Державний торговельно-економічний університет

Кафедра комп’ютерних наук та інформаційних систем

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА
на тему:

«Розробка фреймворка для тестування Web-додатків»

Студента 4 курсу, 5 групи,
спеціальності
122 «Комп’ютерні науки» _____ Кондратенко Максим Ігоревич
підпис студента

Науковий керівник
кандидат технічних наук, доцент _____ Селезньова Руслана Віталіївна
підпис керівника

Гарант освітньої програми
кандидат технічних наук, доцент _____ Демідов Павло Георгійович
підпис керівника

Київ 2024

Державний торговельно-економічний університет

Факультет інформаційних технологій
Кафедра комп'ютерних наук та інформаційних систем
Спеціальність 122 «Комп'ютерні науки»
Освітня програма «Комп'ютерні науки»

Затверджую
Зав. кафедри _____Пурський О.І.
«18» грудня 2023р.

Завдання
на випускну кваліфікаційну роботу студенту

Кондратенку Максиму Ігоревичу
(прізвище, ім'я, по батькові)

1. Тема випускної кваліфікаційної роботи
«Розробка фреймворка для тестування Web-додатків»
Затверджена наказом ректора від *«27» листопада 2023 р. № 4175*

2. Строк здачі студентом закінченої роботи *31 травня 2024 року*

3. Цільова установка та вихідні дані до роботи
Мета роботи: розробка шляхів підвищення ефективності автоматизованого тестування та спрощення розробки сценаріїв для такого тестування за допомогою використання металінгвістичних абстракцій та слів, що входять до словника основної мови програмування.
Об'єкт дослідження: процес автоматизованого тестування програмного забезпечення
Предмет дослідження: моделі, методи та інформаційні технології автоматизованого тестування програмного забезпечення.

4. Перелік графічного матеріалу_____
_____
_____
_____

5. Консультанти по роботі із зазначенням розділів, за якими здійснюється

консультування:

Розділ	Консультант (прізвище, ініціали)	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Пурський О.І.	22.12.2023 р.	22.12.2023 р.
2	Пурський О.І.	22.12.2023 р.	22.12.2023 р.
3	Пурський О.І.	22.12.2023 р.	22.12.2023 р.

6. Зміст випускного кваліфікаційної роботи (перелік питань за кожним розділом)

ВСТУП

РОЗДІЛ 1 ОГЛЯД МЕТОДІВ ТА ЗАСОБІВ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

1.1 Класична модель тестування

1.1.1 Модульне тестування

1.1.2 Сервісне тестування

1.1.3 Тестування користувацького інтерфейсу

1.2 Розширена модель тестування

1.2.1 Тестування контрактів взаємодії

1.2.2 Тестування продуктивності та швидкодії

1.2.3 Тестування безпеки

1.2.4 Тестування локалізації

1.3 Архітектура фреймворків автоматизованого тестування

Висновки до розділу 1

РОЗДІЛ 2 ПРОЄКТУВАННЯ ЗАСОБУ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

2.1 Аналіз вимог до програмного забезпечення

2.2 Архітектура програмного забезпечення

2.3 Реалізація функціоналу програмного забезпечення

Висновки до розділу 2

РОЗДІЛ 3 РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

3.1 Мета та порядок досліджень

3.2 Порівняння шляхом розрахунку відстані Левенштейна

3.3 Порівняння шляхом підрахунку слів та символів

3.4 Порівняння швидкодії фреймворків тестування

Висновки до розділу 3

ВИСНОВКИ

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

ДОДАТКИ

7. Календарний план виконання роботи

№ Пор.	Назва етапів випускної кваліфікаційної роботи	Строк виконання етапів роботи	
		За планом	фактично
1	2	3	4
1	*Вибір теми випускної кваліфікаційної*	*05.10.2023*	*05.10.2023*

	роботи		
2	*Розробка та затвердження завдання на випускну кваліфікаційну роботу*	*18.12.2023*	*18.12.2023*
3	*Вступ*	*02.02.2024*	02.02.2024
4	*РОЗДІЛ 1. Огляд методів та засобів автоматизації тестування*	*26.02.2024*	*26.02.2024*
5	*РОЗДІЛ 2. Проєктування засобу автоматизації тестування*	*05.04.2024*	*05.04.2024*
6	*РОЗДІЛ 3. Результати експериментальних досліджень*	*10.05.2024*	*10.05.2024*
7	*Висновки*	*15.05.2024*	*15.05.2024*
8	*Здача випускної кваліфікаційної роботи на кафедру науковому керівнику*	*20.05.2024*	*20.05.2024*
9	*Попередній захист випускної кваліфікаційної роботи*	*27.05.2024*	*27.05.2024*
10	*Виправлення зауважень, зовнішнє рецензування випускної кваліфікаційної роботи*	*28.05.2024*	*28.05.2024*
12	*Представлення готової зшитої випускної кваліфікаційної роботи на кафедру*	*06.12.2023*	*06.12.2023*
13	*Публічний захист випускної кваліфікаційної роботи*	*За розкладом роботи ЕК*	

Дата видачі завдання «22» грудня 2023 р.

9. Керівник випускної кваліфікаційної роботи Селезньова Р.В
(прізвище, ініціали, підпис)

10. Гарант освітньої програми Демідов П.Г.
(прізвище, ініціали, підпис)

11. Завдання прийняв до виконання студент Кондратенко М.І
(прізвище, ініціали, підпис)

12. Відгук керівника випускної кваліфікаційної роботи

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

Керівник випускної кваліфікаційної роботи

_____
(підпис, дата)

13. Висновок про випускну кваліфікаційну роботу

Випускна кваліфікаційна робота студента Кондратенко М. І.
(прізвище, ініціали)
може бути допущена до захисту в екзаменаційній комісії.

Гарант освітньої програми Демідов П.Г.
(підпис, прізвище, ініціали)

Завідувач кафедри Пурський О.І.
(підпис, прізвище, ініціали)

«_____»_____2024 р.

Анотація

У випускній кваліфікаційній роботі здійснено комплексну засобу автоматизації тестування Web-додатків з метою підвищення ефективності автоматизованого тестування. Теоретично обґрунтовано застосування різноманітних методів та засобів автоматизації тестування. Проведено проєктування засобу автоматизації тестування Web-додатків. Проведено аналіз реалізації функціоналу програмного забезпечення та викладено результати експериментальних досліджень застосування фреймворку.

Ключові слова: фреймворк, Web-додаток, програмне забезпечення, автоматизація, тестування.

Anotation

In the graduation qualification work, a complex tool for automating the testing of Web applications was implemented in order to improve the efficiency of automated testing. The application of various methods and means of testing automation is theoretically substantiated. The tool for automating the testing of Web applications has been designed. The analysis of the implementation of the functionality of the software was carried out and the results of experimental studies of the application of the framework were presented.

Keywords: framework, Web application, software, automation, testing.

ЗМІСТ

ВСТУП

Актуальність теми. Необхідність наукових досліджень в цій сфері виникає з того, що, незважаючи на швидкий розвиток програмного забезпечення та його постійно ростучу складність, інструменти для автоматизованого тестування рідко адаптуються до нових бізнес-вимог, змін у мовах програмування, а також не надають достатньої гнучкості при виборі стратегій тестування. Тестові фреймворки для платформи Node.js, як правило, орієнтовані на специфічні методології тестування, а інтеграція різних видів тестування в одному проекті вимагає додавання додаткових програмних залежностей та координації їх взаємодії.

У початкові періоди розвитку галузі розробки програмного забезпечення, тестування отримувало обмежену увагу. Тестери використовували ручні методики для перевірки функціональності програмного забезпечення та виявлення помилок у бізнес-логіці або інтерфейсі користувача. Проте з посиленням розвитку технологій, мов програмування та збільшенням складності програмних систем, з’являлися нові виклики, які вимагали поліпшення методів та інструментів тестування.

Автоматизоване тестування відіграє важливу роль в таких методологіях розробки, як екстремальне програмування та тестово-орієнтована розробка. В рамках цих методологій, сценарії для тестування формуються ще до того, як починається розробка окремого модуля програмного продукту, і продовжують розширюватися в процесі розвитку продукту.

Ціль автоматизованого тестування полягає в проведенні тестових сценаріїв, які були попередньо розроблені інженерами-тестувальниками за допомогою відповідних методологій, а також у перевірці та порівнянні результатів цих тестів з визначеними або очікуваними результатами. Ці перевірки проводяться за допомогою спеціалізованих інструментів та фреймворків, які є незалежними від розробленого програмного продукту.

