлів, але в автоматизації тестування можна використовувати цей рівень як допоміжний рівень, який матиме різноманітні класи компонентів, як-от допоміжні класи Selenium, допоміжний засіб даних, допоміжний API, реєстратор тощо.
- Крім того, також потрібен шар для зберігання тестових даних. Це рівень даних, який буде безпосередньо взаємодіяти з тестовим рівнем. Щоб виконати певну дію, тестовий рівень передає необхідні дані на бізнес

рівень, який, у свою чергу, взаємодіє з рівнем збереження. Та при цьому важливо не додавати код до рівня тестових даних, оскільки це тільки ускладнить підтримку.

Відповідно до цього підходу, нижні рівні реагують лише на верхній рівень, тому до компонентних рівнів не слід звертатися безпосередньо з тестового рівня. Тестовий рівень викликає бізнес-логіку рівня реалізації, яка потім взаємодіє з компонентним рівнем.

Наступним етапом є реалізація функціональності фреймворку. В цьому контексті використовуються сучасні підходи до програмування, зокрема, об'єктно-орієнтоване програмування, шаблони проєктування та принципи SOLID. Під час розробки важливо враховувати вже розроблену архітектуру, принципи модульності та розширюваності фреймворку для подальшого його покращення та адаптації до змінних вимог.

2.3 Створення репозиторію

Створення git репозиторію для проєкту є першочерговим етапом у процесі розробки програмного забезпечення, оскільки надає можливість контролю версій, спільної роботи та збереження історії змін. Цей інструмент створює основу для керування кодом проєкту та сприяє спільній роботі кількох розробників, забезпечуючи зручний механізм відстеження змін та спільного узгодження робочих версій програми у відповідності до найсучасніших практик розробки.

– Для початку потрібно відкрити командний рядок (термінал) у каталозі проєкту та ввести команду що створить новий git репозиторій в поточній директорії.

  ```
  -> % git init
  ```

– Додати файли до репозиторію командою що позначає файли для подальшого коміту.

  ```
  -> % git add назва_файлу_або_директорії
  ```

- Ввести команду для створення коміту з попередньо підготовленими змінами:

```
-> % git commit -m "Опис змін у коміті"
```

- Це зафіксує зміни у репозиторії з коротким поясненням.
- Залогінитись до обраного хостингу git (наприклад, GitHub, GitLab, Bitbucket), створити новий репозиторій та отримати посилання на нього.
- Додати посилання на віддалений репозиторій до локального. Це дозволить синхронізувати зміни між локальним та віддаленим репозиторіями.

```
-> % git remote add назва_віддаленого посилання_на_репозиторій
```

- Відправити свої коміти на віддалений репозиторій за допомогою команди:

```
-> % git push назва_віддаленого назва_гілки
```

Для даного проєкту було створено git репозиторій automation-framework на хостингу GitHub та виконано такий набір команд:

```
-> % mkdir automation-framework && cd automation-framework
-> % echo "# automation-framework" >> README.md
-> % git init
-> % git add README.md
-> % git commit -m "add README"
-> % git commit -m "add README"
-> % git branch -M main
-> % git remote add origin git@github.com:201dreamers/automation-framework.git
-> % git push -u origin main
```

2.4 Розробка структури папок

Наукові аспекти, які враховуються при розробці структури папок, полягають у впливі організації коду на архітектурні принципи програмування, такі як модульність, зручність використання, розширюваність та підтримка. Наприклад, використання чітко визначених розділів для окремих компонентів дозволяє легко визначати, змінювати та розширювати функціональні можливості фреймворку.

Структура папок впливає на процес розробки через полегшення розуміння коду та його взаємозв'язків, підвищення його читабельності та підтримки. Це сприяє швидкому доступу до потрібних ресурсів, спрощує роботу команди розробників та забезпечує консистентність під час розробки та підтримки фреймворку.

Крім того, належна структура папок забезпечує зручність впровадження нових функцій та модулів, оскільки вона дозволяє розробникам працювати над частинами системи паралельно, збільшуючи продуктивність розробки та сприяючи швидкому виявленню та виправленню помилок.

Структура:

- src/: Ця тека містить вихідний код фреймворку автоматизації, такий як бібліотеки, утиліти та загальні функції, що використовуються тестами. У цій папці можуть бути дві підпапки: основна і тестова.
- src/connectors/: Містить код для під’єднання та спілкування з тестовим девайсом можливими способами.
- src/device/: Містить файли implementation шару. Різноманітні обгортки для опису кроків тестування і роботи з девайсом.
- tests/: Ця папка містить власне тестові кейси, організовані у підпапки на основі відповідних функцій, модулів або компонентів.
- reports/: Містить вихідні звіти тестових запусків, включаючи журнали, скріншоти та метрики.
- resources/: Ця папка містить будь-які додаткові ресурси, необхідні для тестів, такі як зображення, відео або аудіофайли. Вона також може містити конфігурацію, звіти та тестові дані.
- resources/testdata/: Містить тестові дані в різних форматах, таких як JSON, XML або CSV. Вона також може містити конфігураційні файли для різних середовищ або тестових наборів.
- resources/lib/: Ця тека містить сторонні бібліотеки або залежності, необхідні

для фреймворку або тестів

- config/: Містить конфігураційні файли для фреймворку автоматизації, такі як налаштування тестового середовища, тестові дані або параметри виконання тестів. Вони можуть зберігатися як на рівні проєкту, так і в папці resources/.
- utils/: Ця папка містить утиліти та функції, які можуть бути використані тестами, наприклад, функції дати/часу або утиліти для маніпулювання файлами.

2.5 Початкове налаштування оточення

У процесі розробки фреймворку існує послідовність ключових кроків, які сприяють належній підготовці середовища для розробки та виконання тестів. На початку важливим пунктом є створення віртуального оточення, що забезпечує ізольоване середовище для проєкту.

```
-> % python -m venv venv
```

Де `-m venv` вказує на використання модуля `venv` для створення віртуального оточення, а `venv` є ім'ям обраного середовища.

Другим кроком є його активація. Команда

```
-> % source venv/bin/activate
```

активує віртуальне оточення, що робить його поточним для виконання команд у терміналі. Це дозволяє ізолювати та управляти пакетами та залежностями конкретного проєкту без впливу на глобальну конфігурацію Python.

І останнім етапом є встановлення необхідних пакетів за допомогою інструменту pip. Команда

```
-> % pip install pytest paramiko pyyaml pyserial
```

встановлює бібліотеки, необхідні для реалізації функціональності фреймворку, де pytest, paramiko та pyyaml представляють собою пакети для проведення тестування, роботи з SSH-з'єднаннями, Serial з’єднаннями та обробки

Yaml-файлів відповідно.

Ці кроки становлять основу для подальшої розробки у створеному віртуальному оточенні, дозволяючи забезпечити незалежність від глобальної конфігурації системи та забезпечити належну роботу фреймворку.

2.6 Розробка рівня components

Після визначення структури фреймворку та установки відповідного оточення, надалі розробка відбуватиметься шляхом ітераційного підходу на рівнях, що починаються з найнижчого. Першим пріоритетом у цьому процесі є Components layer, який включає в себе модулі, охоплюючи такий функціонал, як забезпечення зв'язку з вбудованими пристроями, використання різноманітних допоміжних класів та функцій, механізми логування, генерацію звітів та взаємодію з API.

2.6.1 Базовий модуль під’єднання до девайсу

В рамках розробки програмних продуктів, що взаємодіють з тестовими пристроями, створено модуль базового з'єднання base_connection.py. Цей модуль містить абстрактний клас BaseConnection, який служить базою для реалізації з'єднань за допомогою протоколів SSH та Serial.

BaseConnection є абстрактним класом, що містить методи та властивості, необхідні для створення взаємодії з тестовими пристроями через різноманітні типи з'єднань. Основною метою цього класу є створення загальної структури для підтримки функцій зчитування та запису даних з пристроїв.

Однією з ключових особливостей BaseConnection є те, що він містить абстрактні методи, такі як open_connection(), close_connection(), is_connected(), що вимагають реалізації у підкласах для конкретних протоколів з'єднання. Це дозволяє дотримуватись принципу поліморфізму та наслідування, щоб забезпечити гнучкість у виборі типу з'єднання.

BaseConnection також містить ряд методів, які забезпечують взаємодію з пристроєм. Наприклад:

- write(data: str): Запис даних до об'єкту з'єднання.
- read(count: float = 1) -> bytes: Зчитування даних з об'єкту з'єднання.
- send_command(command: str, timeout: float = 15, strip: bool = True) -> str: Надсилання команди до пристрою та очікування відповіді.

Ці методи дозволяють виконувати базові операції зчитування, запису та взаємодії з тестовим пристроєм безпосередньо через створений об'єкт з'єднання.

Додатково, клас BaseConnection містить методи для управління станом з'єднання:

- restart_connection(): Перезавантаження з'єднання.
- set_read_timeout(timeout: float): Встановлення часу очікування зчитування.
- Вони дозволяють контролювати стан з'єднання та його параметри для оптимального взаємодії з пристроєм.

Модуль base_connection.py є ключовим компонентом у загальній архітектурі системи, яка спрямована на спрощення взаємодії з різноманітними тестовими пристроями. Він надає абстракцію для роботи з різними типами з'єднань та методи для базової взаємодії з пристроями, що дозволяє розширювати та модернізувати систему без значних змін у вихідних кодах, дотримуючись принципів об'єктно-орієнтованого програмування. Для подальшого розвитку системи можливі напрямки досліджень включають розширення підтримки інших протоколів з'єднання, оптимізацію методів для підвищення продуктивності та дослідження механізмів автоматизації взаємодії з тестовими пристроями.

2.6.2 Модуль SSH з’єднання

Модуль ssh_connection.py є частиною системи для забезпечення з'єднання з пристроями за допомогою протоколу SSH. Його основна функція - забезпечити зручність та безпеку взаємодії з пристроями шляхом використання зашифрованого з'єднання та автентифікації.

Модуль SSHConnection використовує бібліотеку Paramiko для створення та управління з'єднанням з пристроєм через протокол SSH та надає інтерфейс для відкриття з'єднання, передачі команд та читання відповідей з пристрою через зашифроване з'єднання.

Клас SSHConnection приймає параметри, такі як IP-адреса, порт, ім'я користувача та пароль для встановлення з'єднання з пристроєм. Він наслідує функціональність абстрактного класу BaseConnection, що дозволяє створювати єдиний інтерфейс для роботи з різними типами з'єднань.

Метод open_connection() відповідає за встановлення з'єднання через SSH та налаштування об'єкту для спілкування з пристроєм. Метод is_connected() перевіряє стан з'єднання.

Клас SSHConnection надає методи для передачі даних та зчитування відповідей:

- write(data: str): Надсилання даних до об'єкту з'єднання.
- read(count: int = 1) -> bytes: Зчитування даних з об'єкту з'єднання.

Ці методи дозволяють взаємодіяти з пристроєм через з'єднання SSH, надаючи засоби для передачі команд та отримання відповідей. Клас також надає методи для управління параметрами з'єднання, такими як встановлення часу очікування читання (set_read_timeout(timeout: float)).

Дана частина є важливою складовою для забезпечення безпеки та надійності з'єднання з пристроями через SSH протокол. Використання бібліотеки Paramiko та структуровані методи дозволяють ефективно використовувати його для взаємодії з тестовими пристроями через зашифроване з'єднання. Можливі напрямки подальшого розвитку модуля включають розширення підтримки аутентифікації, оптимізацію часу відповіді та підтримку додаткових функцій SSH для більш широкого спектру використання.

2.6.3 Модуль serial з’єднання

Модуль serial_connection.py є компонентом системи для забезпечення з'єднання з пристроями через послідовний (serial) порт. Клас SerialConnection використовує бібліотеку pyserial і його основна задача - надати інтерфейс для відкриття з'єднання, передачі команд та отримання відповідей з пристрою через послідовний інтерфейс з можливістю фільтрації та очистки даних.

Клас SerialConnection приймає параметри, такі як порт, швидкість передачі (baudrate), ім'я користувача та пароль для встановлення з'єднання з пристроєм. Він також успадковує функціональність абстрактного класу BaseConnection, що дозволяє створювати єдиний інтерфейс для роботи з різними типами з'єднань.

Метод check_and_login() використовується для перевірки статусу логіну та виконання авторизації на пристрої через послідовний порт. Методи open_connection() та close_connection() відповідають відповідно за відкриття та закриття з'єднання з пристроєм.

Також реалізовані:

- write(data: str): Надсилання даних до об'єкту з'єднання через послідовний порт.
- read(count: int = 1) -> bytes: Зчитування даних з об'єкту з'єднання через послідовний порт.

Ці методи дозволяють ефективно взаємодіяти з пристроєм через послідовний інтерфейс, надаючи засоби для передачі команд та отримання відповідей.

Клас надає методи для управління параметрами з'єднання, такими як встановлення часу очікування читання (set_read_timeout(timeout: float)).

Модуль serial_connection.py є важливою складовою для забезпечення з'єднання з пристроями через послідовний порт. Використання бібліотеки serial та структуровані методи дозволяють ефективно використовувати та в майбутньому розширювати його.

Можливі напрямки подальшого розвитку модуля включають розширення підтримки різних конфігурацій послідовного порту, оптимізацію часу відповіді та підтримку додаткових функцій для розширення можливостей взаємодії з пристроями через послідовний порт.

Загальна діаграма класів що реалізовані для зв'язку з девайсом показана на рисунку 2.7

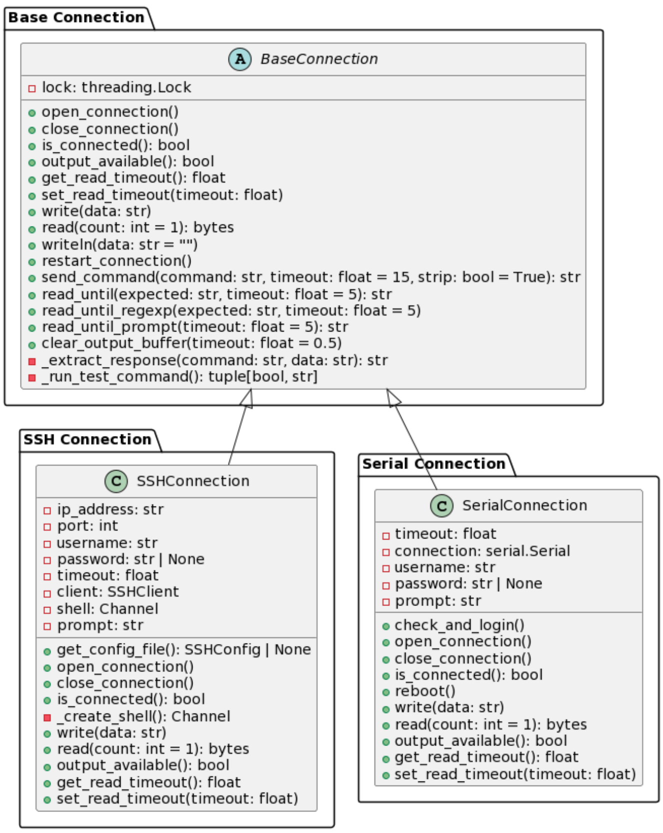


Рисунок 2.7 Діаграма класів конекторів

2.6.4 Налаштування pytest та генерації репортів

Подальша розробка буде проводитись вже з інтеграцією фреймворку pytest, тому налаштування є базовим кроком що потрібно виконати на даному етапі. Його налаштування виконується через конфігураційні файли pytest.ini або прихований формат .pytest.ini та використання різноманітних плагінів що розширюють базовий функціонал.