Сучасні проблеми тестування можна вирішити шляхом дослідження існуючих методів та засобів автоматизованого тестування, їх вдосконалення або розробки нових підходів до автоматизації тестування.

Метою дослідження є розробка шляхів збільшення ефективності автоматизованого тестування та спрощення розробки сценаріїв для такого тестування за допомогою використання металінгвістичних абстракцій та слів, що входять до словника основної мови програмування.

Об’єктом дослідження є процес автоматизованого тестування програмного забезпечення.

Предметом дослідження є моделі, методи та інформаційні технології автоматизованого тестування програмного забезпечення.

Для досягнення мети роботи необхідно виконати наступні завдання:

- вивчити існуючі методики та інструменти для створення сценаріїв автоматизованого тестування;
- проаналізувати доступні інструменти для автоматизації тестування на платформі Node.js;
- вдосконалити методики та інструменти для побудови сценаріїв автоматизованого тестування за допомогою технік метапрограмування;
- створити фреймворк для автоматизованого тестування на платформі Node.js;
- провести експериментальні випробування характеристик запропонованих рішень.

Структура роботи: курсова робота обсягом 61 сторінок складається зі вступу трьох розділів, висновків, списку використаних джерел, який містить 28 найменувань. Робота містить 3 таблиці, 12 рисунків, 3 додатки.

РОЗДІЛ 1 ОГЛЯД МЕТОДІВ ТА ЗАСОБІВ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

1.1 Класична модель тестування

Ручне тестування складних систем було трудомістким і вимагало значних зусиль, оскільки тестові сценарії багаторазово виконувалися і часто не потребували значних змін. Це актуалізувало питання автоматизації процесів тестування та контролю якості програмного забезпечення.

Згідно зі звітом «Pulse of the Profession 2019» [1], більшість команд використовує методології гнучкої або гібридної розробки програмного забезпечення, такі як SCRUM та Kanban.

У цьому ітеративному процесі гнучкої розробки контролю якості надається високий пріоритет для забезпечення швидкої та ефективної роботи. Разом із ручним тестуванням команди все частіше застосовують автоматизоване тестування, що спрощує виконання тестових сценаріїв і значно зменшує потребу в ручному тестуванні.

Автоматизація тестування дозволяє виявляти дефекти в програмах за кілька хвилин після внесення змін у вихідний код, тоді як ручне тестування зайняло б години або дні.

Світові тенденції показують, що темпи розробки програмного забезпечення зростають з кожним роком, що робить автоматизацію тестування важливою в таких сферах, як неперервна інтеграція (continuous integration) та доставлення (continuous delivery) програмного забезпечення [4].

Термін «автоматизоване тестування» вперше запропонував Фредерік Брукс [2]. У своїй книзі він наводить детальні приклади необхідності тестування для

забезпечення концептуальної цілісності продукту програмного, зокрема через використання тестування модульного.

Міжнародна рада з питань сертифікації тестування програмного забезпечення розглядає тестування автоматизоване як застосування програмних рішень для підтримки тестувальних процесів, що включають керування сценаріями, створення дизайну сценаріїв та аналіз результатів тестування [3].

Автоматизація тестування надає командам можливість ефективно та повторно проводити тести, які при ручному тестуванні були б вкрай трудозатратними. Цей метод широко застосовується для регресивного тестування - перевірки програмного забезпечення після внесення змін з метою виявлення дефектів або змін у незмінених частинах через внесені модифікації [3].

Фреймворки служать повністю обладнаними рішеннями для тестування та стають фундаментом для тестувального інженера. Вони надають програмну підтримку, тестове середовище та визначене архітектурне рішення для створення, управління, виконання та аналізу сценаріїв.

З іншого боку, спеціалізовані інструменти для тестування можуть бути спеціально розроблені для певної системи, середовища або процесу. Наприклад, статичний аналізатор коду, який допомагає виявляти помилки у синтаксисі та друкарські помилки.

Багато компаній розглядають автоматизацію тестування програмного забезпечення як спосіб зменшення витрат на тестування та прискорення циклу розробки.

Однак рішення про впровадження автоматизованого тестування може бути неефективним, якщо це не відбувається у відповідному контексті та з урахуванням потреб конкретної організації. Наприклад, повторне виконання тестових сценаріїв і використання тих самих методів тестування може призвести до пропуску деяких дефектів у програмному забезпеченні.

Цей явище отримало назву "парадокс пестицидів" і вперше було описане Борисом Бейзером у книзі "Software Testing Techniques". Він порівняв цю стратегію тестування з повторною обробкою полів від шкідників за допомогою пестицидів у сільському господарстві.

Іншим аспектом, який ускладнює автоматизацію тестування, є необхідність володіння навичками програмування у тестувальників, оскільки сценарії тестування часто формулюються у вигляді програмного коду, який потім автоматично виконується у середовищі тестування.

Ситуацію ускладнює складність сучасних систем з їх гіллястою архітектурою, програмними залежностями між їх компонентами і потребою створення та налаштування спеціальної інфраструктури для їх функціонування. Вимога до тестувальників мати розуміння програмування може бути вирішена передачею написання сценаріїв тестування розробникам програмного забезпечення, але це часто призводить до додаткових витрат або сповільнення темпів розробки.

Дослідження, яке включало аналіз 78 різноманітних джерел, виявило, що проблеми, пов'язані з невдалим використанням автоматизованого тестування, можна розділити на п'ять категорій факторів: проблеми, що виникають у зв'язку з програмним забезпеченням та його архітектурою, труднощі, пов'язані з методиками тестування та написанням сценаріїв тестування, проблеми, що стосуються інструментів для проведення тестування, а також людські й організаційні аспекти, а також комбінацію факторів з цих груп [5].

Дослідження існуючих методів і засобів для автоматизованого тестування, їх вдосконалення або розробка нових підходів можуть змінити ситуацію та вирішити сучасні проблеми у цій галузі.

Наприклад, створення мови для опису сценаріїв тестування, яка була б одночасно зрозумілою та простою для написання, але в той же час використовувала б мову програмування для їх виконання, може допомогти вирішити проблему

необхідності навичок програмування у тестувальників. Це, в свою чергу, залучить більше людей до процесу тестування.

Перед появою гнучких методологій розробки ПЗ, таких як SCRUM, автоматизоване тестування було широко застосовано для спрощення та прискорення тестувальних процесів. Однак автоматизовані тести виявилися вкрай складними у створенні та вимагали глибокого розуміння архітектури системи для їхньої розробки.

Згідно з класифікацією, запропонованою Майком Коном у його роботі «Succeeding with Agile» [6], стратегію автоматизації тестування можна розділити на три рівні:

- модульне тестування;
- сервісне тестування;
- тестування користувацького інтерфейсу.

Ці рівні утворюють традиційну "піраміду тестування", яка представлена на рисунку 1.1.

Рисунок 1.1 – «Класична піраміда тестування» за Майком Коном

На кожному наступному рівні тестування, починаючи з модульних, розширюються тестові сценарії попередніх рівнів, а також додаються нові сценарії інтеграції частин системи. Кількість тестів на кожному рівні залежить від його позиції у "піраміді".

Характерною особливістю є те, що разом зі зростанням рівня тестування збільшуються і витрати часу на розробку тестових сценаріїв. Тести більш високого рівня відрізняються великою крихкістю, оскільки навіть незначні зміни у системі можуть зробити їх непридатними в значній кількості [6].

1.1.1 Модульне тестування

Модульне тестування - це процес перевірки окремих частин програмного забезпечення, відомих як модулі, на предмет правильності їхньої роботи. Це важливий елемент в циклі розробки програмного забезпечення, який допомагає забезпечити якість продукту.

Модульне тестування - це тип тестування, який зосереджується на найменших одиницях програмного забезпечення, відомих як модулі. Модулі можуть бути окремими функціями, процедурами, методами або класами в коді. Ціль модульного тестування - переконатися, що кожний окремий модуль працює правильно в ізоляції від решти системи.

Модульне тестування допомагає виявити та виправити помилки на ранніх стадіях розробки, коли вони найлегше виправити. По-друге, воно допомагає забезпечити, що кожний модуль працює правильно перед тим, як він інтегрується з рештою системи. Це може допомогти запобігти проблемам, які можуть виникнути під час інтеграції.

Модульне тестування зазвичай включає створення та виконання тестових сценаріїв для кожного модуля. Ці сценарії мають перевірити, чи правильно модуль

виконує свої функції при різних вхідних даних. Це може включати перевірку відповідей модуля на коректні вхідні дані, а також на некоректні або неочікувані вхідні дані.

Це процес тестування окремих модулів або компонентів програми для забезпечення їхньої коректної роботи. В основі модульного тестування лежить принцип розділення програми на дрібні частини, кожну з яких можна перевірити окремо, що дозволяє виявляти помилки на ранніх етапах розроблення.

Модульне тестування орієнтоване на найменші функціональні частини програми, зазвичай окремі функції або методи. Кожен модуль тестується ізольовано від інших, щоб перевірити, чи працює він правильно в різних ситуаціях. Для цього створюються тестові сценарії, які містять вхідні дані та очікувані результати. Якщо фактичний результат співпадає з очікуваним, тест вважається успішним.

Модульне тестування зазвичай виконується розробниками програмного забезпечення під час фази розробки. Використовуються спеціальні фреймворки, такі як JUnit для Java, NUnit для .NET, pytest для Python та багато інших. Ці інструменти автоматизують процес тестування, що значно підвищує його ефективність.

Модульне тестування має декілька ключових переваг:

- раннє виявлення помилок (оскільки кожен модуль тестується окремо, помилки можна знайти і виправити ще на ранніх етапах розробки. Це знижує вартість виправлення помилок, оскільки виявлення дефектів на пізніх етапах може бути значно дорожчим);
- полегшення змін та рефакторингу (тести надають впевненість, що зміни в коді не призведуть до неочікуваних проблем. Це особливо важливо під час рефакторингу коду, коли змінюється структура програми без зміни її функціональності);

- покращення якості коду (модульне тестування стимулює розробників писати чистіший, більш структурований код. Модулі повинні бути незалежними та мати чітко визначені інтерфейси, що сприяє кращій організації коду);
- документація (тестові випадки можуть слугувати додатковою документацією для коду, демонструючи, як він повинен працювати в різних ситуаціях).

Практичне застосування модульного тестування включає кілька етапів:

- написання тестових випадків (на цьому етапі розробник створює тестові сценарії для кожного модуля. Важливо передбачити як позитивні, так і негативні випадки, щоб переконатися, що модуль правильно обробляє всі можливі ситуації);
- запуск тестів (тести запускаються автоматично за допомогою фреймворків модульного тестування. Фреймворки забезпечують зручний спосіб організації та виконання тестів);
- аналіз результатів (розробники аналізують результати тестів. Якщо тест не проходить, необхідно виявити причину та виправити помилку в коді);
- регресійне тестування (після внесення змін до коду важливо повторно виконати всі тести, щоб переконатися, що нові зміни не порушили існуючу функціональність).

Модульне тестування - це важливий аспект, який допомагає забезпечити якість кінцевого продукту. Воно допомагає зрозуміти, що кожний модуль працює правильно перед тим, як він інтегрується з рештою системи. Завдяки модульному тестуванню розробники можуть бути впевнені, що їх забезпечення програмне відповідає вимогам та очікуванням користувачів.

Модульне тестування є невід'ємною частиною сучасного процесу розробки програмного забезпечення. Воно допомагає забезпечити високу якість коду, сприяє ранньому виявленню помилок та полегшує процес підтримки та рефакторингу програм. Завдяки автоматизації та використанню спеціалізованих фреймворків,

модульне тестування стало доступним та ефективним інструментом для розробників, що прагнуть створювати надійні та якісні програми.

Модульне тестування лежить в основі "піраміди тестування" і гарантує, що найменші складові забезпечення програмного, такі як функції, методи та класи, працюють відповідно до вимог. За словами Майка Кона [6], ці тести мають становити основну частину автоматизованого тестування, і кількість сценаріїв тестування на цьому рівні повинна бути найбільшою. Зазвичай розробники формують модульні тести без залучення тестувальників, а самі тестові сценарії є програмним кодом, який перевіряє певний функціонал, ізольовано від інших елементів системи. Модульні тести детальні і за допомогою регресивного тестування ефективно виявляють помилки в коді. Завдяки ізольованості тестування, легко встановити місце виникнення дефекту. Це процес перевірки окремих частин програми, відомих як модулі, для виявлення помилок та забезпечення їх правильної роботи.

Перша і найважливіша перевага модульного тестування полягає в тому, що воно дозволяє розробникам ідентифікувати та виправляти помилки на ранніх стадіях розробки. Це може значно скоротити витрати на виправлення помилок, оскільки виправлення помилок на пізніших стадіях може бути дорогим та трудомістким.

Модульне тестування також сприяє кращому розумінню коду. Коли розробники тестують окремі модулі, вони мають можливість глибше зрозуміти, як працює кожна частина програми. Це може поліпшити якість коду та зробити його більш стійким до помилок.

Однак модульне тестування не обмежується лише виявленням помилок. Воно також допомагає забезпечити, що програма відповідає вимогам та очікуванням користувачів. Це може бути особливо корисним при розробці програмного забезпечення, яке повинно відповідати строгим стандартам якості та надійності.

Отже, модульне тестування допомагає забезпечити якість продукту, зменшує витрати на виправлення помилок та допомагає розробникам краще розуміти свій код. Тому важливо включати модульне тестування в процес розробки забезпечення програмного.

В процесі модульного тестування, ізоляція компонентів системи часто здійснюється шляхом заміни залежних елементів. Це передбачає розробку спеціалізованого програмного компонента, який під час тестування виступає в ролі тих компонентів, на які посилається або від яких залежить тестовий компонент. Ізоляція компонентів допомагає уникнути небажаних ефектів під час тестування та звужує область пошуку помилок при їх виявленні, оскільки тестовий компонент не залежить від інших компонентів. В окремих ситуаціях розробники можуть не звертати увагу на ізоляцію компонентів системи під час проведення модульного тестування. Така ситуація може виникнути, коли можливі побічні ефекти, які вибрано ігнорувати, наприклад, при використанні вбудованих компонентів мови програмування.

Існує загальновідома проблема ізоляції компонентів в мовах програмування, які дозволяють обмежувати безпосередній доступ до компонентів системи з метою приховування їх специфічної реалізації. У таких обставинах, якщо компонент залежить від іншого компонента, доступ до якого обмежено, можна відмовитися від цієї залежності. Таке рішення вкрай небажане, оскільки порушує принципи мови програмування і створює розбіжність між середовищами тестування та виробничого виконання програми [9].

У своєму творі “xUnit Test Patterns: Refactoring Test Code”, Джерард Мецарос виділив низку методів для ізоляції компонентів системи під час модульного тестування, які він згрупував під назвою “двійники” (англ. test doubles). Принципи імітації, які вони застосовують, мають схожий характер, проте робота цих

компонентів під час виконання сценарію тестування відрізняється значно. Джерард ідентифікував наступні методи імітації:

- порожній компонент (англ. dummy) - це об'єкти, які використовуються як заповнювачі в списку аргументів, але фактично ніколи не використовуються; на виклики до них не надаються відповіді;

- компонент-заглушка (англ. stub) - це об'єкт, який відповідає на передані йому виклики заздалегідь підготовленими відповідями, але ігнорує будь-які інші виклики, які не були попередньо налаштовані; іноді це компонент, який частково імітує поведінку реального компонента, але з обмеженим функціоналом;

- компонент-шпигун або компонент-посередник (англ. spy) - подібно до компонентів-заглушок, він частково реалізований, але відповідає на виклики до нереалізованих методів, переадресовуючи відповідальність за відповідь до вихідного компонента системи; часто використовується як посередник між викликами до вихідного компонента;

- імітований компонент або макет (англ. mock) - це компонент, який повністю імітує поведінку вихідного компонента за допомогою реалізації його інтерфейсу, але не має справжньої функціональності, а лише імітує її.

Ізолювання компонентів один від одного під час модульного тестування не лише сприяє більш точному виявленню дефектів у разі їх виникнення, а й прискорює виконання сценаріїв. Перевірка мінімальних компонентів системи в умовах, коли доступ до баз даних, файлової системи, програмного вводу-виводу або виконання запитів до зовнішніх систем обмежено, дозволяє виконати тисячі тестових сценаріїв всього за декілька хвилин [7, 9].

Цілі ізоляції можна досягнути за допомогою методу введення залежностей (dependency injection), який полягає в наданні зовнішніх залежностей програмному компоненту. Він передбачає інверсію контролю, коли фреймворк для тестування забезпечує всі потрібні залежності системним компонентам або розташовує їх у

спеціальному тестовому середовищі, де ізоляція компонентів здійснюється за допомогою пісочниць.

Ключовим елементом модульних тестів є забезпечення перевірки всіх складних сценаріїв використання компонентів системи, які включають наступні методи.