Розробка конфігураційного файлу.

```
[pytest]
testpaths = tests
pythonpath = .
cache_dir = reports/.pytest_cache

timeout = 300
durations = 5

log_file = reports/pytest-logs.log
log_format = %(asctime)s %(message)s
log_date_format = %Y-%m-%dT%H:%M:%S

addopts = "--pdbcls=IPython.terminal.debugger:Pdb --tb=long"

markers =
    slow
    serial_only
    ssh_only
    smoke
```

- Параметри testpaths та pythonpath визначають місцезнаходження тестів та необхідних залежностей. Це робить тестування більш систематичним та зручним для управління.
- Параметр cache_dir дозволяє зберігати проміжні результати, зменшуючи час, необхідний для повторного виконання тестів.
- Параметр timeout дозволяє установлювати максимальний час виконання тестів, що корисно для уникнення зависання в разі проблемних тестових сценаріїв.
- Параметри log_file, log_format, log_date_format визначають формат та місце збереження логів, що спрощує аналіз результатів тестування та відладку.
- Параметр addopts дозволяє передавати додаткові опції командному рядку pytest, що може бути корисним для налаштування специфічних опцій та

інструментів.

- Маркери, які використовуються у роботі – slow, serial_only, ssh_only, smoke – допомагають класифікувати тести та краще управляти ними відповідно до їхньої призначеності.
- Плагіни для розширення можливостей pytest
- pytest-instafail надає зручну функціональність швидкого перегляду помилок під час тестування, автоматично виводячи їх інформацію, що полегшує відлагодження коду.
- pytest-timeout дозволяє встановлювати таймаути для окремих тестів або для сесії в цілому, що допомагає управляти часом виконання, особливо в умовах, коли тестування може затягуватися.
- allure-pytest – це плагін, який інтегрує функціональність генерації докладних та інформативних звітів про виконані тести за допомогою фреймворку Allure. Це сприяє створенню зручного інтерфейсу для аналізу результатів тестування.
- pytest-check дозволяє використовувати додаткові перевірки, які можуть бути корисними для детальнішого тестування функціоналу. Це може включати перевірку відповідності деяких умов чи очікцувань, що робить тести більш повними. Також він надає додаткові засоби для створення тестів шляхом параметризації, допомагаючи поліпшити рівень покриття коду та надійність тестів.

Загалом використання конфігураційних файлів, таких як ini файли, та плагінів, дозволяють швидко масштабувати функціональні можливості фреймворку, і, за потреби, кожен проєкт може змінювати ці опції під себе, а також додавати нові.

Паралельно з налаштуванням pytest, потрібно також налаштувати генерацію репортів, адже вона забезпечує зручний та структурований спосіб візуалізації результатів виконання тестів. Це важливий етап у процесі розробки програмного забезпечення з кількох причин:

- Генерація репортів надає зручний засіб для представлення результатів тестування в зрозумілій та структурованій формі. Це дозволяє швидше та ефективніше оцінювати стан програми, адже інформація про пройдені та провалені тести, а також про природу помилок, відображається у зручному форматі.
- Згенеровані репорти допомагають розробникам та інженерам швидко виявляти та аналізувати проблеми у програмному коді або в тестових сценаріях. Це дозволяє оперативно реагувати на помилки, виправляти їх та вдосконалювати програмне забезпечення.
- Генерація репортів у формі, яка легко читається та розуміється, сприяє покращенню співпраці між членами команди розробників. Репорти можуть бути передані для аналізу іншим інженерам або стейкхолдерам, що спрощує спільну роботу над виявленням та вирішенням проблем.
- Вони надають об'єктивну основу для прийняття рішень щодо якості програмного забезпечення та допомагають здійснювати аналіз ефективності тестів, виявляти тенденції у якості коду та допомагають у плануванні подальших дій для покращення якості продукту.

Як фреймворк для генерації звітів буде використано allure, оскільки він спеціально розроблений для автоматизованих тестів та має чудову інтеграцію з pytest. Він надає засоби для створення зручних та інформативних звітів, які дозволяють аналізувати результати тестування з високою точністю та ефективністю.

Allure спрямований на полегшення процесу аналізу результатів тестування, надаючи звіти, що відображають якісні показники тестів у зручному для сприйняття вигляді. Основні функціональні особливості:

- Allure генерує звіти, які містять різноманітну інформацію про виконання тестів: від кількості пройдених і провалених тестів до статистики про час виконання та інші метрики.
- Фреймворк надає можливість візуального представлення результатів

тестування у вигляді графіків, діаграм та інших візуальних елементів, що полегшує аналіз та порівняння результатів.

- Allure є універсальним та підтримує інтеграцію з різними мовами програмування та фреймворками тестування, включаючи Python з pytest.
- Після запуску тестів з підтримкою Allure, фреймворк збирає дані про виконання тестів та їх результати, створюючи файли, що містять інформацію у форматі, придатному для генерації звіту.

Для перегляду звіту Allure використовують спеціальні інструменти, наприклад, Allure Command Line Interface або інші програми, які дозволяють аналізувати та інтерактивно взаємодіяти з результатами тестування. Зазвичай, використання цих інструментів дає можливість детально досліджувати результати тестування, переглядати статистику, а також виконувати фільтрацію та сортування даних для отримання більш точного аналізу.

Останньою частиною components шару, яку потрібно розробити — це модуль логування. Логування у фреймворку для тестування вбудованих пристроїв є важливим і потрібним з кількох причин:

- Логування дозволяє відстежувати виконання тестів і фіксувати помилки. Це надає можливість швидко знаходити та усувати проблеми під час розробки та підтримки коду.
- Записування логів під час тестування дозволяє зберігати історію запусків тестів, включаючи інформацію про виконані кроки, результати тестів, помилки і виключення, що спрощує аналіз та відлагодження в майбутньому.
- Логи можуть містити дані про час виконання кожного кроку тесту, ресурси, використані під час виконання, що дозволяє оцінювати продуктивність програмного забезпечення та виявляти можливі проблеми з продуктивністю.
- Логування може стати частиною документації тестів, описуючи послідовність виконаних дій та очікувані результати. Така інформація є корисною для розуміння тестів, особливо для нових членів команди або під

час підтримки системи.

– Збирання даних з логів протягом тривалого часу дозволяє аналізувати тренди, виявляти проблемні місця та забезпечувати звітність про стан системи чи тестових наборів.

Розроблений модуль для логування використовує бібліотеку loguru та забезпечує можливість створення та налаштування логерів для виведення повідомлень на стандартні потоки виведення (`stdout` та `stderr`). При розробці використано об'єктно-орієнтований підхід до логування, що дозволяє створювати та налаштовувати різні логери з різними рівнями логування та форматуванням повідомлень. Підхід до створення логерів в системі буде відображеним у три окремі категорії звіту allure.

У коді визначено дві константи енумерацій: LoggerName та LogLevel. LoggerName містить значення stdout та stderr для вказання стандартних потоків виведення. LogLevel включає в себе різні рівні логування, такі як TRACE, DEBUG, INFO, SUCCESS, WARNING, ERROR та CRITICAL. Також присутня функція `create_filter`, яка створює фільтр для логера за значенням LoggerName, дозволяючи зв'язувати різні логери з конкретними параметрами.

Налаштування логерів відбувається через виклик методу logger.configure. Логер використовує два обробники (handlers) для stdout та stderr, кожен з яких має свою власну конфігурацію, включаючи рівень логування, формат та фільтр, що використовує функцію create_filter для забезпечення відповідності записів до конкретного логера. В кінці створюються два окремі логери для stdout та stderr відповідно, використовуючи метод logger.bind для пов'язання їх з відповідними іменами LoggerName.STDOUT та LoggerName.STDERR.

2.7 Розробка рівня implementation

Враховуючи попередньо визначений Components layer, наступним етапом є розробка Implementation layer, який уособлює бізнес-логіку та обгортки, що створюються для спрощення та оптимізації використання існуючих консольних команд.

На відміну від Components layer, який фокусується на рівні доступу до пристроїв та різноманітних функцій, Implementation layer відповідає за впровадження та реалізацію високорівневих завдань, орієнтованих на специфіку вбудованих пристроїв. Зокрема, Implementation layer спрямований на створення абстракцій, які не лише полегшують інтеракцію з пристроями, але й забезпечують більш зрозумілі та читабельні шари для взаємодії з ними. Це включає в себе розробку функцій, які надають зручний інтерфейс для використання функціоналу вбудованих пристроїв, забезпечуючи високий рівень простоти використання для користувача фреймворку.

Для розв’язання задач які повинен виконувати даний шар було розроблено модуль device_lib.py Він представляє собою концептуальну структуру, спрямовану на вирішення проблеми абстракції консольних команд вбудованих пристроїв через методи та функції Python. Його головна мета полягає у створенні інтерфейсу, який спрощує взаємодію з пристроями, дозволяючи розробникам використовувати більш читабельні методи та функції для виклику консольних команд.

Цей модуль складається з класів Command та DeviceConnection (рис. 2.8), що забезпечують зручний доступ до консолі через інтерфейс методів. Клас Command представляє окрему команду з її можливими підкомандами та аргументами, передаючи її текстове представлення через з'єднання, встановлене за допомогою DeviceConnection.

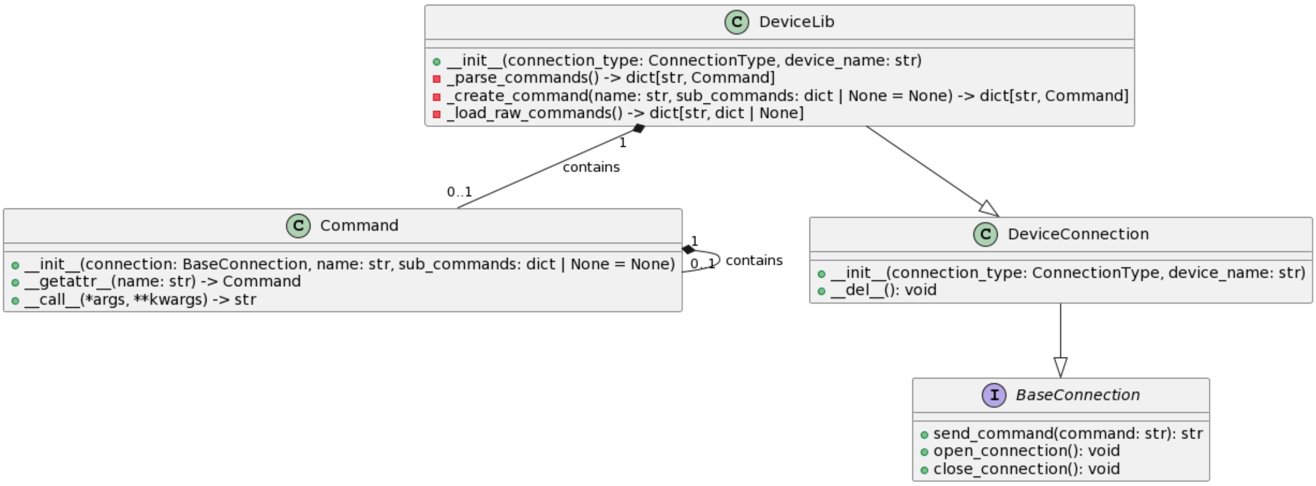


Рисунок 2.8 Діаграма класів модуля device_lib.py

Крім того, модуль використовує файл COMMANDS_YAML, який містить опис доступних команд та їх підкоманд для кожного пристрою. Цей файл дозволяє визначити їх ієрархію та надає основу для створення об'єктів класу Command та при цьому спрощує розширення функціоналу фреймворку шляхом зміни або додавання нових команд до цього файлу, не вносячи змін в основний код. Процес використання модуля передбачає створення об'єкту класу DeviceLib, який ініціалізує з'єднання з пристроєм та створює інтерфейс для взаємодії з доступними командами пристрою через атрибути-функції. Цей підхід дозволяє інженерам та розробникам використовувати більш абстрактний та зрозумілий для них інтерфейс, замість прямої взаємодії з текстовим представленням, що легко призводить до виникнення помилок при розробці.

Необхідно відзначити, що в коді можуть бути покращення та оптимізації. Наприклад, слід розглянути можливість оптимізації створення об'єктів команд та перевірити методи забезпечення безпеки підключення до пристроїв. Крім того, можна розширити функціонал для підтримки різних типів підключень та удосконалити обробку помилок.

Використання патернів розробки є важливим у програмній інженерії через їхню здатність до систематизації та стандартизації рішень для типових проблем, що виникають під час розробки програмного забезпечення. Патерни розробки є абстрактними моделями, що визначають способи вирішення конкретних задач у відповідності з найкращими практиками та принципами проєктування. Вони базуються на знаннях і досвіді попередніх розробників, узагальнюючи ефективні та перевірені методики для створення високоякісного програмного забезпечення.

В реалізації даного модулю використано такі патерни розробки як Command, Composite та Strategy.

Command Pattern - це паттерн проєктування, який дозволяє інкапсулювати запити або операції у вигляді об'єктів. Він конвертує запити на виконання певних дій у самостійні об'єкти, що дозволяє параметризувати клієнтів з різними запитами, вносити їх у чергу або реєструвати запити.

У деяких випадках програми потребують здійснювати різні дії, які можуть бути узагальнені до об'єктів команд. Однак, пряме зв'язування між викликом оператора та його виконанням може бути неефективним або складним у виконанні.

Command Pattern (рис. 2.9) дозволяє зберігати інформацію про запити як об'єкти та виконувати ці запити або передавати їх як аргументи іншим об'єктам. Він інкапсулює запит як об'єкт, що дозволяє визначати параметри запиту, такі як його виклик, об'єкт, що виконує операцію, тощо.