Еквівалентне розбиття - це метод “чорного ящика” для тестування, при якому діапазон значень змінної поділяється на підмножини, що мають однакове значення, а потім на основі використання одного значення з кожної підмножини розробляються тестові сценарії. Наприклад, якщо деякий компонент приймає значення в діапазоні від 1 до 10 як аргумент, то для створення тестового сценарію потрібно вибрати одне випадкове значення з допустимого діапазону (наприклад, число 7) і одне з недопустимого (наприклад, число 0).

“Аналіз граничних значень” - це методика “чорного ящика”, яка передбачає створення тестових сценаріїв, заснованих на граничних значеннях, а саме на мінімальному та максимальному значеннях відсортованого еквівалентного розбиття. Для ілюстрації, в першому підпункті, граничними значеннями будуть числа 1 та 10, які представляють мінімальне та максимальне значення діапазону, а також числа 0 та 11, які є мінімальними та максимальними граничними значеннями за межами цього діапазону.

“Причинно-наслідковий аналіз” - це методика “чорного ящика”, яка вимагає розробки тестових сценаріїв, заснованих на причинах (входах або стимулах системи) та їх відповідних наслідках (виходах або реакціях системи). Наприклад, тестовий сценарій для розгалуженого алгоритму може включати причини (специфічний набір вхідних даних для алгоритму), на основі яких алгоритм вибирає відповідну модель реакції, що є результатом цієї взаємодії.

“Передбачення помилки” - це метод тестування, який передбачає розробку тестових сценаріїв, заснованих на інформації про раніше виявлені проблеми в системі або загальних знаннях про потенційні дефекти в системах.

Вичерпне тестування - це стратегія розробки тестових сценаріїв, яка передбачає включення в тест усіх можливих комбінацій вхідних даних та передумов.

Нечітке тестування - це процес перевірки програмного забезпечення, який має на меті виявити слабкі місця в системі безпеки компонентів, шляхом подачі великої кількості випадкових даних, відомих як “нечіткі дані”.

У той же час, модульні тести не мають бути сильно залежними від конкретної реалізації компонентів системи, щоб бути гнучкими у майбутніх змінах. Якщо сценарії тестування занадто сильно спрямовані на виробничий код, то вони швидко втрачають свою актуальність при навіть незначних змінах у вихідному коді [8]. Щоб уникнути надмірної деталізації модульних тестів, слід використовувати техніки тестування методом «чорного ящика» та перевіряти лише ту поведінку системи, яку можна спостерігати без докладних знань про внутрішнє устрій системи.

Під час написання модульних сценаріїв тестування, команди звертають увагу на один з ключових показників завершеності тестування за допомогою модульних тестів - покриття вихідного коду тестовими сценаріями. Однак, часто робиться помилка, коли команди намагаються досягти 100% покриття вихідного коду проекту тестами, включаючи не тільки бізнес-логіку, а й незначний код. Хоча важливо тестувати загальнодоступний інтерфейс програми, тестування тривіального коду може бути зайвим.

Отже, у проєктах, де команди використовують підхід, що передбачає одночасне написання як вихідного коду системи, так і тестів до неї, модульне тестування служить засобом, який сприяє програмістам глибше розуміти суть проблеми та уявити граничні випадки використання функціоналу, який вони розробляють. Сценарії модульного тестування не вимагають значних зусиль для розробки, вони надають докладний опис окремих найменших компонентів системи та створюють основу для сценаріїв тестування на вищому рівні.

1.1.2 Сервісне тестування

Більшість складних систем сьогодні складаються з різноманітних підсистем та модулів, які взаємодіють між собою. Наприклад, типова система обробки даних може використовувати бази даних, файлову систему та взаємодіяти з іншими системами через мережу Інтернет. Вони ізолюються між собою для прискорення виконання тестового сценарію. Однак взаємодія системи та всіх її компонентів між собою надзвичайно важлива.

Інтеграційне тестування, також відоме як сервісне тестування, це процес, в якому перевіряються інтерфейси системи та взаємодія між інтегрованими компонентами. Оскільки розробка різних модулів або підсистем системи зазвичай виконується ізольовано та різними командами, виникає проблема подальшої інтеграції цих частин у єдину систему. Основна мета інтеграційного тестування полягає в перевірці роботи всіх модулів як одного цілого та в тому, щоб переконатися, що розроблені модулі працюють разом належним чином, а їх інтерфейси відповідають дизайну архітектури системи.

Сервісне тестування є важливим елементом в процесі розробки програмного забезпечення, який допомагає забезпечити надійність та ефективність сервісів. Це процес перевірки функціональності, продуктивності, безпеки та інших аспектів сервісів у програмному забезпеченні. Однією з ключових переваг сервісного тестування є здатність виявляти та виправляти помилки на ранніх стадіях розробки.

Сервісне тестування також допомагає забезпечити, що програмне забезпечення відповідає вимогам користувачів та бізнесу. Це може бути особливо важливо для програмного забезпечення, яке використовується в критичних для бізнесу або високонавантажених середовищах, де відмова сервісу може призвести до значних втрат.

Однак, необхідно зазначити, що сервісне тестування - це не одноразовий процес. Воно повинно бути вбудоване в життєвий цикл розробки програмного

забезпечення і проводитися регулярно для забезпечення постійної якості та надійності. Сервісне тестування є невід'ємною частиною розробки програмного забезпечення. Воно допомагає забезпечити надійність та ефективність сервісів, зменшує витрати на виправлення помилок та допомагає відповідати вимогам користувачів та бізнесу. Тому важливо включати сервісне тестування в процес розробки програмного забезпечення.

На сьогоднішній день існують два методи проведення сервісного тестування:

- Спосіб, що полягає в виклику реальних інтегрованих компонентів, що передбачає перевірку бізнес-логіки системи відповідно до поставлених вимог шляхом виконання справжніх викликів до файлової системи, баз даних або інших компонентів. Під час використання цього підходу сценарії виконання тестів можуть бути повільними, оскільки вони вимагають очікування відповіді від залежних систем. Крім того, створення сценаріїв сервісного тестування та створення середовища для їх виконання потребує значних витрат ресурсів.

- ізоляція компонент зовнішніх - виклики до баз даних або інших зовнішніх компонентів, які впливають на швидкість тестування, мають бути замінені на фальшиві компоненти, які повністю відтворюють їхню роботу, в той час як виклики до внутрішніх компонентів системи мають залишитися незмінними. Цей метод ізоляції зовнішніх компонентів значною мірою залежить від вимог до проекту та критичності часткової заміни зовнішніх компонентів.

В основному, використання другого методу призводить до більш надійних результатів тестування, оскільки при появі дефекту через ізоляцію зовнішніх інтеграцій, можна більш точно встановити джерело проблеми. В комбінації з тестуванням контрактів взаємодії, сервісні тести, створені за методом ізоляції зовнішніх компонентів, виконуються швидко і не вимагають значних зусиль для їхньої реалізації.

В сучасній архітектурі програмного забезпечення, яка зазвичай складається з рівнів абстракції, інтеграційне тестування відбувається на границях цих рівнів або на

границі самої системи. Це концептуально демонструє дії, які призводять до інтеграції програмного забезпечення з зовнішніми компонентами. Сервісні тести, як правило, допомагають виявити дефекти інтерфейсів та контрактів викликів, але вони також можуть виявити проблеми з асинхронністю, багатопоточністю, станами гонки, дефектами API системи, а головне, вони допомагають визначити, чи готова бізнес-логіка програмного забезпечення до релізу [8].

1.1.3 Тестування користувацького інтерфейсу

Сучасні програми зазвичай включають інтерфейс для користувачів: це може бути веб-інтерфейс, мобільний інтерфейс, інтерфейс командного рядка або API для взаємодії з зовнішніми службами. Тести інтерфейсу користувача, які знаходяться на вершині “традиційної піраміди тестування”, призначені для імітації дій користувача в програмному інтерфейсі (наприклад, натискання кнопок миші, введення тексту і т.д.), і програмний інтерфейс має відповідати на цю взаємодію в передбачуваний спосіб. Загалом, кількість сценаріїв тестування інтерфейсу користувача повинна бути мінімальною серед усіх сценаріїв, оскільки будь-які зміни в інтерфейсі користувача можуть зробити ці тести недійсними, а розробка таких сценаріїв тестування є важкою задачею. Тестування інтерфейсу користувацького допомагає забезпечити зручність та ефективність взаємодії користувача з продуктом. Це процес перевірки відповідності інтерфейсу вимогам користувачів та бізнесу, а також виявлення та виправлення помилок інтерфейсу.

Однією з ключових переваг тестування інтерфейсу користувацького є здатність виявляти та виправляти проблеми, які можуть впливати на задоволеність користувача. Це може включати все, від простих помилок в дизайні до більш складних проблем з навігацією та взаємодією.