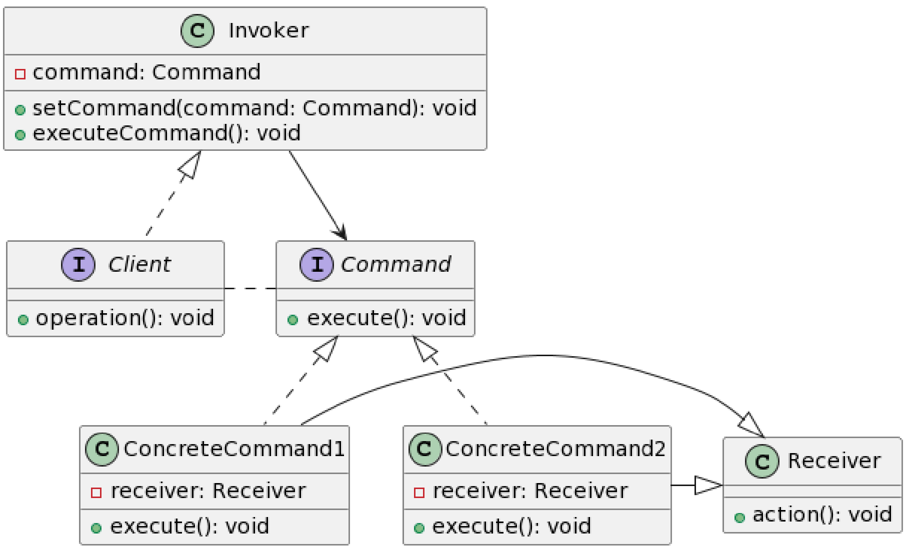


Рисунок 2.9 Command pattern

Command Pattern варто використовувати, коли потрібно параметризувати об'єкти з операціями, виконувати запити в різний час, зберігати історію запитів або створювати відмінні відкати. Не варто застосовувати цей паттерн, якщо програма не вимагає виконання різних дій або оператори не потребують інкапсуляції у вигляді об'єктів. Його структура складається з:

- Client: Клас, що ініціює запити.
- Invoker: Викликає команди для виконання.
- Command: Інтерфейс або абстрактний клас команди.
- ConcreteCommand1 та ConcreteCommand2: Конкретні реалізації команд, які виконують певні дії.
- Receiver: Виконує фактичну роботу, пов'язану з виконанням команд.

У розробленому модулі, паттерн Command використовується для інкапсуляції консольних команд девайсу у вигляді python об'єктів. Це дозволяє параметризувати та виконувати ці команди через методи класу DeviceLib, а також робить їх більш гнучкими для майбутніх змін чи розширень.

Composite Pattern - структурний паттерн проєктування, який дозволяє класам утворювати деревоподібні структури для представлення складних ієрархій об'єктів (рис. 2.10). Він дозволяє клієнтам обробляти окремі об'єкти та їх складні композиції однаковим чином. При розробці програм може виникати необхідність у представленні об'єктів у формі деревоподібних структур, де кожен об'єкт може бути оброблений індивідуально або як частина більшої структури. Проблемою є забезпечення єдинообразного доступу та операцій над цими об'єктами незалежно від їх вкладеності.

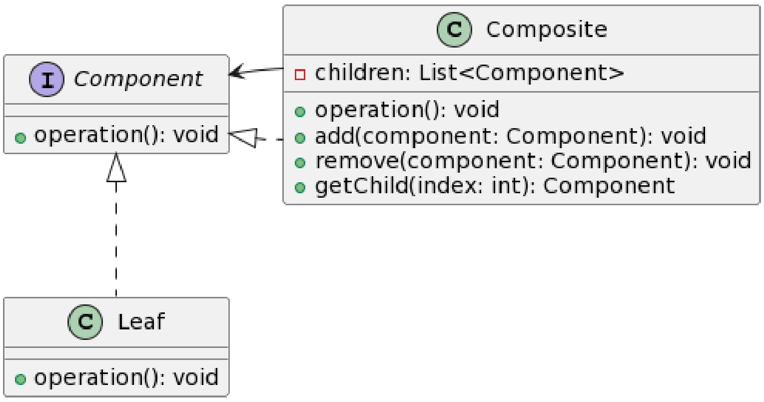


Рисунок 2.10 Composite pattern

Composite Pattern дозволяє об'єднувати об'єкти в структури дерева для того, щоб клієнти могли обробляти індивідуальні об'єкти та їх композиції у єдиний спосіб. Він визначає загальний інтерфейс для всіх компонентів у дереві, незалежно від того, є цей об'єкт простим або складним. Його варто використовувати, коли об'єкти можна об'єднати в ієрархічну структуру дерева та коли потрібно однаково обробляти як окремі об'єкти, так і групи об'єктів. Не варто застосовувати цей паттерн, якщо ієрархія об'єктів не має структури дерева або якщо немає необхідності у спільній обробці окремих та складних об'єктів однаковим чином.

Його структура складається з:

– Component: Інтерфейс або абстрактний клас, що визначає загальний інтерфейс для всіх компонентів у дереві.
– Leaf: Клас, що представляє листя у структурі, не має дочірніх компонентів.
– Composite: Клас, що представляє складну композицію об'єктів, може містити дочірні компоненти.

У розробленому модулі паттерн Composite використовується для представлення структури команд та підкоманд у вигляді дерева в об'єкті Command. Кожна команда може бути як листям (якщо вона не має підкоманд), так і складною композицією, що містить підкоманди. Це дозволяє однаково обробляти окремі команди та їх складні комбінації.

Strategy Pattern - це поведінковий паттерн проєктування, який дозволяє визначити сімейство алгоритмів і робити їх взаємозамінними (рис. 2.11). Цей паттерн дозволяє об'єктові змінювати стратегію виконання деякого завдання під час виконання програми.

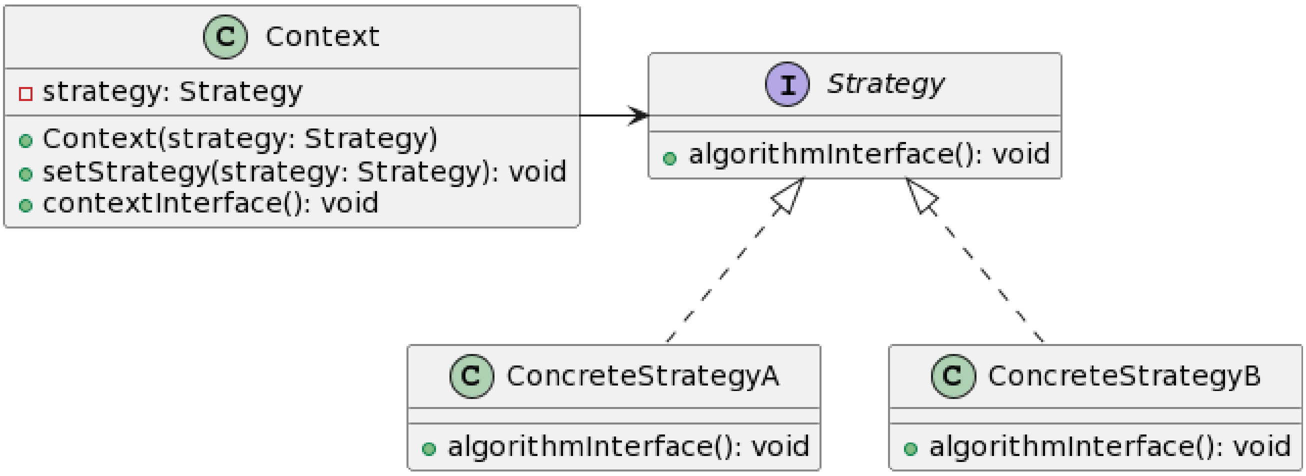


Рисунок 2.11 Strategy pattern

Під час розробки програм може виникати необхідність у виборі алгоритму або поведінки, яка повинна бути використана динамічно в залежності від вхідних даних, ситуації або умов. Проблемою є необхідність забезпечити гнучкість та розширюваність системи в цьому відношенні. Strategy Pattern пропонує виділити сімейство алгоритмів у власні класи, які реалізують спеціальний інтерфейс або абстрактний клас. Потім ці алгоритми можуть бути вибрані, замінені або комбіновані динамічно у залежності від потреби.

Strategy Pattern варто використовувати, коли існує сімейство схожих алгоритмів, між якими необхідно переключатися під час виконання програми або коли потрібно мати можливість замінити один алгоритм іншим без зміни клієнтського коду. Не варто застосовувати цей паттерн, якщо немає потреби у виборі різних алгоритмів або якщо стратегії не мають спільного інтерфейсу для заміни.

Його структура складається з:

- Strategy: Інтерфейс або абстрактний клас, що визначає загальний інтерфейс для всіх стратегій.
- ConcreteStrategyA та ConcreteStrategyB: Конкретні реалізації стратегій з

методом algorithmInterface(), які визначають специфічну реалізацію алгоритму.

- Context: Клас, що використовує паттерн Strategy, містить посилання на об'єкт Strategy та дозволяє змінювати стратегію під час виконання.

У розробленому модулі паттерн Strategy використовується для представлення різних типів підключення до пристрою у класі DeviceConnection. Кожен тип підключення (SSH або Serial) реалізує специфічний алгоритм підключення, що дозволяє змінювати стратегію підключення у залежності від потреби без впливу на основний клієнтський код.

На поточному етапі розробки було створено два головні компоненти фреймворку для автоматизованого тестування вбудованих пристроїв. Ці компоненти, Components layer та Implementation layer, становлять базову архітектурну основу системи та відповідають за виконання основних функцій. Далі у його структурі передбачені Test layer, який орієнтований на зберігання тестових сценаріїв, а також Test Data layer, де розміщуються різноманітні конфігурації та дані для виконання тестів, що зберігаються у форматах .yaml, .json, .csv. Високий рівень взаємодії з цими компонентами передбачено для автоматизаторів, які використовуватимуть фреймворк, проте на стадії його розробки залишаються певні складові, які належить додати для завершення створення системи.

2.8 Розробка рівня test data

Існує проблематика, пов'язана із зберіганням та управлінням інформацією про доступні тестові девайси та їх використанням у тестових сценаріях, яка вирішується на даному шарі. Однією з ключових задач є створення структури, яка містила б вичерпні дані про вбудовані пристрої, включаючи їх характеристики, функціональні можливості та параметри, які необхідно враховувати під час виконання тестів. Крім того, ефективне управління цими даними передбачає розроблення механізмів взаємодії з тестовими сценаріями з метою автоматизації процесу вибору, конфігурування та використання вбудованих пристроїв у різних тестових умовах. Такий підхід до організації даних дозволить оптимізувати процеси тестування та забезпечити більшу репрезентативність тестових сценаріїв у контексті вбудованих пристроїв.

2.8.1 Зберігання конфігурацій девайсів та їх парсинг

Для розв'язання даної проблеми використовується файл у форматі YAML, який має певну структуру та мету. Розроблений парсер забезпечує зчитування та обробку цих файлів, надаючи зручний інтерфейс для отримання конфігурацій.

Призначення файлу YAML з конфігураціями девайсів

1. Зберігання конфігурацій: У файлі зберігається інформація про різні пристрої та їх параметри, такі як IP-адреса, MAC-адреса, порти для SSH та послідовного з'єднання, швидкість передачі тощо.
2. Уніфікація та застосування за замовчуванням: Наявність розділу "default" дозволяє визначити значення за замовчуванням для параметрів конфігурації. Це спрощує визначення конфігурацій для нових пристроїв та уніфікує їхню структуру.

Формат файлу YAML відповідає ключам та значенням, які представляють конфігурації для різних пристроїв. Частина структури файлу:

```
default:
  ip: null
  mac: null
  ssh_port: 22
  serial_port: null
  baudrate: 115200

mikrotik_rb2011u1as:
  ip: 192.168.88.1
```

```
  mac: F44637415168
  serial_port: /dev/ttyUSB0
```

У цьому прикладі розділ "default" встановлює значення за замовчуванням для параметрів, а "mikrotik_rb2011u1as" - це конкретний пристрій з його унікальною конфігурацією, яких можна створити безліч в залежності від кількості пристроїв для тестування. Значення, відсутні у конфігурації пристрою, беруться із розділу "default".

Розроблено структуру, яка є розширюваною та в якій зберігаються дані про тестовий девайс. Наступним кроком є реалізація модуля що розумітиме як зчитувати дану структуру та надавати її дані в тестові сценарії.

2.8.2 Роль та реалізація парсера YAML файлу

Клас DevicesParser відповідає за обробку та структурування інформації, отриманої з файлу YAML (рис. 2.12). Він завантажує дані з файлу та виконує обробку значень за замовчуванням, дозволяючи забезпечити використання загальних параметрів для різних пристроїв. Це полегшує роботу з конфігураціями пристроїв, дозволяючи зберігати їхні спільні значення в одному місці та застосовувати їх усюди, де це потрібно.

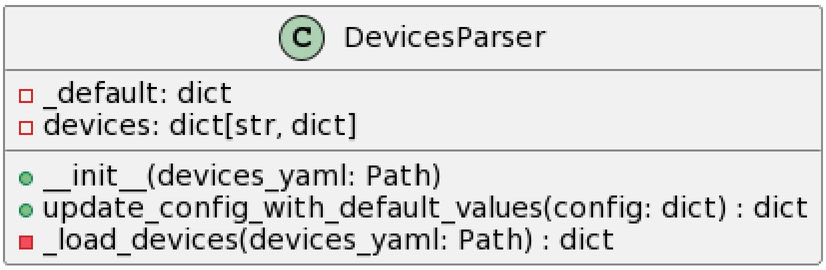


Рисунок 2.12 Клас парсера

Розроблений парсер зчитує вміст YAML файлу та обробляє його для подальшого використання в системі, а також забезпечує застосування параметрів "default" до конфігурацій конкретних пристроїв, які не мають визначених певних параметрів.

Перевагами використання YAML формату та окремого парсера є:

- Легкість зберігання та редагування: Формат YAML дозволяє структуровано зберігати дані у вигляді ключів та значень, що робить файл зрозумілим та зручним для редагування.
- Можливості розширення: Уніфікований формат дозволяє легко додавати нові пристрої та їхні конфігурації, використовуючи значення за замовчуванням.
- Гнучкість: Використання парсера дозволяє системі фреймворку легко читати та використовувати дані конфігурацій, забезпечуючи гнучкість у роботі з різними пристроями.

2.9 Розробка рівня test

2.9.1 Реалізація conftest.py

На даному шарі при розробці фреймворку потрібно створити root conftest.py файл. Даний файл є важливим елементом структури pytest, який відіграє значну роль у забезпеченні параметризації, конфігурації та спільного використання фікстур та налаштувань між тестовими модулями.

Основна функціональність conftest.py полягає у визначенні фікстур - це ресурси, які можуть бути використані в тестах для забезпечення початкових умов, даних або інших необхідних об'єктів для виконання тестових сценаріїв. Фікстури, визначені у даному файлі, можуть бути автоматично викликані у тестах, які належать до тієї ж ієрархії каталогів, що й conftest. Крім того, conftest.py дозволяє встановлювати глобальні налаштування тестів, такі як обробка командної стрічки, підключення додаткових плагінів, налаштування середовища тощо. Це спрощує управління тестовими параметрами та робить процес тестування більш гнучким та легким у супроводженні.

Загальна функціональність conftest.py допомагає забезпечити однорідність налаштувань тестів у межах проєкту, сприяючи зручності та чистоті коду. Це допомагає уникнути дублювання коду та забезпечити консистентність у виконанні тестів.

В даному фреймворку conftest.py містить низку hooks (застосування функцій у певні моменти життєвого циклу pytest), які регулюють та модифікують поведінку тестів під час їх виконання. У цьому контексті, hooks у pytest використовуються для забезпечення конфігурації, маніпуляції параметрами запуску, та впорядкування виконання тестів.

pytest_addoption(parser) - функція що додає опції командного рядка для встановлення параметрів, які можуть використовуватися під час виконання тестів. В даному випадку, опції --connection та --device приймають значення для визначення типу з'єднання та імені пристрою відповідно.

pytest_collection_modifyitems(session, config, items) - визначає, які тести слід пропустити або виконати, залежно від заданих у командному рядку параметрів, наприклад, типу з'єднання. Вона перевіряє маркери кожного тесту та вирішує, чи потрібно пропустити тест на основі зазначеного типу з'єднання.