Тестування інтерфейсу користувацького також допомагає забезпечити, що програмне забезпечення відповідає вимогам користувачів та бізнесу. Це може бути

особливо важливо для програмного забезпечення, яке використовується в критичних для бізнесу середовищах, де незручний або неефективний інтерфейс може призвести до втрат продуктивності або доходів.

Однак, необхідно зазначити, що тестування інтерфейсу користувацького - це не одноразовий процес. Воно повинно бути вбудоване в життєвий цикл розробки забезпечення програмного і проводитися регулярно для забезпечення постійної якості та зручності інтерфейсу.

Тестування користувацького інтерфейсу є невід'ємною частиною розробки програмного забезпечення. Воно допомагає забезпечити зручність та ефективність взаємодії користувача з продуктом, зменшує витрати на виправлення помилок інтерфейсу та допомагає відповідати вимогам користувачів та бізнесу. Тому важливо включати тестування інтерфейсу користувацького в процес розробки забезпечення програмного.

В залежності від використовуваної мови програмування, технологій та виду інтерфейсу, оптимальний сценарій тестування користувацького інтерфейсу вимагає заміни бізнес-логіки системи на “дублікат” та створення модульних тестів для абстрактного шару користувацького інтерфейсу. Однак, не завжди можливо чітко розділити шари абстракцій під час тестування, тому допустимо проводити тестування в режимі end-to-end - це тип тестування, при якому бізнес-процеси перевіряються від початку до кінця в тестовому середовищі, яке максимально наближене до виробничого. Згідно з думкою Майкла Кона, тестування користувацького інтерфейсу не відрізняється від end-to-end тестування, але в контексті сучасних можливостей фреймворків для тестування, ці поняття є досить незалежними одне від одного.

Наскрізні тести дають найбільшу впевненість у тому, що розроблений програмний продукт працює належним чином. Процес наскрізного тестування полягає у встановленні програми в тестовому середовищі, яке максимально схоже на виробниче, з усіма необхідними зовнішніми залежностями та даними. Однак,

створення сценаріїв наскрізного тестування є великою та трудомісткою задачею, а також вимагає постійного оновлення. В результаті, наскрізні тести можуть давати помилково позитивні результати або виходити з ладу з неочікуваних причин. Чим більш складним є інтерфейс користувача, тим складнішим стає розробка таких сценаріїв тестування. Введення реклами в веб-додатки, проведення промо-кампаній, додавання інших елементів або анімаційних компонентів, які можуть раптово з’явитися на веб-сторінці, ускладнює тестування інтерфейсу користувача.

Одним з найбільш викликаючих аспектів тестування користувацького інтерфейсу є встановлення, чи не відбулися зміни в графічному інтерфейсі внаслідок модифікацій у вихідному коді, які не повинні були на нього впливати. Проблема полягає в тому, що комп’ютер не може автоматично перевірити відповідність візуального відображення графічного інтерфейсу, тому це питання залишається невирішеним. Використання машинного навчання або штучного інтелекту може допомогти вирішити цю проблему до певної міри. Наразі існують інструменти, які можуть робити скріншоти екрану користувача та порівнювати їх для виявлення відмінностей. Однак, через вищезазначені проблеми, виникає виклик правильної ідентифікації очікуваних змін та налаштування чутливості до виявлення інтерфейсу, який був змінений ненавмисно.

Додатково, тестування користувацького інтерфейсу системи, що використовує мікросервісну архітектуру, може вимагати виконання та конфігурації значної кількості компонентів системи. Це може бути абсолютно нездійсненним в локальному тестовому середовищі і може вимагати використання спеціалізованих технічних рішень, таких як контейнеризація.

Через обмеження та виклики, що виникають під час тестування користувацького інтерфейсу та end-to-end тестування, кількість тестових сценаріїв на цьому рівні має бути якомога меншою. Для більшості систем достатньо провести тестування лише критичних сценаріїв використання програми. Крім того, можна провести додаткове ручне тестування з участю інженерів-тестувальників.

В таких умовах, завдяки наявності значної кількості тестових сценаріїв нижчих рівнів в “піраміді тестування”, можна з упевненістю стверджувати про повну функціональність системи. Тестування крайніх ситуацій використання системи не є необхідним [9].

1.2 Розширена модель тестування

В більшості сучасних проектів модульні тестові сценарії переважають над тестами користувацького інтерфейсу. Це обумовлено тим, що основним об’єктом перевірки є бізнес-логіка програми, а тестування інтерфейсу виступає в ролі допоміжного завдання. Проте з появою нових типів програмного забезпечення з’являються і специфічні вимоги до їх тестування. Наприклад, при тестуванні мобільних додатків акцент зміщується на користувацький інтерфейс, а тестування взаємодії з зовнішніми компонентами відіграє менш важливу роль.

Розширена модель тестування є важливим інструментом в сфері розробки програмного забезпечення, який допомагає забезпечити більш глибоке розуміння та ефективність в процесі тестування. Ця модель включає в себе не лише традиційні методи тестування, але й додає нові елементи та підходи для більш повного та ефективного тестування.

Однією з ключових переваг розширеної моделі тестування є її гнучкість. Вона дозволяє розробникам адаптувати процес тестування до конкретних вимог та обставин, що може допомогти забезпечити більш точні та ефективні результати тестування.

Розширена модель тестування також сприяє більш глибокому розумінню процесу тестування. Вона включає в себе різні методи тестування, від традиційних до новітніх, що дозволяє розробникам краще розуміти, як вони можуть використовувати тестування для підвищення якості своїх продуктів.

Розширена модель тестування вимагає більшої кваліфікації та розуміння. Розробники повинні мати глибокі знання про різні методи тестування та здатність ефективно їх застосовувати. Розширена модель тестування є важливим інструментом в процесі розробки програмного забезпечення. Вона допомагає забезпечити більш глибоке розуміння та ефективність в процесі тестування, що може призвести до створення більш якісних продуктів. Тому важливо включати розширену модель тестування в процес розробки програмного забезпечення.

“Класична піраміда тестування” представляє собою узагальнену модель автоматизованого тестування, структура якої не обов’язково має бути відтворена в повному обсязі компаніями в їхніх проектах. З розвитком методів тестування “класична піраміда” отримала розширення і може включати різні рівні тестування та їх кількість, в залежності від конкретних потреб бізнесу. На малюнку 1.2 представлена найбільш вживана “розширена піраміда тестування”. Ця методика включає в себе додатковий етап перевірки контрактів взаємодії між компонентами системи. Цей тест зосереджений на перевірці відповідності та послідовності повідомлень, які генеруються в системі під час обміну даними між її компонентами.

Рисунок 1.2 – Приклад «розширеної піраміди тестування»

“Розширена піраміда тестування” та методики її викладу перш за все зосереджуються на бізнес-цінностях та ризиках, що виникають протягом життєвого циклу розробки ПЗ. Вона не є і не може бути уніфікованою, оскільки кожен проект

потребує специфічних підходів до тестування. Замість того, щоб зосереджуватися на числі тестів, “розширена піраміда тестування” пропонує звертати увагу на тип та якість тестування для запобігання можливим ризикам відповідного бізнес-сектору.

Сучасні вимоги до тестування в контексті швидкого розвитку програмного забезпечення виявляють декілька важливих недоліків “класичної піраміди тестування”. Однак емпіричні правила, які були введені цією моделлю, залишаються важливими і сьогодні. “Класична піраміда” - це ефективна модель, на основі якої проекти можуть формувати власні стратегії тестування і, використовуючи описані методи створення сценаріїв тестування, досягати мети тестування.

1.2.1 Тестування контрактів взаємодії

Тестування контрактів взаємодії - це важливий аспект розробки програмного забезпечення, який забезпечує надійність та стабільність системи. Цей процес включає перевірку взаємодії між різними компонентами системи з метою виявлення та усунення помилок.

Контракт взаємодії - це угода між двома або більше компонентами про те, як вони будуть взаємодіяти. Він визначає, які дані будуть передаватися, які функції будуть виконуватися, та які відповіді очікувати. Контракти взаємодії можуть бути формально визначені в коді або документації.

Тестування контрактів взаємодії допомагає визначити, що всі компоненти системи правильно взаємодіють між собою. Це може допомогти виявити помилки на ранніх стадіях розробки, коли вони найлегше виправити. Крім того, це може допомогти забезпечити, що система буде працювати надійно після внесення змін або оновлень.

Тестування контрактів взаємодії зазвичай включає створення тестових сценаріїв, які перевіряють, чи правильно компоненти взаємодіють між собою. Це

може включати перевірку того, чи правильно передаються дані, чи правильно виконуються функції, та чи отримуються очікувані відповіді.

Тестування контрактів взаємодії - це важливий аспект розробки програмного забезпечення, який допомагає забезпечити надійність та стабільність системи. Це допомагає виявити та виправити помилки на ранніх стадіях розробки, а також забезпечує, що система буде працювати надійно після внесення змін або оновлень. Завдяки тестуванню контрактів взаємодії розробники можуть бути впевнені, що їх система відповідає вимогам та очікуванням користувачів.

Одним з найчастіше використовуваних сценаріїв застосування “двійників” у тестуванні є імітація зовнішніх компонентів системи, які розміщені на віддалених вузлах мережі Інтернет. Зазвичай ці компоненти розробляються окремою командою, їх продуктивність може значно відрізнятися від основної системи, а зв’язок компонентів через Інтернет може бути нестабільним, але виникає проблема точного відтворення контракту зовнішнього сервісу “двійником”.

Рішенням цієї проблеми є розробка сценаріїв для перевірки контрактів взаємодії, які систематично виконуються в рамках неперервної інтеграції та здійснюють запити до реальних зовнішніх компонентів для перевірки стабільності їх контрактів. Якщо під час виконання цих сценаріїв виникають помилки, це свідчить про потребу в оновленні сценаріїв сервісного тестування для адаптації до змін, і це повинно призупинити процес інтеграції розроблених модулів до вирішення цих проблем. Однак, цей підхід має свої недоліки: сценарії виконуються з певною періодичністю (зазвичай раз на день), тому якщо зовнішня система постійно змінюється, цей підхід може бути не достатньо ефективним.

Альтернативний метод вирішення цієї проблеми може полягати у синхронізації метаданих контрактів між системами та застосуванні шаблону “Контракт, спрямований на користувача” [11]. Так, перед сервісним тестуванням або перевіркою контрактів взаємодії, система може вимагати опис контракту від

зовнішньої системи через Інтернет і порівняти його з контрактом у сценарії тестування. Якщо контракт залишився незмінним, тести проводяться з використанням “дублікатів” у звичайному режимі. Однак, якщо контракт змінився, сценарії тестування можуть вважатися застарілими і потребувати змін, або вони можуть виконуватися з використанням реальних викликів до зовнішніх систем. Крім того, можливо провести тестування контрактів взаємодії на боці зовнішньої системи, щоб команда розробників була впевнена у цілісності інтеграції їх компоненту з іншими компонентами системи. Вибір певного підходу визначається специфічними вимогами до тестування та архітектурою програмного додатку [12].

1.2.2 Тестування продуктивності та швидкодії

Зазвичай, тестування продуктивності та швидкодії проводиться з метою оцінки ефективності, стабільності та чутливості компонента або системи до різних навантажень. Однак, тестування швидкодії також може використовуватися для аналізу інших якісних характеристик програмного додатку, таких як стійкість до відмов, масштабованість та ефективність використання ресурсів під час виконання.

Існує п'ять основних напрямків виконання тестування швидкодії:

− тестування під навантаженням – це вид тестування, який проводиться з метою оцінки здатності системи працювати стабільно під певним навантаженням (наприклад, при максимальній очікуваній кількості користувачів системи) [3];

− тестування конфігурації системи – це вид тестування, який проводиться для встановлення найкращої апаратної та програмної конфігурації системи, щоб забезпечити її стійкість до навантаження [3];

− стрес-тестування – це вид тестування, який проводиться для оцінки роботи системи або її компонентів при передбачуваних або надмірних навантаженнях, з можливістю часткового відмови компонентів системи [3].

1.2.3 Тестування безпеки

Програмні продукти, зокрема ті, що надають значущі переваги або зберігають критичні дані користувачів, часто стають ціллю для різноманітних атак. Кіберзлочинці, хакери, шахраї, а іноді навіть службовці, можуть стати загрозою для програмного додатку і можуть намагатися не лише завдати шкоди системі, але й викрасти важливу інформацію.

Тестування безпеки програмних продуктів - це процедура виявлення слабкостей у системах захисту інформації за допомогою знань про потенційні види атак та вразливості, які можуть призвести до відмови в обслуговуванні системи або до втрати контролю над інформацією. Метою розробників програмного забезпечення та інженерів з безпеки є розробка надійних механізмів захисту від можливих атак, а також запобігання атакам шляхом виявлення та усунення слабкостей системи безпеки.

Виділяють п’ять ключових областей забезпечення безпеки програмних рішень, а саме:

− мережева безпека: включає в себе пошук слабких місць мережевої інфраструктури;

− безпека системного програмного забезпечення: включає в себе пошук слабких місць в операційній системі, базах даних та іншому програмному забезпеченні, на якому базується програмний застосунок;

− захист на рівні користувацького інтерфейсу: включає в себе розробку валідації користувацького вводу на стороні клієнта або інтерфейсу програмного забезпечення.

Більша частина згаданих методів тестування може бути автоматизована за допомогою спеціалізованих інструментів тестування, таких як OWASP Zed Attack Proxy [14], а також за допомогою використання баз даних про відомі вразливості безпеки, таких як «Національна база даних вразливостей» [15]. Однак автоматизація

не гарантує абсолютної безпеки програмного застосунку, тому цей тип тестування вимагає обов’язкової участі інженерів з безпеки та тестування для забезпечення надійності системи [16].

1.2.4 Тестування локалізації

Тестування локалізації - це процес перевірки перекладів, інтернаціоналізації, орфографії та форматів тексту програмного додатку, супутньої документації, а також відповідних систем або їх компонентів. Цей вид тестування дозволяє оцінити якість адаптації програмного рішення для конкретної цільової групи з урахуванням її культурних характеристик.

Тестування автоматизації локалізації може бути реалізовано за допомогою різних підходів, але найбільш вживаним є метод, який нагадує тестування користувацького інтерфейсу. Додаток запускається в тестовому середовищі для кожної підтримуваної мови. Потім, за допомогою фреймворку для тестування, виконується або знімок екрану користувача для подальшого порівняння з попередніми версіями, або текст компонентів інтерфейсу порівнюється з вихідними даними перекладу та локалізації.

Одним з викликів при тестуванні локалізації є перевірка пристосованості користувацького інтерфейсу до мов, які пишуться справа наліво. Для таких мов елементи користувацького інтерфейсу мають бути відображені в дзеркальному порядку по горизонталі. Автоматизація тестових сценаріїв для цих випадків є складною задачею, оскільки об’єктом тестування стає зовсім інший інтерфейс. На сьогодні не існує ефективних методів для проведення такого типу автоматизованого тестування, тому локалізацію найчастіше тестують вручну [17].

1.3 Архітектура фреймворків автоматизованого тестування

Збільшення ефективності тестування, яке досягається за допомогою автоматизації, призводить до покращення якості програмного продукту і значного зниження витрат на його контроль. При використанні інструментів для автоматизації тестування, а також при розробці власних фреймворків для тестування, основна увага має бути приділена архітектурі та проектуванню програми автоматизації, оскільки наявність будь-яких помилок у такому ПЗ може свідчити про критичні вразливості в процесі тестування, що може призвести до появи ложно-позитивних результатів, зниження якості програмного забезпечення та збільшення витрат на його обслуговування.

Фреймворк для автоматизованого тестування - це платформа, яка розроблена шляхом інтеграції різноманітних апаратних і програмних ресурсів, інструментів та служб підтримки якості ПЗ, що об’єднані відповідною архітектурою. Таким чином, фреймворк тестування надає всі потрібні інструменти та керівництва для опису та організації сценаріїв тестування, їх виконання та порівняння результатів тестування з еталонами або очікуваними результатами. Крім того, він включає в свою архітектуру методи та підходи до проведення автоматизованого тестування, що допомагає новачкам розпочати створення сценаріїв тестування [18].

Основні цілі використання фреймворку для автоматизації включають:

- покращення ефективності та спрощення процесу створення сценаріїв для автоматичного тестування;
- можливість повторного використання сценаріїв для проведення регресійного тестування системи;
- впровадження структурованої методики розробки для забезпечення цілісності дизайну системи та зниження ризику випадкових дефектів;

- використання надійних інструментів для виявлення дефектів та аналізу основних причин їх появи з мінімальним людським втручанням у процес тестування;
- зменшення залежності від експертів у галузі бізнесу за допомогою автоматизованої реакції на зміни умов тестування та тестового середовища;
- ефективне використання ресурсів для тестування складних систем, а також забезпечення автоматизованого постійного контролю за процесом тестування.