Також розроблено fixtures (фікстури), що використовуватимуться у всіх тестових сценаріях:

- device_name: Повертає ім'я пристрою, вказане через параметр командного рядка.
- connection_type: Визначає тип з'єднання з пристроєм на основі значення, введеного через параметр командного рядка --connection. Вона перевіряє введене значення та встановлює відповідний тип з'єднання (SSH або SERIAL).
- device_lib: Ця фікстура створює екземпляр класу DeviceLib з визначеними параметрами з'єднання та імені пристрою, а після виконання тестів закриває підключення.

Кожна з цих функцій та фікстур відповідає за конкретний аспект управління процесом тестування, надаючи можливість налаштування параметрів, фільтрації тестів та створення необхідних об'єктів для їх виконання. Це дозволяє покращити управління тестовими середовищами та забезпечити виконання тестів для різних типів з'єднань з вбудованими пристроями.

2.9.2 Розробка тесту

Як приклад роботи фреймворку розроблено тестовий сценарій smoke_test.py. Файл smoke_test.py містить функціонал, спрямований на проведення тестування з'єднання з вбудованими пристроями шляхом використання фікстур та параметризації тестів. Тест test_connection: функціонально перевіряє можливість підключення до вбудованого пристрою через різні типи з'єднань (SSH або SERIAL), використовуючи параметризацію тестів. Він перевіряє, чи вдалося підключитися до пристрою вказаним типом з'єднання, і далі виконує метод "beep", що подає звуковий сигнал.

Тест test_connection використовує параметризацію, щоб запустити його для кожного типу з'єднання окремо. Це досягається за допомогою mark.parametrize, де для кожного типу з'єднання (SSH або SERIAL) створюється окремий тест з відповідним маркером mark.ssh_only або mark.serial_only. Параметризація тестів дозволяє одному тестовому випробуванню запускатися з різними наборами вхідних даних або налаштувань.

3 МЕТОДИКА ВИКОРИСТАННЯ СИСТЕМИ

Однією з невід'ємних складових, яка визначає структуру та функціональність фреймворку для автотестування, є його документація. Ця документація, у свою чергу, складається з описаних алгоритмів роботи з фреймворком, набору рекомендаційних правил для інженерної команди, які працюють з цим інструментарієм, а також послідовних інструкцій, спрямованих на його коректне налаштування та уніфікацію середовища.

Цей аспект фреймворку є не менш важливим ніж кодова база, оскільки він забезпечує однаковий початковий набір знань і розуміння принципів роботи фреймворку серед усієї інженерної команди. Це важливо для того, щоб уникнути непорозумінь та розбіжностей, особливо у випадках виникнення проблемних ситуацій. Однакове розуміння принципів функціонування фреймворку допомагає зосередити увагу інженерів на ефективному тестуванні вбудованих пристроїв, уникаючи зайвої витрати часу на розбирання та вирішення проблем, пов'язаних саме з інструментарієм тестування.

3.1 Інструкція з установки та налаштування

3.1.1 Загальний опис функціоналу

У цій інструкції розглядається фреймворк на базі "pytest", що є основою даного репозиторію. "pytest" представляє собою програмний інструмент для автоматизованого тестування програмного забезпечення загалом. Даний фреймворк розроблений на його основі та включає в себе ряд функціональних можливостей для спрощення тестування інтеграції з тестовими пристроями.

Система має чітко визначену структуру файлів та надає основні можливості для взаємодії з тестовими пристроями.

```
.
├── config
│   ├── device_secrets.py
│   ├── __init__.py
│   ├── loggers.py
│   └── paths.py
├── pytest.ini
├── README.md
├── resources
│   ├── console_commands.yaml
│   ├── devices.yaml
│   ├── lib
│   └── testdata
├── src
│   ├── connectors
│   │   ├── base_connection.py
│   │   ├── exceptions.py
│   │   ├── __init__.py
│   │   ├── serial_connection.py
│   │   └── ssh_connection.py
│   ├── device_lib
│   │   ├── device_lib.py
│   │   └── exceptions.py
│   └── devices
│       ├── device_parser.py
│       └── __init__.py
└── tests
    ├── conftest.py
    └── framework_checks
        └── smoke_test.py
```

Основна функціональність фреймворку включає в себе:

– Можливість встановлення зв'язку з тестовим пристроєм через SSH або Serial з'єднання. Це дозволяє виконувати автоматизовані тести на пристроях із збереженням конфігурацій та налаштувань. Також за потреби можна розширити типи з’єднання якщо потрібно, адже використана архітектура дозволяє це зробити.
– Фреймворк дозволяє зберігати конфігурацію безлічі тестових пристроїв. Це спрощує управління та доступ до необхідних налаштувань для виконання автоматизованих тестів.
– Надає інтуїтивно зрозумілі абстракції для взаємодії з консольними командами пристроїв, що дозволяє легко і ефективно виконувати тести, використовуючи команди без необхідності глибокого розуміння їх внутрішньої реалізації.
– Дозволяє легко створювати тести, фокусуючись на їхньому функціоналі та перевірці, не потребуючи в подробицях з'єднання та взаємодії з тестовими

пристроями. Це спрощує процес створення тестів та робить їх більш зрозумілими та підтримуваними.

– Підтримує можливість автоматичної генерації звітів у форматі Allure, що надає зручний спосіб візуалізації результатів тестування та дозволяє швидко отримувати та аналізувати дані щодо виконання тестів для зручної інтерпретації та прийняття рішень.

Загальною метою даного фреймворку є спрощення процесу автоматизованого тестування та забезпечення зручного і надійного інтерфейсу для взаємодії з тестовими пристроями.

3.1.2 Клонування репозиторію за допомогою SSH

Для клонування даного репозиторію спершу необхідно згенерувати SSH ключ та додати його до приватного облікового запису GitHub. Нижче наведено послідовні кроки:

Відкрити термінал на вашій локальній машині та ввести команду

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

де your_email@example.com - електронна адреса, пов'язана з приватним обліковим записом GitHub. Під час генерації ключа буде запропоновано вказати розташування для збереження ключа та пароль для захисту ключа.

Далі необхідно запустити агент ssh у фоновому режимі.

```
eval "$(ssh-agent -s)"
```

В залежності від оточення, може знадобитися використовувати іншу команду. Наприклад, користуватися root-доступом, запустивши sudo -s -H перед запуском ssh-agent, або використовувати exec ssh-agent bash чи exec ssh-agent zsh для запуску ssh-agent.

Наступним кроком є додавання свого приватного ключа SSH до ssh-agent. Якщо ключ було створено з іншою назвою, замість id_ed25519 у команді потрібно замінити назву файлу приватного ключа.

```
ssh-add ~/.ssh/id_ed25519
```

Ці команди допоможуть створити агента ssh у фоновому режимі та додати вказаний приватний ключ до агента ssh для подальшого використання під час авторизації. Створений ключ необхідно додати до облікового запису GitHub. Для цього увійти в приватний обліковий запис GitHub. Натиснути на аватар у верхньому правому куті сторінки, оберати "Settings", а потім у лівому боці "SSH and GPG keys" та "New SSH key" або "Add SSH key" (рис. 3.1). Далі необхідно скопіювати зміст файлу ~/.ssh/id_rsa.pub на локальній машині (це публічний SSH ключ) та вставити його до відповідного поля на GitHub.

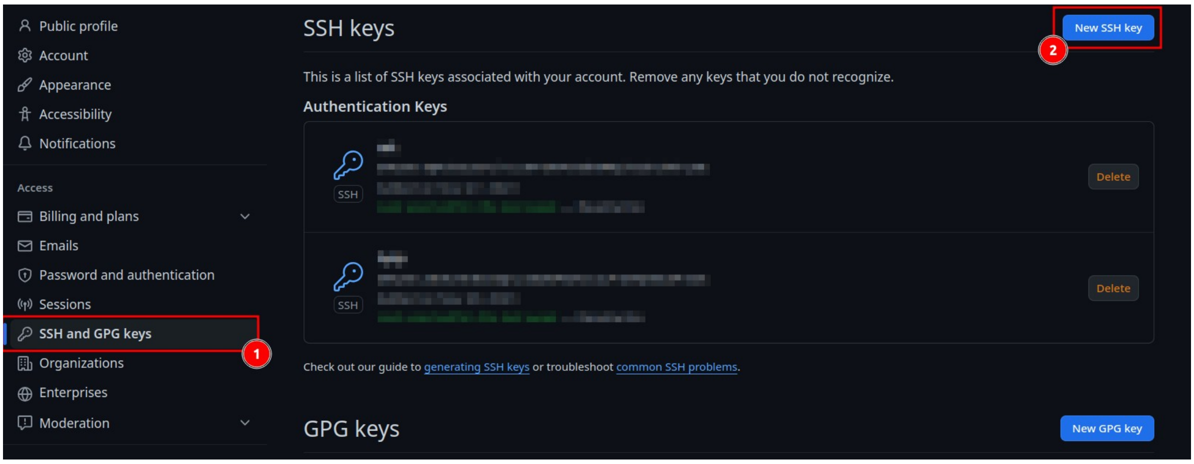


Рисунок 3.1 Додавання публічного SSH ключа

Останнім кроком є клонування репозиторію на локальну машину. Використовуючи командний рядок, потрібно перейи до каталогу, де буде зберігатись клонована версію репозиторію та виконати команду

```
git clone git@github.com:201dreamers/automation-framework.git,
```

Ці кроки забезпечують успішне створення SSH ключа, його додавання до облікового запису GitHub, а також клонування репозиторію на вашу локальну машину, забезпечуючи безпечний та захищений канал зв'язку між вашими даними та сервером GitHub.

3.1.3 Зберігання секретних даних

У даному проєкті секретні дані, які не повинні бути опубліковані в репозиторії, зберігаються у вигляді змінних оточення. Цей підхід передбачає, що кожен інженер, який працює з проєктом, самостійно встановлює ці змінні на своєму робочому середовищі. Наприклад, такі секретні дані, як ім'я користувача пристрою (`DEVICE_USERNAME`) та пароль пристрою (`DEVICE_PASSWORD`), повинні бути встановлені вручну.

Для встановлення змінних оточення, можна скористатися командним рядком вашої операційної системи:

```
export DEVICE_USERNAME=your_username
export DEVICE_PASSWORD=your_password
```

Цей підхід є безпечним та корисним з декількох причин:

- Зберігання секретних даних у вигляді змінних оточення дозволяє уникнути їхньої публікації у відкритому репозиторії, зменшуючи ризик несанкціонованого доступу до цих даних.
- Кожен інженер може налаштувати своє робоче середовище, встановивши власні значення для змінних оточення, що дає можливість індивідуального налаштування без необхідності змінювати загальний код проєкту.
- Цей підхід спрощує управління конфіденційними даними, оскільки не вимагає постійних змін у коді або в репозиторії для зміни чутливих інформаційних даних.

Такий підхід зберігання секретних даних у вигляді змінних оточення є важливим з погляду безпеки та робочої ефективності, оскільки дозволяє уникнути ненавмисного розголошення конфіденційної інформації та спрощує налаштування робочого середовища кожного інженера.

3.1.4 Створення віртуального оточення та установка залежностей

У даному проєкті використовується версія Python 3.11. Для забезпечення ізоляції середовища та уникнення конфліктів залежностей рекомендується створювати віртуальне оточення venv.

```
python3 -m venv venv
```

Для активації віртуального оточення використовуються команди залежно від операційної системи:

На Windows:

```
venv\Scripts\activate
```

На Unix або MacOS:

```
source venv/bin/activate
```

Після активації віртуального оточення, для установки всіх необхідних залежностей з використанням pip виконується команда, де файл requirements.txt містить всі необхідні залежності проєкту

```
pip install -r requirements.txt
```

Бібліотеки та модулі що є залежностями даного проєкту:

- pyyaml: Бібліотека для роботи з YAML-файлами, що дозволяє зчитувати та записувати дані у форматі YAML.
- paramiko: Клієнт SSH для Python, який надає можливість взаємодії з пристроями за допомогою протоколу SSH.
- ipython: Потужна інтерактивна оболонка для Python, що надає розширені можливості для розробки та експериментів.
- pyserial: Бібліотека для роботи з послідовним портом (Serial port) для Python, що дозволяє взаємодіяти з пристроями через порти RS-232/RS-485.
- pytest-instafail: Плагін для pytest, який надає миттєве відображення результатів тестів при їх виконанні.
- pytest-timeout: Плагін для pytest, який дозволяє встановлювати таймаути для тестів.

- allure-pytest: Плагін для збору результатів тестів та генерації звітів у форматі Allure.
- pytest-check: Плагін для pytest, який надає можливість здійснювати різноманітні перевірки під час виконання тестів.
- loguru: Проста та елегантна бібліотека для логування у Python, яка дозволяє зручно організовувати логи в програмах.
- python-lsp-server: Сервер LSP (Language Server Protocol) для підтримки роботи редакторів та інструментів, які підтримують LSP, для роботи з Python.
- ruff: лінтер, що перевіряє файли з кодом на наявність помилок різного типу та на відповідність загальноприйнятим на проєкті стандартам.

Встановлення цих бібліотек у віртуальне оточення дозволяє забезпечити необхідність для розробки, тестування та оптимізації функціоналу проєкту з дотриманням версійної сумісності та безпеки.

3.2 Запуск тестів та опції для pytest

У фреймворку pytest існує ряд різноманітних опцій, які надають можливості для керування та налаштування процесу виконання тестів. Нижче перераховано ті, які найбільш часто використовуються в повсякденних запусках:

-m: Опція для вибору тестів за міткою (маркуванням). Дозволяє виконувати тільки ті тести, які мають вказану мітку. Наприклад: -m smoke виконає тільки тести, які мають мітку "smoke".

-k: Фільтрує тести за вказаним рядком у їхніх іменах. Наприклад: -k test_login виконає тільки тести, ім'я яких містить "test_login".

–connection, –device: Опції для вказання параметрів підключення до пристрою або його ідентифікатора, які можуть бути використані в тестах, які потребують специфічного пристрою для виконання.

–alluredir, –clean-alluredir: Опції для вказання каталогу, де будуть збережені файли звітів Allure, а також для очищення цього каталогу перед виконанням тестів.

-x, –maxfail: Опції для припинення виконання тестів після перевищення певної кількості невдало виконаних тестів (-x зупинить виконання після першої помилки, а -maxfail=N зупинить після N невдалих тестів).

–pdb, –pdbcls: Опції для включення відлагоджувальника Python (pdb) у випадку помилки тесту (--pdb встановлює відлагоджувальник за замовчуванням, а --pdbcls дозволяє вибір альтернативного відлагоджувальника).

Вони дозволяють інженерам використовувати pytest з більшою гнучкістю та контролем під час виконання тестів. Кожна опція має своє призначення та дозволяє налаштовувати виконання тестів відповідно до потреб тестування, що сприяє покращенню ефективності тестового процесу та швидкості виявлення помилок.

3.3 Збереження, перегляд та розповсюдження звітів pytest-allure

Pytest-allure генерує звіти тестування у форматі Allure, які містять інформацію про виконані тести, їх статуси, логи, скріншоти та інші деталі, необхідні для аналізу результатів тестування.

Звіти тестування pytest-allure зазвичай зберігаються у вказаному каталозі, який можна встановити за допомогою параметру `--alluredir` під час запуску pytest.

```
pytest --alluredir=allure-results
```

Тут `allure-results` - це каталог, де будуть збережені файли звітів pytest-allure.

В даному проєкті всі звіти повинні зберігатись в папці reports.

Для перегляду звітів pytest-allure вам знадобиться Allure Command Line Interface (CLI). Ви можете встановити його, скачавши потрібний файл з офіційного сайту.

Після встановлення Allure CLI можна відобразити звіти тестування за допомогою команди:

```
allure serve path/to/allure-results
```

Ця команда запустить локальний веб-сервер і відкриє звіти у веб-браузері за адресою `http://localhost:port`, де `port` - це порт, який використовує Allure для відображення звітів.

Для архівування звітів та їхнього розповсюдження з іншими інженерами вам потрібно створити архів з каталогу, де зберігаються файли звітів pytest-allure:

```
tar -czvf allure-results.tar.gz allure-results
```

Ця команда створить архів `allure-results.tar.gz` з каталогу `allure-results`. Далі архів можна надіслати іншим інженерам для подальшого аналізу результатів тестування. Інші інженери можуть розархівувати звіти за допомогою команди:

```
tar -xzvf allure-results.tar.gz
```

Такий процес дозволяє легко архівувати та обмінюватись звітами тестування pytest-allure між членами команди для спільного аналізу результатів тестів та виявлення проблем.

3.4 Правила форматування коду

3.4.1 PEP8

Дотримання стандартів PEP 8 (стиль коду для Python) є обов'язковим. PEP 8 - це офіційний документ, що містить рекомендації щодо стилю коду для мови програмування Python. Він визначає правила форматування, розміщення пробілів, назви змінних, коментарів та інші конвенції, які допомагають створювати читабельний та однорідний код у проєктах.

- Використовуйте 4 пробіли для відступів (не табуляції).
- Відступи в коді повинні бути консистентними в усьому проєкті.

- Рекомендована максимальна довжина рядка для даного проєкту - 120 символів.
- Максимальна довжина для коментарів та докстрінгів - 80 символів.
- Уникайте зайвого порожнього простору на початку та в кінці рядків або після коми, крапки з комою і т.д.
- Робіть окремі імпорти для кожного модуля.
- Уникайте використання import *.
- Використовуйте розумні та описові назви змінних, функцій, методів та класів.
- Дотримуйтеся конвенцій щодо використання зміїного стилю (snake_case) для функцій та змінних, та PascalCase для класів.
- Використовуйте зрозумілі коментарі для пояснення складних частин коду.
- Додавайте докстрінги для модулів, класів та функцій.
- Розділяйте функції та класи порожніми рядками для полегшення читання.
- Уникайте загальних операцій типу except Exception:. Обирайте специфічні типи винятків.
- Вирівнюйте вирази для полегшення читання, але не перевантажуйте вирівнюванням.
- Використовуйте дужки для розбиття довгих логічних ліній коду.
- Читабельність та документація:
- Код повинен бути зрозумілим для інших учасників команди. Надавайте перевагу зрозумілості коду над скороченням.
- Додавайте коментарі та документацію до функцій, особливо якщо їх призначення неочевидне.

3.4.2 pytest

- Створюйте чистий та добре організований код для тестів.
- Використовуйте pytest-check плагін для перевірки очікувань результатів

тестів, якщо в тесті є більше двох перевірок.
- Надавайте зрозумілі назви тестам, що вони перевіряють. Назва кожного тестового комплекту повинна починатись або закінчуватись на test_ або _test відповідно.
- Кожен тестовий сценарій повинен починатись зі слова test_.

3.4.3 Функції та модулі

- Функції повинні виконувати одну конкретну задачу. Дотримуйтесь принципу однієї відповідальності (Single Responsibility Principle).
- Модулі мають бути логічно організованими та містити пов'язаний функціонал.
- Використовуйте винятки (exceptions) для обробки помилок замість відновлення (recovery) з використанням try-except.
- Загальні практики:
- Використовуйте модульність та перевикористання коду. Уникайте дублювання коду.
- Використовуйте гіт для контролю версій. Код має бути збереженим та документованим у системі контролю версій.

3.4.4 Тестування

- Переконайтеся, що всі тести завершились успішно перед тим, як залити їх до основної гілки.
- Впевніться, що всі тести працюють та покривають необхідні сценарії.
- Спільні конвенції команди:
- Визначте спільні конвенції для кодування в команді та дотримуйтесь їх.
- Проводьте регулярні код-рев'ю та обговорення для покращення якості коду.

Ці рекомендації допоможуть забезпечити чистий, організований та легко зрозумілий код для автоматизаторів проєкту, що використовують Python та pytest.

3.5 Правила роботи з репозиторієм

Назва гілки має завжди розпочинатись з номера задачі. Це дозволяє відстежувати зміни системі трекінгу. Щоб створити гілку, у кореневій папці репозиторію виконуються наступні команди.

```
git checkout -b <TICKET-XXX_NEW_BRANCH_NAME>
```

NEW_BRANCH_NAME може бути коротким описом тікета, щоб полегшити пошук гілки. Дозволені символи для назви гілки: підкреслення, дефіси, косі риски та всі буквено-цифрові символи. Приклад: PROJ-123_fix_connection_issue

Перед здійсненням коміту змін, спочатку необхідно підготувати файли. Щоб переглянути, які зміни можна підготувати до коміту, використовується git status. Список файлів буде перераховано під заголовком Changes not staged for commit.

Для додавання файлів до коміту можна використовувати:

```
git add <FILENAME>
git add -u
```

Після підготовки файлів для коміту, виконується команда, де в повідомленні також вказується номер задачі, що дозволить відстежувати коміти, що пов'язані з тікетом.

```
git commit -m "TICKET-XXX <ПОВІДОМЛЕННЯ ПРО КОМІТ>"
```

Щоб відправити зміни до віддаленого репозиторію та встановити відстеження, на вашій локальній гілці виконується:

```
git push --set-upstream origin HEAD
```

Це відправить вашу локальну гілку на віддалений репозиторій та встановить відстеження за нею. З відстеженням, коли виконується git status, можна побачити стан локальної гілки порівняно з віддаленою гілкою. Наприклад, кількість комітів локально, які ще не були відправлені.

Після того як гілка встановлена для відстеження віддаленої, для всіх майбутніх випадків використовується git push

3.6 Створення запиту на злиття

Для створення PR на GitHub потрібно перейти на вкладку Pull requests та у верхньому правому куті натиснути на New pull request. Потім клацнути на базову гілку та виберати цільову (зазвичай master або main). З'явиться відмінність, де відображаються змінені файли та коміти, тоді необхідно натиснути на Create pull request та додати відповідних рецензентів до запиту на злиття, обравши значок шестерні в правій бічній панелі

В описі цього запиту обов'язково потрібно заповнити наступний шаблон який дозволить всім рецензентам коротко ознайомитись з метою даного полреквесту та змінами присутніми в ньому. Це зменшить час, витрачений на перевірку та допоможе зекономити ресурси команди.

```
<!-- Update ticket number, hyperlink will automatically form -->
PROJ-XXX

**KPI Metrics**
| Metric                  | Count |
|------------------------|-------|
| Test Cases Modified     | 0     |
| Test Cases Added        | 6     |
| Library Improvements    | 0     |
| Framework Improvements  | 1     |

<!-- Please provide of a one line summary of each library improvement  -->
**Library Improvements**
* N/A
```

````
<!-- Please provide of a one line summary of each framework improvement -->
**Framework Improvements**
* N/A

<!-- Please provide of a one line summary of each test case type added -->
**Test Case Summary**
* N/A

---
**Description**
<!-- Please describe the changes in this PR -->

---
**Run Command**
```bash
pytest ...
```
---
**Screenshot of Passing Result**
````

Основні правила:

– Дотримуватися шаблону запиту на злиття
– Додавати номер задачі як у заголовку PR, так і у повідомленні коміту файлу, щоб зв'язати трекінгову систему та Git
– Додавати номер задачі в Testrail, щоб зв'язати трекінгову систему і Testrail
– Не додавати в репозиторій ніякі секретні дані чи паролі.
– Форматувати файли *.py згідно правил описаних в документі
– Записувати URL тест-кейсу Testrail у кожному автоматизованому тестовому сценарії
– Переконатися, що всі тести пройшли запуск успішно.

ВИСНОВКИ

Представлена робота відображає процес розробки комплексу для тестування вбудованих систем, включаючи теоретичну базу та практичну реалізацію фреймворку. Основною метою цієї роботи було створення інструменту, що спрямований на вирішення специфічних проблем тестування вбудованих систем, у контрасті з загальними фреймворками, які часто не враховують усіх їх особливостей.

Значний акцент у роботі було зроблено на виборі та визначенні вимог до вбудованого пристрою для тестування, оскільки його універсальність та спроможність працювати з іншими системами мали ключове значення для успішної інтеграції. Встановлені вимоги до системи визначили основні параметри та межі проєкту, які є критичними для успішності розробки.

Розглянуті різноманітні інструменти та мови програмування, призвели до вибору Python та Pytest як основних інструментів для створення фреймворку. Цей вибір був обґрунтований їхньою гнучкістю та можливістю легкої інтеграції з розроблюваною системою.

Другий розділ роботи присвячений детальному аналізу існуючих архітектур фреймворків, оцінці їхніх переваг та недоліків з метою визначення оптимальної моделі для подальшої розробки. Цей процес аналізу включав розгляд структури, основних компонентів, принципів та парадигм, які лежать в основі кожної архітектури.

В ході дослідження було визначено, що для розробки фреймворку для тестування вбудованих систем оптимальною архітектурою є модульна структура, розділена на чіткі компоненти. Ця модель дозволяє ефективно розширювати та модифікувати функціонал, а також легко інтегрувати нові компоненти для різних типів тестів.

Визначивши відповідну архітектуру, було розроблено структуру фреймворку, розбиту на чотири основних шари:

– Components: Включає модулі для з'єднання з вбудованим пристроєм через протоколи SSH та Serial, що забезпечують доступ до пристрою для виконання тестів.

– Implementation: Абстракції для роботи з консольними командами вбудованого пристрою, надаючи можливість виклику та управління його функціоналом з фреймворку.
– Test: Основна логіка тестів, яка використовує інші шари для здійснення тестування функціоналу вбудованого пристрою.
– Test Data: Дані, необхідні для виконання тестів, такі як вхідні параметри, очікувані результати тощо.

Аналіз та вибір оптимальної архітектури фреймворку став важливим кроком у процесі розробки системи, а кожен з цих шарів був докладно розглянутий, описаний та реалізований з урахуванням специфіки вбудованих систем та їхнього тестування. Такий розподіл функціоналу дозволяє забезпечити високий рівень модульності та гнучкості фреймворку, спрощуючи його розвиток та підтримку у майбутньому.

У останньому розділі роботи розглядається процес розробки документації, спрямованої на інструктування інженерної команди щодо коректного використання розробленого фреймворку для тестування вбудованих систем. Ця документація стала ключовою для забезпечення правильного використання інструментарію та максимальної ефективності процесу тестування. Вона містить алгоритми роботи з фреймворком, детальні рекомендації для інженерної команди, що використовує цей інструментарій, та послідовні інструкції для коректного налаштування та використання. Вона є важливим ресурсом, що допомагає не лише зрозуміти принципи роботи з фреймворком, але й забезпечує єдність підходів у команді, сприяючи збільшенню продуктивності та якості розробки в цілому.

У цілому, розроблений фреймворк та його документація відображають хороший рівень систематизації та організації, що сприятиме ефективному тестуванню вбудованих систем та спростить процес роботи з цим інструментом.

ПЕРЕЛІК ПОСИЛАНЬ

1. BCC Research. Embedded Systems: Technologies and Markets / BCC Research // BCC Research (2020). – 2020.
2. Dustin E. Automated software testing: introduction, management, and performance / Elfriede Dustin, Jeff Rashka, John Paul. – [Б. м.] : Addison-Wesley Professional, 1999. – 608 с.
3. Kit E. Software testing in the real world: improving the process / Edward Kit. – New York, N.Y : ACM Press, 1995. – 252 с.
4. Prokopets E. QA automation: how to design a proven framework [Електронний ресурс] / Elena Prokopets // Evantis. – Режим доступу: https://www.edvantis.com/blog/qa-automation-framework/#htoc-sample-test-automation-framework (дата звернення: 02.11.2023).
5. Платунов А. Вбудовані системи керування / Антон Платунов. – [Б. м.] : Control Engineering, 2013.
6. Barrett S. F. Embedded systems design and applications with the 68HC12 and HCS12 / Steven F. Barrett. – Upper Saddle River, NJ : Pearson/Prentice Hall, 2004.
7. Биковський С. В. Проектування з'єднаних вбудованих систем (Hardware/Software Co-Design) / С. В. Биковський. – [Б. м.] : Ун-т ІТМО, 2016. – 105 с.
8. Teich J. Hardware/Software codesign: the past, the present, and predicting the future [Електронний ресурс] / Jürgen Teich // Proceedings of the IEEE. – 2012. – Т. 100, Special Centennial Issue. – С. 1411–1430. – Режим доступу: https://doi.org/10.1109/jproc.2011.2182009 (дата звернення: 15.11.2023).
9. Ganssle J. 2013 embedded systems market study [Електронний ресурс] / Jack Ganssle // Embedded.com. – Режим доступу: https://www.embedded.com/2013-embedded-systems-market-study/ (дата звернення: 05.11.2023).
10. Wolf W. H. Hardware-software co-design of embedded systems [Електронний ресурс] / W. H. Wolf // Proceedings of the IEEE. – 1994. – Т. 82, № 7. – С. 967–

989. – Режим доступу: https://doi.org/10.1109/5.293155 (дата звернення: 27.11.2023).

11. Model transformation method of27.11.2023 embedded system hardware for improving design process performance [Електронний ресурс] / Arif Sasongko [та ін.] // International journal on electrical engineering and informatics. – 2011. – Т. 3, № 2. – С. 217–233. – Режим доступу: https://doi.org/10.15676/ijeei.2011.3.2.7 (дата звернення: 30.10.2023).
12. Parkhomenko A. Complex requirements analysis for the high-level design of Embedded Systems / A. Parkhomenko, O. Gladkova // Вісник НУ "Львівська політехніка" "Комп'ютерні системи проектування. Теорія і практика". – № 808.
13. Kit E. Software testing in the real world: improving the process / Edward Kit. – New York, N.Y : ACM Press, 1995. – 252 c.
14. Automotive embedded systems handbook [Електронний ресурс] / ред.: N. Navet, F. Simonot-Lion. – [Б. м.] : CRC Press, 2017. – Режим доступу: https://doi.org/10.1201/9780849380273 (дата звернення: 06.11.2023).
15. Badgett T. Art of software testing / Tom Badgett, Glenford J. Myers, Corey Sandler. – [Б. м.] : Wiley & Sons, Incorporated, John, 2011.
16. Broekman B. Testing embedded software / Bart Broekman, Edwin Notenboom. – [Б. м.] : Addison-Wesley Professional, 2002. – 368 c.
17. An orchestrated survey of methodologies for automated software test case generation [Електронний ресурс] / Saswat Anand [та ін.] // Journal of systems and software. – 2013. – Т. 86, № 8. – С. 1978–2001. – Режим доступу: https://doi.org/10.1016/j.jss.2013.02.061 (дата звернення: 06.11.2023).
18. Zhu H. Software unit test coverage and adequacy [Електронний ресурс] / Hong Zhu, Patrick A. V. Hall, John H. R. May // ACM computing surveys. – 1997. – Т. 29, № 4. – С. 366–427. – Режим доступу: https://doi.org/10.1145/267580.267590 (дата звернення: 12.11.2023).
19. Bertolino A. Software testing research: achievements, challenges, dreams

[Електронний ресурс] / Antonia Bertolino // Future of software engineering, Minneapolis, MN, USA, 23–25 трав. 2007 р. – [Б. м.], 2007. – Режим доступу: https://doi.org/10.1109/fose.2007.25 (дата звернення: 12.11.2023).