Метою створення архітектури фреймворку для автоматизованого тестування є розробка системи автоматизації, яка може бути одночасно структурованою та зрозумілою, але при цьому гнучкою до змін у мовах програмування, технологіях та методах тестування. Викликом при розробці такої архітектури є вибір найкращої комбінації методів тестування, методологій та інструментів автоматизованого тестування. Створення гнучкої архітектури на початкових стадіях проектування є ключовим етапом, оскільки воно впливає на успіх подальшої розробки та визначає необхідні витрати не лише на проектування фреймворку, але й на його подальше використання в виробничому середовищі [19].

Архітектура фреймворків для автоматизованого тестування, що розглядається з позицій підходів до структуризації сценаріїв тестування, може бути умовно розділена на п’ять категорій (див. рис. 1.3).

Рисунок 1.3 – Типи фреймворків автоматизованого тестування за принципом опису сценаріїв тестування

Вибір певної архітектури залежить від вимог бізнес-домену, і саме через це серед програмного забезпечення можна зустріти широкий спектр фреймворків для автоматизованого тестування.

Фреймворк, що використовує архітектуру лінійних сценаріїв тестування, є основним інструментом для автоматизованого тестування з підходом «Запис та Відтворення». Сценарії тестування в такому фреймворку представляють собою процедурний код, який відтворює послідовність дій для виконання кроків тестування. Створення сценаріїв не вимагає спеціальних навичок програмування, а лінійність сценаріїв тестування робить їх зрозумілими.

Ці фреймворки часто застосовуються в проектах малого розміру або в тих, де можлива автоматизація створення тестових сценаріїв за допомогою методу послідовного запису етапів тестування. Фреймворк, що базується на архітектурі тестових сценаріїв, керованих даними, орієнтований на розміщення тестових даних поза межами самого тестового сценарію, що дозволяє реюзабельність сценаріїв за рахунок передачі в них різних наборів даних. В цьому контексті, датасети для тестування зберігаються в файловій системі або в базах даних.

Розділення даних та етапів тестового сценарію може істотно скоротити кількість тестових сценаріїв, а також забезпечити гнучкість для модифікації сценарію шляхом оновлення його даних. Однак, цей метод не завжди є оптимальним, оскільки система може вимагати унікальних варіантів тестування функціональності, що при цьому методі призведе до створення великої кількості тестових сценаріїв та даних для них. Іншим значним недоліком цього методу є складність розуміння процесу тестування, а також необхідність в спеціальних навичках програмування для розробки таких тестових сценаріїв.

Фреймворк, що має архітектуру тестових сценаріїв, керованих ключовими словами, спрямований на зв’язок коду системи з відповідними ключовими словами,

які використовуються в тестових сценаріях для виконання дій у системі. Як і в підході до створення сценаріїв, керованих даними, в таких сценаріях логіка сценарію, його дані та ключові слова розділяються та зберігаються на зовнішніх ресурсах. Послідовність ключових слів зазвичай записується у формі таблиці, що зіставляє події в системі та ключові слова, і описує етапи тестового сценарію. Після створення цієї таблиці, розробляється відповідний програмний код, який відповідає створеним ключовим словам, і який буде виконуватися під час виконання тестового сценарію.

Плюсом підходу до створення сценаріїв, керованих ключовими словами, є здатність до повторного використання ключових слів у різноманітних тестових сценаріях, проте мінусом є його обмежений діапазон застосування. Цей підхід найчастіше використовується для тестування користувацького інтерфейсу, але його використання для тестування може бути непридатним через високу трудомісткість. Фреймворк з архітектурою тестових сценаріїв, керованих поведінкою, є найбільш вживаним підходом, який спрямований на визначення причинно-наслідкових відносин системи.

Ціль таких фреймворків полягає в наданні універсального набору інструментів для команди розробників, інженерів з тестування та менеджменту компанії. В основному це досягається за допомогою використання зовнішніх мов для опису тестових сценаріїв, які схожі на природну мову, і які потім обробляються тестовим фреймворком для створення програмного коду кінцевих тестових сценаріїв.

Однак, головним недоліком цього підходу є необхідність в навичках програмування у інженерів з тестування, а також створення тестових сценаріїв мовою програмування.

Гібридні фреймворки для автоматизованого тестування об’єднують існуючу архітектуру та методи опису тестових сценаріїв, які найкраще відображають процеси тестування відповідного бізнес-домену.

Хоча створення таких фреймворків є важкою задачею, вони найефективніше вирішують завдання тестування. Велика гнучкість їх архітектури дозволяє настроювати фреймворк під специфічні вимоги, вибирати інструменти та методи тестування, а також розширювати його функціональність за допомогою системи додаткових модулів.

Особливістю всіх фреймворків автоматизованого тестування є процес, за яким вони працюють: сценарії тестування витягуються з визначеного ресурсу, кроки сценаріїв виконуються в заданій послідовності з використанням додаткових даних, а сам фреймворк керує програмним додатком і перевіряє результати виконання тестових сценаріїв.

Загальна архітектура фреймворку автоматизованого тестування представлена на рисунку 1.4.

Рисунок 1.4 – Загальна архітектура фреймворку автоматизованого тестування

Отже, завдяки гнучкій архітектурі, більшість видів фреймворків автоматизованого тестування можуть адаптуватися до змін у проєктах з метою зменшення ручного тестування та автоматизації регресійного тестування. Застосуванням різних методів тестування та автоматичним аналізом результатів

можна виявити первинну причину дефекту та надати інформацію для його виправлення [20].

Висновки до розділу 1

Локалізаційне тестування - це процес перевірки якості перекладів, інтернаціоналізації, орфографії та форматування тексту програмного додатку, супутньої документації, а також відповідних систем або їх складових.

Цей вид тестування допомагає оцінити, наскільки добре програмний продукт адаптований для конкретної цільової аудиторії, враховуючи її культурні особливості. Автоматизоване перевіряння локалізації може бути виконане за допомогою різноманітних технік, проте найчастіше використовується метод, що має багато спільного з тестуванням користувацького інтерфейсу. Програмний продукт стартує в тестовому середовищі для кожної з підтримуваних мов.

Після цього, використовуючи фреймворк для тестування, робиться або знімок екрану користувача, який потім перевіряється на відповідність попереднім версіям, або текст компонентів інтерфейсу порівнюється з вихідними даними перекладів та локалізації.

Модульне тестування допомагає виявити та виправити помилки на ранніх стадіях розробки, коли вони найлегше виправити. По-друге, воно допомагає забезпечити, що кожний модуль працює правильно перед тим, як він інтегрується з рештою системи. Це може допомогти запобігти проблемам, які можуть виникнути під час інтеграції. Модульне тестування зазвичай включає створення та виконання тестових сценаріїв для кожного модуля. Ці сценарії мають перевірити, чи правильно модуль виконує свої функції при різних вхідних даних. Це може включати перевірку відповідей модуля на коректні вхідні дані, а також на некоректні або неочікувані вхідні дані. Модульне тестування орієнтоване на найменші функціональні частини програми, зазвичай окремі функції або методи. Кожен модуль тестується ізольовано від інших, щоб перевірити, чи працює він правильно в різних ситуаціях. Для цього

створюються тестові сценарії, які містять вхідні дані та очікувані результати. Якщо фактичний результат співпадає з очікуваним, тест вважається успішним.

Модульне тестування становить фундамент “піраміди тестування” і забезпечує впевненість, що найдрібніші елементи програмного забезпечення, такі як функції, методи та класи, функціонують відповідно до встановлених вимог. Фреймворки для автоматизованого тестування є невід’ємною складовою будь-якого сучасного проекту. Покращення ефективності тестування, яке досягається за допомогою автоматизації, призводить до підвищення якості програмного продукту і значно зменшує витрати на його моніторинг.

Отже, завдяки гнучкій архітектурі, більшість фреймворків для автоматизованого тестування адаптуються до змін у проєктах з метою зменшення ручного тестування та автоматизації регресійного тестування. Застосуванням різних методів тестування та автоматичним аналізом результатів можна виявити основну причину дефекту та надати інформацію для його виправлення.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ЗАСОБУ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

2.1Аналіз вимог до програмного забезпечення

Оскільки основний елемент фреймворку для автоматизованого тестування - це формальна мова, створена для опису сценаріїв тестування, то головною метою при створенні фреймворку є програмна реалізація цієї мови. Це досягається шляхом конвертації інструкцій створеної мови на інструкції іншої мови програмування. Для вирішення цієї задачі було обрано мову програмування JavaScript і програмну платформу Node.js, яка є середовищем JavaScript, заснованим на двигуні JavaScript Chrome V8 [18].