20. Pezzè M. Software testing and analysis: process, principles, and techniques / Mauro Pezzè. – [Hoboken, N.J.] : Wiley, 2008. – 488 c.

21. Gaudel M.-C. Testing can be formal, too [Електронний ресурс] / Marie-Claude Gaudel // TAPSOFT '95: theory and practice of software development. – Berlin, Heidelberg, 1995. – С. 82–96. – Режим доступу: https://doi.org/10.1007/3-540-59293-8_188 (дата звернення: 13.11.2023).

22. Offutt J. Generating tests from UML specifications [Електронний ресурс] / Jeff Offutt, Aynur Abdurazik // Lecture notes in computer science. – Berlin, Heidelberg, 1999. – С. 416–429. – Режим доступу: https://doi.org/10.1007/3-540-46852-8_30 (дата звернення: 13.11.2023).

23. Projected state machine coverage for software testing [Електронний ресурс] / G. Friedman [та ін.] // ACM SIGSOFT software engineering notes. – 2002. – Т. 27, № 4. – С. 134–143. – Режим доступу: https://doi.org/10.1145/566171.566192 (дата звернення: 04.11.2023).

24. Offutt J. Generating tests from UML specifications [Електронний ресурс] / Jeff Offutt, Aynur Abdurazik // Lecture notes in computer science. – Berlin, Heidelberg, 1999. – С. 416–429. – Режим доступу: https://doi.org/10.1007/3-540-46852-8_30 (дата звернення: 12.11.2023).

25. Projected state machine coverage for software testing [Електронний ресурс] / G. Friedman [та ін.] // ACM SIGSOFT software engineering notes. – 2002. – Т. 27, № 4. – С. 134–143. – Режим доступу: https://doi.org/10.1145/566171.566192 (дата звернення: 23.11.2023).

26. Tretmans J. Model based testing with labelled transition systems [Електронний ресурс] / Jan Tretmans // Formal methods and testing. – Berlin, Heidelberg, 2008. – С. 1–38. – Режим доступу: https://doi.org/10.1007/978-3-540-78917-8_1 (дата звернення: 22.11.2023).

27. de Alfaro L. Interface automata [Електронний ресурс] / Luca de Alfaro, Thomas A. Henzinger // ACM SIGSOFT software engineering notes. – 2001. – Т. 26, № 5. – С. 109–120. – Режим доступу: https://doi.org/10.1145/503271.503226 (дата звернення: 23.11.2023).
28. Groz R. Relating conformance test coverage to formal specifications [Електронний ресурс] / Roland Groz, Olivier Charles, Josiane Renévot // Formal description techniques IX. – Boston, MA, 1996. – С. 195–210. – Режим доступу: https://doi.org/10.1007/978-0-387-35079-0_12 (дата звернення: 30.11.2023).
29. Test selection, trace distance and heuristics [Електронний ресурс] / L. M. G. Feijs [та ін.] // IFIP advances in information and communication technology. – Boston, MA, 2002. – С. 267–282. – Режим доступу: https://doi.org/10.1007/978-0-387-35497-2_20 (дата звернення: 29.11.2023).
30. Jard C. TGV: theory, principles and algorithms [Електронний ресурс] / Claude Jard, Thierry Jéron // International journal on software tools for technology transfer. – 2004. – Т. 7, № 4. – С. 297–315. – Режим доступу: https://doi.org/10.1007/s10009-004-0153-x (дата звернення: 01.12.2023).
31. Generating finite state machines from abstract state machines [Електронний ресурс] / Wolfgang Grieskamp [та ін.] // ACM SIGSOFT software engineering notes. – 2002. – Т. 27, № 4. – С. 112–122. – Режим доступу: https://doi.org/10.1145/566171.566190 (дата звернення: 02.12.2023).
32. Optimal strategies for testing nondeterministic systems [Електронний ресурс] / Lev Nachmanson [та ін.] // ACM SIGSOFT software engineering notes. – 2004. – Т. 29, № 4. – С. 55–64. – Режим доступу: https://doi.org/10.1145/1013886.1007520 (дата звернення: 04.12.2023).
33. Grieskamp W. Action machines: a framework for encoding and composing partial behaviors [Електронний ресурс] / Wolfgang Grieskamp, Nicolas Kicillof, Nikolai Tillmann // International journal of software engineering and knowledge engineering. – 2006. – Т. 16, № 05. – С. 705–726. – Режим доступу:

https://doi.org/10.1142/s0218194006002963 (дата звернення: 29.11.2023).

34. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem [Електронний ресурс] / Mark Harman [та ін.] // 2010 IEEE international conference on software testing verification and validation workshop (ICSTW), Paris, France, 6–10 квіт. 2010 р. – [Б. м.], 2010. – Режим доступу: https://doi.org/10.1109/icstw.2010.31 (дата звернення: 29.11.2023).
35. Wegener J. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system [Електронний ресурс] / Joachim Wegener, Oliver Bühler // Genetic and evolutionary computation – GECCO 2004. – Berlin, Heidelberg, 2004. – С. 1400–1412. – Режим доступу: https://doi.org/10.1007/978-3-540-24855-2_160 (дата звернення: 27.11.2023).
36. Derderian K. Automated unique input output sequence generation for conformance testing of fsms [Електронний ресурс] / K. Derderian // The computer journal. – 2005. – Т. 49, № 3. – С. 331–344. – Режим доступу: https://doi.org/10.1093/comjnl/bxl003 (дата звернення: 27.11.2023).
37. Evolutionary testing of autonomous software agents [Електронний ресурс] / Cu D. Nguyen [та ін.] // Autonomous agents and multi-agent systems. – 2011. – Т. 25, № 2. – С. 260–283. – Режим доступу: https://doi.org/10.1007/s10458-011-9175-4 (дата звернення: 27.11.2023).
38. Automated test data generation for aspect-oriented programs [Електронний ресурс] / Mark Harman [та ін.] // The 8th ACM international conference, Charlottesville, Virginia, USA, 2–6 берез. 2009 р. – New York, New York, USA, 2009. – Режим доступу: https://doi.org/10.1145/1509239.1509264 (дата звернення: 27.11.2023).
39. Constructing test suites for interaction testing [Електронний ресурс] / M. B. Cohen [та ін.] // 25th international conference on software engineering, 2003. proceedings., Portland, OR, USA, 10 трав. 2003 р. – [Б. м.], 2003. – Режим доступу: https://doi.org/10.1109/icse.2003.1201186 (дата звернення:

27.11.2023).

40. Integration test of classes and aspects with a multi-evolutionary and coupling-based approach [Електронний ресурс] / Thelma Elita Colanzi [та ін.] // Search based software engineering. – Berlin, Heidelberg, 2011. – С. 188–203. – Режим доступу: https://doi.org/10.1007/978-3-642-23716-4_18 (дата звернення: 27.11.2023).
41. Harman M. Strong higher order mutation-based test data generation [Електронний ресурс] / Mark Harman, Yue Jia, William B. Langdon // The 19th ACM SIGSOFT symposium and the 13th European conference, Szeged, Hungary, 5–9 верес. 2011 р. – New York, New York, USA, 2011. – Режим доступу: https://doi.org/10.1145/2025113.2025144 (дата звернення: 27.11.2023).
42. TimeAware test suite prioritization [Електронний ресурс] / Kristen R. Walcott [та ін.] // The 2006 international symposium, Portland, Maine, USA, 17–20 лип. 2006 р. – New York, New York, USA, 2006. – Режим доступу: https://doi.org/10.1145/1146238.1146240 (дата звернення: 27.11.2023).
43. Improving network applications security [Електронний ресурс] / Concettina Del Grosso [та ін.] // The 2005 conference, Washington DC, USA, 25–29 черв. 2005 р. – New York, New York, USA, 2005. – Режим доступу: https://doi.org/10.1145/1068009.1068185 (дата звернення: 27.11.2023).
44. Alshahwan N. Automated web application testing using search based software engineering [Електронний ресурс] / Nadia Alshahwan, Mark Harman // 2011 26th IEEE/ACM international conference on automated software engineering (ASE), Lawrence, KS, USA, 6–10 листоп. 2011 р. – [Б. м.], 2011. – Режим доступу: https://doi.org/10.1109/ase.2011.6100082 (дата звернення: 25.11.2023).
45. What is the benefit of a model-based design of embedded software systems in the car industry? [Електронний ресурс] / Manfred Broy [та ін.] // Software design and development. – [Б. м.], 2014. – С. 310–334. – Режим доступу:

https://doi.org/10.4018/978-1-4666-4301-7.ch017 (дата звернення: 27.11.2023).
46. Selic B. The pragmatics of model-driven development [Електронний ресурс] / B. Selic // IEEE software. – 2003. – Т. 20, № 5. – С. 19–25. – Режим доступу: https://doi.org/10.1109/ms.200Z.1231146 (дата звернення: 27.11.2023).
47. Pyserial documentation [Електронний ресурс] // pythonhosted. – Режим доступу: https://pythonhosted.org/pyserial/ (дата звернення: 25.11.2023).
48. What is serial communication [Електронний ресурс] // coderey. – Режим доступу: https://www.codrey.com/embedded-systems/serial-communication-basics/ (дата звернення: 20.11.2023).
49. Sharma L. Test case specification [Електронний ресурс] / Lakshay Sharma // toolsqa. – Режим доступу: https://toolsqa.com/software-testing/test-case-specification/ (дата звернення: 18.11.2023).
50. PyTest documentation [Електронний ресурс] // pytest. – Режим доступу: https://docs.pytest.org/en/7.1.x/contents.html (дата звернення: 10.11.2023).
51. PyYaml documentation [Електронний ресурс] // pyyaml. – Режим доступу: https://pyyaml.org/wiki/PyYAMLDocumentation (дата звернення: 18.11.2023).

ДОДАТОК А

```
==> ./config/__init__.py <==
from config.device_secrets import DeviceSecrets
from config.loggers import stderr_logger, stdout_logger
from config.paths import FrameworkPaths

==> ./config/device_secrets.py <==
import os


class DeviceSecrets:
    USERNAME = os.getenv("DEVICE_USERNAME", "admin")
    PASSWORD = os.getenv("DEVICE_PASSWORD")

==> ./config/loggers.py <==
from collections.abc import Callable
from enum import Enum
from sys import stderr, stdout

from loguru import logger


class LoggerName(Enum):
    STDOUT = "stdout"
    STDERR = "stderr"


class LogLevel(Enum):
    TRACE = "TRACE"
    DEBUG = "DEBUG"
    INFO = "INFO"
    SUCCESS = "SUCCESS"
    WARNING = "WARNING"
    ERROR = "ERROR"
    CRITICAL = "CRITICAL"


def create_filter(name: LoggerName) -> Callable[[dict], bool]:
    """Creates filter which allows to bind a different logger with
    parameter `name`
    """
    def filter(record: dict):
```

```
        return record["extra"].get("name") == name
    return filter


DEFAULT_LOG_FORMAT = "{time} {level: <8} | {message}"
DEFAULT_LOG_LEVEL = LogLevel.INFO

# 2 loggers that can be used in tests. This will allow to create 3
separate
# categories in allure report (one is general logger by default)
logger.configure(
    handlers=[
        {
            "sink": stdout,
            "level": DEFAULT_LOG_LEVEL.value,
            "format": DEFAULT_LOG_FORMAT,
            "filter": create_filter(LoggerName.STDOUT)
        },
        {
            "sink": stderr,
            "level": DEFAULT_LOG_LEVEL.value,
            "format": DEFAULT_LOG_FORMAT,
            "filter": create_filter(LoggerName.STDERR)
        }
    ]
)


stdout_logger = logger.bind(name=LoggerName.STDOUT)
stderr_logger = logger.bind(name=LoggerName.STDERR)

==> ./config/paths.py <==
import os
from pathlib import Path


class FrameworkPaths:
    FRAMEWORK_ROOT = Path(os.getenv("FRAMEWORK_DIR", os.getcwd()))
    CONFIG_DIR = FRAMEWORK_ROOT / "config"
    RESOURCES_DIR = FRAMEWORK_ROOT / "resources"
    REPORTS_DIR = FRAMEWORK_ROOT / "reports"

    DEVICES_YAML = RESOURCES_DIR / "devices.yaml"
    COMMANDS_YAML = RESOURCES_DIR / "console_commands.yaml"
```

```
==> ./resources/console_commands.yaml <==
beep:
blink:
delay:
put:
quit:
ping:

system:
  reboot:
  identity:
    print:
  clock:
    print:

==> ./resources/devices.yaml <==
default:
  ip: null
  mac: null
  ssh_port: 22
  serial_connected: false
  serial_port: null
  baudrate: 115200

mikrotik_rb2011u1as:
  ip: 192.168.88.1
  mac: F44637415168
  ssh_port: 22
  serial_connected: true
  serial_port: /dev/ttyUSB0
  baudrate: 115200

dummy_device:
  ip: 127.0.0.1

==> ./src/connectors/ssh_connection.py <==
"""Library to enable connection to the device through SSH"""
import logging
import os

from paramiko import AutoAddPolicy, SSHClient
from paramiko.channel import Channel
from paramiko.config import SSHConfig

from config import stdout_logger
```

```
from src.connectors.base_connection import BaseConnection
from     src.connectors.exceptions     import     ConnectionTestError,
ReadTimeoutError


class SSHConnection(BaseConnection):
    """Library to enable connection to the device through SSH."""

    def __init__(
         self, ip_address: str, username: str, password: str | None =
None,
        port: int = 22, timeout: float = 10
    ):
        """Initializes the SSH connection

           Let me root needs to be present on the device for this
connection
        method to work.
        """
        super().__init__()

        self.ip_address = ip_address
        self.port = port
        self.username = username
        self.password = password
        self.timeout = timeout

        self.client = SSHClient()
        self.client.set_missing_host_key_policy(AutoAddPolicy())
        self.shell = None

        self.prompt = self.SHELL_PROMPT

        logging.basicConfig()
        logging.getLogger("paramiko").setLevel(logging.DEBUG)

    @staticmethod
    def get_config_file() -> SSHConfig | None:
        """Gets the config file from the user
        :return:
            - ssh_config: SSH config file
        """
        ssh_config = SSHConfig()
        ssh_config_file = os.path.expanduser("~/.ssh/config")
```

```
        if not os.path.isfile(ssh_config_file):
            return

        with open(ssh_config_file) as file:
            ssh_config.parse(file)

        return ssh_config

    def open_connection(self):
        """Opens the SSH connection and establishes a shell
        :raises:
            - ConnectionTestError if the login to the shell and echo
command is not
                successful
        """
        stdout_logger.info(f"Connecting to device {self.ip_address}:
{self.port}")
        self.client.connect(
            hostname=self.ip_address,
            port=self.port,
            username=self.username,
            password=self.password,
            timeout=10,
            look_for_keys=False,
            allow_agent=False
        )

        self.shell = self._create_shell()
        self.set_read_timeout(self.timeout)

        success, response = self._run_test_command()
        if not success:
            raise ConnectionTestError(response)
        stdout_logger.success("Connection established\n")

    def is_connected(self) -> bool:
        """Checks the connection status."""
        if self.client.get_transport() is not None:
            return self.client.get_transport().is_active()
        return False

    def _create_shell(self) -> Channel:
        shell = self.client.invoke_shell(term="vt100", width=512,
height=24)
        return shell
```

```
    def close_connection(self):
        """Closes the connection."""
                 stdout_logger.info(f"Closing  connection  to  device
{self.ip_address}:{self.port}")
        if self.is_connected():
            try:
                self.send_command("quit", timeout=1)
            except ReadTimeoutError:
                          stdout_logger.info("Console  is  no  longer
accessible")
        self.client.close()
        stdout_logger.success("Connection closed\n")

    def write(self, data: str):
        """Writes the data to the shell
        :param data: Data to write to the shell
        """
        self.shell.sendall(data)

    def read(self, count: int = 1) -> bytes:
        """Reads the data from the shell
        :param count: Number of bytes to read from the shell
        :return:
            - data: The data read (bytes)
        """
        data = b""
        if self.output_available():
            data = self.shell.recv(count)
        return data

    def output_available(self) -> bool:
        """The status of the output buffer for reads."""
        return self.shell.recv_ready()

    def get_read_timeout(self) -> float:
        """Gets the read timeout
        :return:
            - timeout: The timeout in seconds (int, float)
        """
        return self.shell.gettimeout()

    def set_read_timeout(self, timeout: float):
        """Sets the read timeout
        :param timeout: The timeout in seconds (int, float)
```

```
        """
        self.shell.settimeout(timeout)

==> ./src/connectors/serial_connection.py <==
"""Library to enable connection to the device through serial"""
from time import sleep

import serial

from config import stdout_logger
from src.connectors.base_connection import BaseConnection
from src.connectors.exceptions import ConnectionTestError


class SerialConnection(BaseConnection):
    """Library to enable connection to the device through serial."""

    def __init__(
        self, port: int, username: str, password: str | None = None,
        baudrate: int = 115200, timeout: float = 2
    ):
        """Initializes the serial connection instance

           Let me root needs to be present on the device for this
connection
        method to work.
        """
        super().__init__()

        self.timeout = timeout

        self.connection = serial.Serial()
        self.connection.port = port
        self.connection.baudrate = baudrate
        self.username = username
        self.password = password
        self.set_read_timeout(timeout)

        self.prompt = self.SHELL_PROMPT

    def login(self):
        # Enter newline to check for login prompt
        self.writeln()
        self.clear_output_buffer()
        self.writeln()
```

```
        sleep(1)

        response = self.read(200).decode()

        # Check for login prompt and login
        if "Login: " in response:
            self.clear_output_buffer()
            self.writeln(self.username)
            self.read_until("Password: ")
            self.writeln(self.password)
        else:
            self.writeln()

    def open_connection(self):
        """Opens the serial connection
        :raises:
             - ConnectionTestError if the login to the shell and echo
command is not
                successful
        """
                        stdout_logger.info(f"Connecting   to   device
{self.connection.port} {self.connection.baudrate}")
        self.connection.open()
        self.login()

        success, response = self._run_test_command()
        if not success:
            raise ConnectionTestError(response)
        stdout_logger.success("Connection established\n")

    def close_connection(self):
        """Closes the connection."""
                 stdout_logger.info(f"Closing  connection  to  device
{self.connection.port}")
        if self.is_connected():
            self.clear_output_buffer()
        self.connection.close()
        stdout_logger.success("Connection closed\n")

    def is_connected(self) -> bool:
        """Checks the connection status."""
        return self.connection.is_open

    def reboot(self):
        self.writeln("reboot")
```

```
        self.read_until("login: ", timeout=180)

        self.check_and_login()

    def write(self, data: str):
        """Writes the data to the shell
        :param data: Data to write to the shell
        """
        self.connection.write(data.encode())
        self.connection.flush()
        sleep(0.05)  # implicit sleep for all writes

    def read(self, count: int = 1) -> bytes:
        """Reads the data from the shell"""
        data = b""
        if self.output_available():
            data = self.connection.read(count)
        return data

    def output_available(self) -> bool:
        """The status of the output buffer for reads"""
        return bool(self.connection.in_waiting)

    def get_read_timeout(self) -> float:
        """Gets the read timeout"""
        return self.connection.timeout

    def set_read_timeout(self, timeout: float):
        """Sets the read timeout"""
        self.connection.timeout = timeout

==> ./src/connectors/exceptions.py <==
"""Connection error exceptions"""
from typing import Any


class ReadTimeoutError(Exception):
    def __init__(self, expected: str, got: str, timeout: float):
        self.expected = expected
        self.got = got
        self.timeout = timeout

    def __str__(self):
            return f"Expected '{self.expected}' in {self.timeout}
seconds. Only read '{self.got}'"
```

```
class ConnectionTestError(Exception):
    def __init__(self, response):
        self.response = response

    def __str__(self):
               return  f"Device  connection  test  failed.  Response:
{repr(self.response)}"


class ConnectionClosedError(Exception):
    def __str__(self):
        return "Connection is closed"


class NoSuchConnectionTypeError(Exception):
    def __init__(self, connection_type: Any):
        self.connection_type = connection_type

    def __str__(self):
        return f"No such connection type: {self.connection_type}"

==> ./src/connectors/__init__.py <==
from enum import Enum

from src.connectors.base_connection import BaseConnection
from src.connectors.exceptions import NoSuchConnectionTypeError
from src.connectors.serial_connection import SerialConnection
from src.connectors.ssh_connection import SSHConnection


class ConnectionType(Enum):
    SSH = "ssh"
    SERIAL = "serial"

==> ./src/connectors/base_connection.py <==
import re
from abc import ABC, abstractmethod
from threading import Lock
from time import time

from config import stdout_logger
from src.connectors.exceptions import ReadTimeoutError
```

```
class BaseConnection(ABC):
    """Base connection library containing higher level functions

      This library cannot be used alone. The individual connection
specific read
      and write functions must be defined in the inherited child
class.
    """

    SHELL_PROMPT = r"\[\w+@\w+\] [/\w]*> "

    def __init__(self):
        self.lock = Lock()

    def __del__(self):
        """Closes the connection when the object is destroyed."""
        self.close_connection()

    @abstractmethod
    def open_connection(self):
        """Opens the connection"""
        ...

    @abstractmethod
    def close_connection(self):
        """Closes the connection"""
        ...

    @abstractmethod
    def is_connected(self):
        """Checks the connection status"""
        ...

    @abstractmethod
    def output_available(self):
        """The status of the output buffer for reads."""
        ...

    @abstractmethod
    def get_read_timeout(self) -> float:
        """Gets the read timeout
        :return:
            - timeout: The timeout in seconds
        """
```

```
        ...

    @abstractmethod
    def set_read_timeout(self, timeout: float):
        """Sets the read timeout
        :param timeout: The timeout in seconds
        """
        ...

    @abstractmethod
    def write(self, data: str):
        """Writes the data to the shell
        :param data: Data to write to the shell
        """
        ...

    @abstractmethod
    def read(self, count: float = 1) -> bytes:
        """Reads the data from the shell
        :param count: Number of bytes to read from the shell
        :return:
            - data: The data read
        """
        ...

    def writeln(self, data: str = ""):
        """Writes the data to the shell with a newline
        :param data: Data to write to the shell
        """
        self.write(f"{data}\r\n")

    def restart_connection(self):
        """Restarts the connection by closing and reopening it."""
        self.close_connection()
        self.open_connection()

    def send_command(self, command: str, timeout: float = 15, strip:
bool = True) -> str:
        """Sends a command and waits for the shell prompt within
        the desired timeout

        :param command: The string command
        :param timeout: The timeout before the prompt is read
        :return:
            - response: The returned output
```

```
        """
        try:
            self.lock.acquire()
            self.clear_output_buffer()

            stdout_logger.info(f"=> {command}")
            self.writeln(command)
            self.read_until_prompt(timeout)

            data = self.read_until_prompt(timeout)
            stdout_logger.debug(f"Raw response: <{data}>")
            response = self._extract_response(command, data)

            self.clear_output_buffer()
        finally:
            self.lock.release()

        response = response.strip() if strip else response
        stdout_logger.info(f"=< {response}")

        return response

    def read_until(self, expected: str, timeout: float = 5) -> str:
        """Reads the shell until the expected string
        :param expected: The expected string
        :param timeout: The timeout
        :return:
            - output: The read output
        :raise:
               - ReadTimeout: If the expected string is not read in
time
        """
        max_time = time() + timeout
        expected_bytes = expected.encode()
        output = b""
        while time() < max_time:
            output += self.read(1)
            if expected_bytes in output:
                return output.decode()
           raise ReadTimeoutError(expected, output.decode().strip(),
timeout)

    def read_until_regexp(self, expected: str, timeout: float = 5):
        """Reads the shell until the regular expression
        :param expected: The expected regular expression
```

```
        :param timeout: The timeout
        :raise:
              - ReadTimeout: If the expected string is not read in
time
        """
        max_time = time() + timeout
        regexp = re.compile(expected)
        output = b""
        while time() < max_time:
            output += self.read(1)
            try:
                if regexp.search(output.decode()):
                    return output.decode()
            except UnicodeDecodeError:
                pass  # Bytes read is not complete
           raise ReadTimeoutError(expected, output.decode().strip(),
timeout)

    def read_until_prompt(self, timeout: float = 5) -> str:
        """Reads the shell until the shell prompt
        :param timeout: The timeout
        :return:
            - output: The read output
        :raise:
              - ReadTimeout: If the expected string is not read in
time
        """
        output = self.read_until_regexp(self.prompt, timeout)
        return output

    def clear_output_buffer(self, timeout: float = 0.5):
        """Clears the output buffer by reading
        The timeout is the amount of time to wait since
        the last read. It is reset every time new data is read
        :param timeout: The timeout
        """
        max_time = time() + timeout
        while time() < max_time:
            if self.output_available():
                self.read()
                max_time = time() + timeout

    def _extract_response(self, command: str, data: str) -> str:
        """Extract the response from the data read from the shell
        The response is in the format of
```

```
            `<command>\\n\\r<response>[admin@...] > `
        The response is the substring of starting from the command +
newline end
        to the beginning of username
        :param command: The command sent
        :param data: The data returned from the shell
        :return:
            - response: The returned output
        """

        # Clean Command and Data
        command = re.sub(r"[^\r]\n", r"\r\n", command)  # missing
carriage return
        data = re.sub(r"\r\n>;? ", "\r\n", data)  # block responses
        data = re.sub(r"\'", "'", data)  # escaped quotes

        start = data.find(command) + len(command) + 1  # 1 is new
line character length
        end = data.rfind("[admin@")
        response = data[start:end]

        return response

    def _run_test_command(self) -> tuple[bool, str]:
        """Runs test command and compares output
        :return:
            - success: test result status
            - response: response
        """
        self.send_command("beep")
        response = self.send_command("system identity print")
        success = response == "name: MikroTik"
            stdout_logger.info(f"Initial test finished. Successful:
{success}")
        return success, response

==> ./src/devices/device_parser.py <==
from pathlib import Path

from yaml import safe_load


class DevicesParser:
    """Parses yaml file with described devices.
     The file should have default section, where all default fields
```

```
are
      determined. If some device doesn't contain some field, the
default field
    will be used.
    """

    def __init__(self, devices_yaml: Path):
        raw_devices = self._load_devices(devices_yaml)
        self._default: dict = raw_devices.pop("default")

        self.devices: dict[str, dict] = {}
        for device, config in raw_devices.items():
                                            self.devices[device]    =
self.update_config_with_default_values(config)

      def  update_config_with_default_values(self,  config:  dict)  ->
dict:
        for key, val in self._default.items():
            if key not in config:
                config[key] = val

        return config

    @staticmethod
    def _load_devices(devices_yaml: Path) -> dict[str, dict]:
        with devices_yaml.open("r") as fobj:
            devices = safe_load(fobj.read())

        return devices

==> ./src/devices/__init__.py <==
from config import FrameworkPaths
from src.devices.device_parser import DevicesParser

DEVICES:               dict[str,               dict]                 =
DevicesParser(FrameworkPaths.DEVICES_YAML).devices

==> ./src/device_lib/device_lib.py <==
from yaml import safe_load

from config import DeviceSecrets, FrameworkPaths
from src.connectors import (
    BaseConnection,
    ConnectionType,
    NoSuchConnectionTypeError,
```

```
    SerialConnection,
    SSHConnection
)
from src.device_lib.exceptions import (
    NoSuchCommandError,
    NoSuchDeviceError,
    NoSuchSubCommandError
)
from src.devices import DEVICES


class Command:
    def __init__(self, connection: BaseConnection, name: str,
sub_commands: dict | None = None):
        self.connection = connection
        self.name = name
        self._sub_commands = sub_commands

    def __getattr__(self, name) -> "Command":
        sub_command = self._sub_commands.get(name) if
self._sub_commands else None
        if sub_command is None:
            raise NoSuchSubCommandError(self.name, name)

        return sub_command

    def __call__(self, *args, **kwargs) -> str:
        arguments = " ".join(args)
        return self.connection.send_command(f"{self.name}
{arguments}")


class DeviceConnection:
    def __init__(self, connection_type: ConnectionType, device_name:
str):
        self.device = DEVICES.get(device_name)
        if self.device is None:
            raise NoSuchDeviceError(device_name)

        if connection_type is ConnectionType.SSH:
            self.connection = SSHConnection(
                self.device["ip"],
                DeviceSecrets.USERNAME,
                DeviceSecrets.PASSWORD,
                self.device["ssh_port"]
```

```
            )
        elif connection_type is ConnectionType.SERIAL:
            self.connection  = SerialConnection(
                self.device["serial_port"],
                DeviceSecrets.USERNAME,
                DeviceSecrets.PASSWORD,
                self.device["baudrate"]
            )
        else:
            raise NoSuchConnectionTypeError(connection_type)

        self.connection.open_connection()

    def __del__(self):
        self.connection.close_connection()


class DeviceLib(DeviceConnection):
    def __init__(self, connection_type: ConnectionType, device_name:
str):
        super().__init__(connection_type, device_name)
        self._commands =self._parse_commands()

    def __getattr__(self, name: str) -> Command:
        command = self._commands.get(name)
        if command is None:
            raise NoSuchCommandError(name)
        return command

    def _parse_commands(self) -> dict[str, Command]:
        raw_cmds = self._load_raw_commands()
        commands = {}
        for cmd_name, sub_cmds in raw_cmds.items():
                commands.update(self._create_command(cmd_name,
sub_cmds))

        return commands

    def _create_command(self, name: str, sub_commands: dict | None =
None) -> dict[str, Command]:
        sub_command_objects = None

        if sub_commands:
            sub_command_objects = {}
            for sub_cmd_name, sub_sub_cmd in sub_commands.items():
```

```
                                                sub_command_objects   =
self._create_command(sub_cmd_name, sub_sub_cmd)

                    command_obj   =   Command(self.connection,   name,
sub_command_objects)
        return {name: command_obj}

    @staticmethod
    def _load_raw_commands() -> dict[str, dict | None]:
        with open(FrameworkPaths.COMMANDS_YAML) as fobj:
            return safe_load(fobj.read())

==> ./src/device_lib/exceptions.py <==
class NoSuchDeviceError(Exception):
    def __init__(self, device_name: str):
        self.device_name = device_name

    def __str__(self):
        return f"{self.device_name} device doesn't exist"


class NoSuchCommandError(Exception):
    def __init__(self, cmd_name: str):
        self.cmd_name = cmd_name

    def __str__(self):
        return f"{self.cmd_name} command doesn't exist"


class NoSuchSubCommandError(Exception):
    def __init__(self, cmd_name: str, sub_cmd_name: str):
        self.cmd_name = cmd_name
        self.sub_cmd_name = sub_cmd_name

    def __str__(self):
         return f"Subcommand {self.sub_cmd_name} doesn't exist under
{self.sub_name} command"

==> ./tests/framework_checks/smoke_test.py <==
from pytest import fixture, mark, param

from src.connectors import ConnectionType
from src.device_lib.device_lib import DeviceLib
```

```
@fixture(scope="function", autouse=True)
def device_lib(request, device_name):
                                          connection_type           =
request.node.callspec.params.get("connection_type_under_test")
    device_lib = DeviceLib(connection_type, device_name)
    yield device_lib
    device_lib.connection.close_connection()


@mark.parametrize(
    "connection_type_under_test",
    (param(ConnectionType.SSH, marks=mark.ssh_only),
     param(ConnectionType.SERIAL, marks=mark.serial_only))
)
def test_connection(device_lib, connection_type_under_test):
    assert device_lib.connection.is_connected()
    device_lib.beep()

==> ./tests/conftest.py <==
from pytest import fixture, mark

from src.connectors import ConnectionType, NoSuchConnectionTypeError
from src.device_lib.device_lib import DeviceLib


# #####
# Hooks
# #####

def pytest_addoption(parser):
    """Configures the command line options"""
    parser.addoption(
        "--connection",
        action="store",
        help="Connection type",
        default="ssh"
    )
    parser.addoption(
        "--device",
        action="store",
        help="Device name"
    )


def pytest_collection_modifyitems(session, config, items):
```

```
    """Determines which tests need to be skipped based on
    * Connection
    """
    connection = config.getoption("--connection").lower()

    for item in items:
        markers = tuple(item.iter_markers())

        for marker in markers:
            # Skip by connection
                  if "_only" in marker.name and connection not in
marker.name:
                item.add_marker(
                       mark.skip(reason=f"Test requires {connection}
connection"))


# ########
# Fixtures
# ########
@fixture(scope="session")
def device_name(request):
    return request.config.getoption("--device")


@fixture(scope="session")
def connection_type(request):
    connection = request.config.getoption("--connection")

    if connection.lower() == "ssh":
        connetion_type = ConnectionType.SSH
    elif connection.lower() == "serial":
        connetion_type = ConnectionType.SERIAL
    else:
        raise NoSuchConnectionTypeError(connection)

    return connetion_type


@fixture(scope="session")
def device_lib(connection_type, device_name):
    device_lib = DeviceLib(connection_type, device_name)
    yield device_lib
    device_lib.connection.close_connection()
```

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗОВАНИХ СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА
на тему: «Розробка фреймворку автоматичного тестування вбудованих систем на основі pytest»

Виконав: студент групи КСДМ-61, Гакман Дмитро
Керівник: к.т.н., доцент, Антоненко Артем

Мета, обʼєкт та предмет роботи

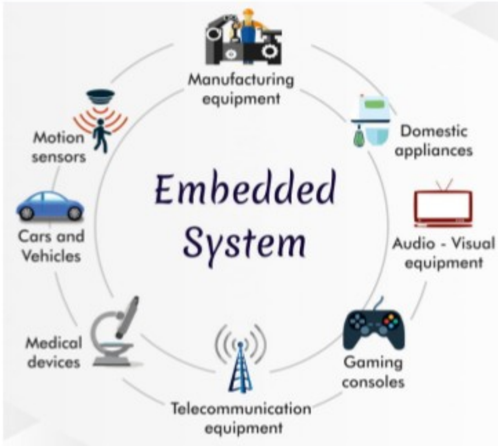
Мета роботи – розробка комплексу для тестування вбудованих систем з урахуванням їхніх специфічних особливостей, включаючи теоретичну базу та практичну реалізацію фреймворку.

Об'єкт дослідження – процес розробки комплексу для тестування вбудованих систем, у тому числі вибір та визначення вимог до вбудованого пристрою для тестування, а також аналіз та вибір оптимальної архітектури фреймворку.

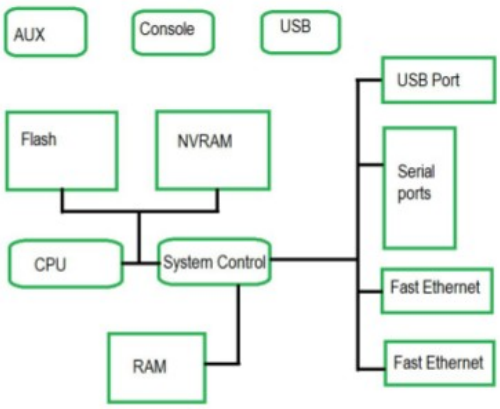
Предмет дослідження – фреймворк для тестування вбудованих систем, включаючи компоненти, такі як з'єднання з пристроєм, роботу з консольними командами, логіку тестування та роботу з тестовими даними.

Звіт BCC Research Team

Глобальний ринок вбудованих систем має зрости з 207,3 мільярдів доларів США у 2020 році до 267,3 мільярдів доларів США до 2025 року зі зведеним річним темпом зростання (CAGR) 5,2% за період 2020-2025 років.



Маршрутизатор як пристрій для тестування



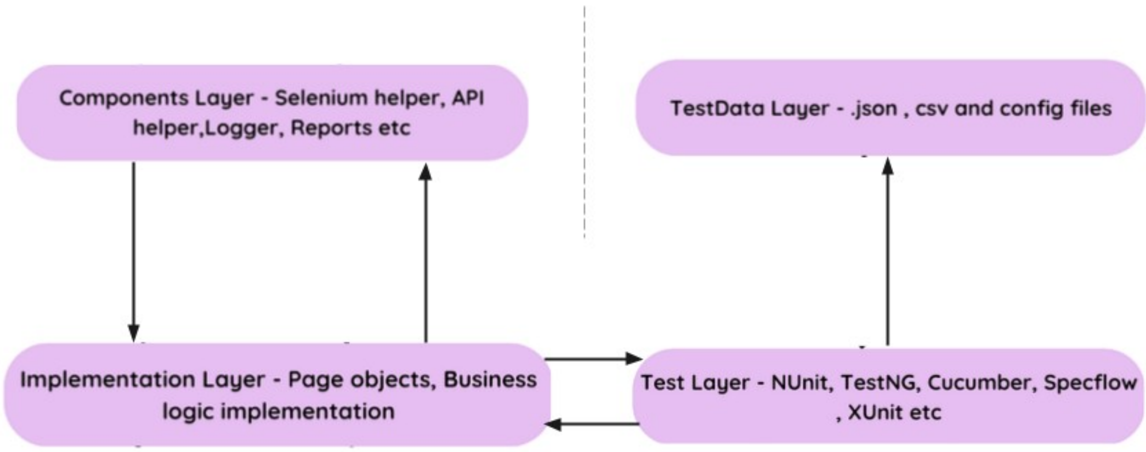
1. Обмежені обчислювальні ресурси в порівнянні з сучасними серверами чи комп'ютерами.

2. Вбудовані операційні системи з командним рядком, такі як Cisco IOS, RouterOS або OpenWRT.

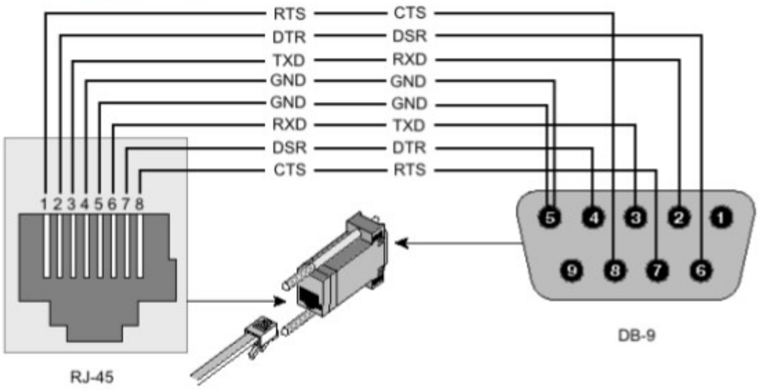
3. Різноманітні методи підключення: Ethernet, Wi-Fi, під'єднання по SSH чи Serial, інші мережеві протоколи та інтерфейси.

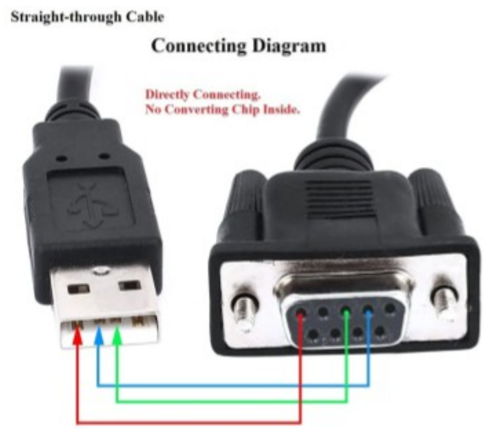
4. Докладну та доступна документація, включаючи технічні специфікації, керівництва з налаштування та інструкції з використання.

Архітектура фреймворку



Serial connection

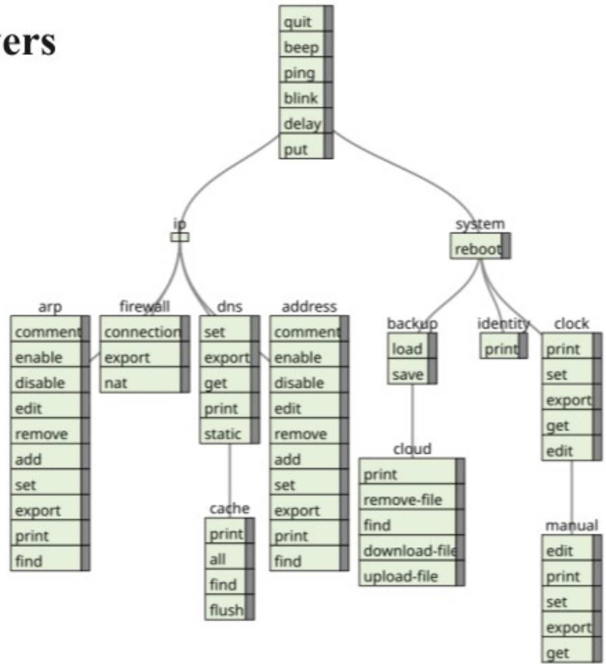




Implementation & Test data layers

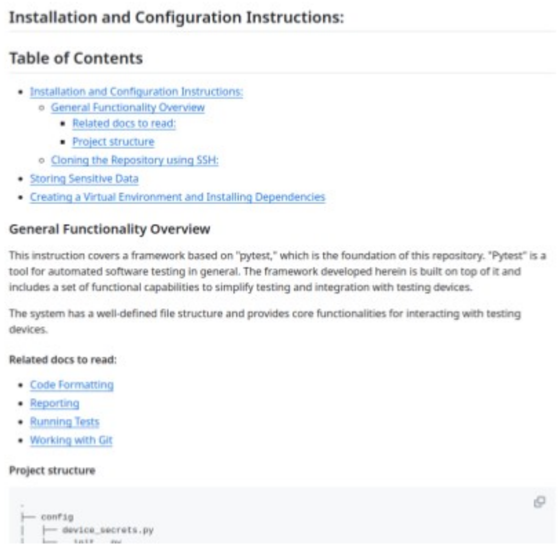

```yaml
default:
  ip: null
  mac: null
  ssh_port: 22
  serial_port: null
  baudrate: 115200

mikrotik_rb2011u1as:
  ip: 192.168.88.1
  mac: F44637415168
  serial_port: /dev/ttyUSB0
```

Методика використання системи

- Інструкція з установки та налаштування
- Запуск тестів та опції для Pytest
- Збереження, перегляд та розповсюдження звітів pytest-allure
- Правила форматування коду
- Правила роботи з репозиторієм
- Створення запиту на злиття (Pull Request)



Installation and Configuration Instructions:

Table of Contents

General Functionality Overview

This instruction covers a framework based on "pytest," which is the foundation of this repository. "Pytest" is a tool for automated software testing in general. The framework developed herein is built on top of it and includes a set of functional capabilities to simplify testing and integration with testing devices.

The system has a well-defined file structure and provides core functionalities for interacting with testing devices.

Related docs to read:

- Code Formatting
- Reporting
- Running Tests
- Working with Git

Project structure

```
.
├── config
│   ├── device_secrets.py
```

Висновки

- Проведено детальний аналіз існуючих архітектур фреймворків, оцінено їхні переваги та недоліки, на основі яких розроблено структуру фреймворку.

- Визначено вимоги до вбудованого пристрою для тестування, та розроблено модулі що безпосередньо працюють із вбудованим пристроєм через протоколи SSH та Serial, щоб забезпечити зв'язок із пристроєм для виконання тестів.

- Створено систему, яку можна використовувати як базис для подальшого розширення автоматизації тестування вбудованих девайсів, відрізняючись від загальних фреймворків, які часто не враховують їх особливостей.

- Розроблено документацію, спрямовану на інструктування інженерної команди щодо коректного використання розробленої системи.

Дякую за увагу!

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Дмитро ГАКМАН

Назва: Розробка фреймворку автоматичного тестування вбудованих систем на основі pytest

Координатор: Артем АНТОНЕНКО

Підрозділ: ННІІТ

Коефіцієнт подібності 1: 3.1

Коефіцієнт подібності 2: 1.9

Тривога: 1

Після аналізу Звіту подібності констатую наступне:

☐ виявлені в роботі запозичення є сумлінними і не мають ознак плагіату. Тому робота визнається самостійною і допускається до захисту;

☐ виявлені в роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і самостійності її автора. Роботу направити на доопрацювання;

☐ виявлені в роботі запозичення є недобросовісними і мають ознаки плагіату або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. У зв'язку з чим, робота не допускається до захисту.

……………………… ……………………………………

Дата *Підпис Наукового керівника*