Згідно з індексом GitHub [25], JavaScript визнана найбільш використовуваною мовою програмування для проєктів з відкритим кодом у світі, що підтверджує актуальність вибору цієї мови.

Виходячи з результатів аналізу архітектури фреймворків автоматизованого тестування, а також характеристик існуючих інструментів тестування для програмної платформи Node.js, фреймворк для автоматизованого тестування має відповідати таким вимогам:

- надійність (фреймворк має гарантувати, що результати тестування точно відображають стан системи на час проведення тестування);
- переносимість. Фреймворк має бути здатним без проблем встановлюватися в різних технологічних стеках та середовищах;
- масштабованість. У разі потреби розширення проекту, фреймворк має надавати можливість модифікації та переструктуризації сценаріїв тестування та їх ресурсів за допомогою впорядкованої структуризації;

− повторюваність. Фреймворк має забезпечувати можливість многоразового запуску групи сценаріїв тестування без потреби вносити зміни або проводити додаткові налаштування;

− прозорість. Всі результати тестування та активність фреймворку мають бути чітко відображені в лог-файлах, доступних для користувачів, а також на пристроях системного вводу-виводу;

− версіонування. Фреймворк має вести облік тестових сценаріїв, всієї супутньої інформації, а також результатів тестування;

− гнучкість. Фреймворк має бути гнучкою тестовою платформою, яка підтримує додавання нових функціональних можливостей, включаючи мови опису тестових сценаріїв, за допомогою створення внешних програмних додатків;

− сучасність. Фреймворк для тестування має використовувати сучасні можливості технологій та мов програмування.

Необов’язково але бажано, фреймворк автоматизованого тестування може відповідати наступним характеристикам:

- Віддалене виконання. Фреймворк має надавати можливість формування та завантаження завдань, які можуть бути виконані згодом або за командою від зовнішнього джерела;
- Відстеження історії виконання. Фреймворк має зберігати історію виконання тестових сценаріїв; у випадку невдалого запуску сценарію, при наступному запуску такому сценарію повинен бути надано пріоритет;
- Відмовостійкість. У випадку критичних збоїв у роботі фреймворку, він повинен автоматично виявляти причину збою, визначати спосіб відновлення та реагувати відповідним чином без додаткового втручання;
- Різноманітність. Фреймворк має надавати доступ до повного набору інструментів для автоматизованого тестування, методів та методологій тестування;
- Покриття. Фреймворк має надавати статистику покриття коду системи тестовими сценаріями у результаті тестування, якщо це необхідно. [21].

2.2 Архітектура програмного забезпечення

Враховуючи, що в проектах середнього розміру на Node.js кількість сценаріїв автоматизованого тестування зазвичай коливається від 50 до 200, виконання кожного тестового сценарію та перевірка його результатів мають бути операціями, що виконуються швидко і ефективно використовують системні ресурси.

Node.js - це програмна платформа, яка працює в однопотоковому режимі, де блокуючі операції, такі як читання та запис файлів, робота з базою даних і т.д., виконуються асинхронно. Код програми на Node.js обробляється в так званому “циклі подій”, а також надає автоматичний пул потоків для виконання завдань, які можуть вимагати значного процесорного часу. Однак, оскільки пул потоків має обмеження, програми, написані для Node.js, повинні обережно використовувати його. Основним принципом розробки для цієї платформи є те, що “Node.js забезпечує максимальну продуктивність, за умови, що робота, пов’язана з кожним клієнтом, мінімізована в будь-який момент часу” [26].

Отже, для забезпечення високої продуктивності програми та скорочення часу виконання тестового сценарію, ключовим вимогам до платформи Node.js є зменшення складності та кількості операцій. Використання метапрограмування для реалізації синтаксису формальної мови дозволяє ігнорувати парсинг формальної мови. Однак, для перекладу інструкцій формальної мови в команди JavaScript, потрібно створити спрощене абстрактне синтаксичне дерево, яке описує структуру та елементи тестового сценарію, але ця задача не є складною з точки зору алгоритміки. Інші процеси роботи автоматизованого тестувального фреймворку не вимагають значних ресурсів, тому максимальну продуктивність можна досягти шляхом рівномірного розподілу завдань між доступними потоками платформи Node.js.

На основі результатів аналізу вимог до програмного забезпечення та потенційних архітектурних рішень для інструментів автоматизованого тестування,

була створена архітектура програмного додатку, яка представлена на діаграмі компонентів (рисунок 3.1).

Рисунок 3.1 – Архітектура застосунку для автоматизації тестування

Архітектурний підхід включає такі компоненти:

- ядро фреймворку: забезпечує взаємодію всіх компонентів архітектури, реалізацію формальної мови для опису тестових сценаріїв, а також служить входом та виходом системи;
- командний рядок: інструменти для організації програмного вводу та виводу;
- управління залежностями: компонент для контролю віртуальних машин та інструмент для впровадження залежностей для тестового програмного застосунку;
- виконавець тестових сценаріїв: компонент для пошуку тестових сценаріїв, їх пріоритизації, організації в чергу та паралельного виконання;
- програмний контекст: бібліотека змінних, констант, посилань, яка є програмним середовищем певного компонента [22].

2.3 Реалізація функціоналу програмного забезпечення

Створення вбудованої предметно-орієнтованої мови в JavaScript є процесом реалізації формальної мови. В цьому контексті, кожне ключове слово формальної мови стає функціональним об’єктом в JavaScript. Наприклад, ключове слово “scenario”

виступає як позначений шаблонний літерал, де “scenario” є позначкою, що вказує на створення об’єкта сценарію, а шаблонний літерал представляє собою рядок, що визначає назву сценарію. Позначка шаблону, будучи функціональним об’єктом, через замикання контексту функції повертає інший функціональний об’єкт в контекст, який встановлює межі сценарію тестування і приймає додаткові поля сценарію тестування як аргументи.

Використання ключових слів “before” та “after”, а також інших ключових слів, що описують лінійну послідовність дій, втілюється через застосування монад. Монади - це функціональні об’єкти зі станом, які зберігають лінійну послідовність взаємопов’язаних команд.

Ключові слова, такі як “steps” та “test”, представлені через функціональну композицію, яка відображає ієрархічну структуру полів сценарію. Ця функціональна композиція розпочинається з ключового слова формальної мови, за яким у круглих дужках подаються аргументи - це поля сценарію, які мають ієрархічні зв’язки. Таким чином, формується структура сценарію, її компоненти поділяються на розділи відповідно до принципу виконання, який визначається стандартом тестування.

З погляду синтаксису JavaScript та семантичного представлення програмного сценарію, синтаксис формальної мови представляє собою набір ключових слів, які розташовані в глобальних змінних вихідного коду сценарію тестування. Однак, якщо б ці ключові слова були розташовані в глобальних змінних, то їх контекст міг би розповсюджуватися на користувацький програмний застосунок, що могло б мати вплив на його роботу і в крайньому випадку могло б призвести до відмови виконання [23].

Щоб вирішити цю проблему, було використано віртуальну машину JavaScript-двигуна Chrome V8, яку надає програмна платформа Node.js. Ця віртуальна машина дозволяє виконувати код програми в ізольованому середовищі (так званих

“пісочницях”) і точково керувати контекстом виконання програми, який не розповсюджується на інші сценарії, процеси та потоки без явного дозволу.

Створена формальна мова також передбачає можливість визначення змінних для використання в рамках тестового сценарію. У JavaScript змінні визначаються за допомогою ключових слів “let” або “var”, а константи - за допомогою ключового слова “const”.

Після цих ключових слів слідує ім’я змінної або константи, а потім через знак “дорівнює” встановлюється бажане значення змінної або константи. Проте, через те, що синтаксис створеної формальної мови дозволяє визначати змінні без використання будь-яких ключових слів перед їхнім оголошенням, виникли труднощі з реалізацією такого синтаксису в рамках мови програмування.

Програмний контекст віртуальної машини представляє собою об’єкт, який може змінюватися під час виконання програмного сценарію. Ці зміни можуть включати редагування змінних, перевизначення або створення нових глобальних змінних, створення констант і т.д.

У цьому випадку, для вирішення задачі, потрібно обернути програмний контекст віртуальної машини в проксі-об’єкт і створити такі перехоплювачі:

− для отримання змінної - проксі-об’єкт за допомогою рефлексії перевіряє, чи була змінна створена. Якщо змінна була створена раніше, він повертає її значення;

− для створення змінної - проксі-об’єкт отримує назву змінної та її бажане значення і за допомогою рефлексії створює цю змінну в програмному контексті віртуальної машини.

В результаті того, що проксі-об’єкт відслідковує процеси створення та доступу до глобальних змінних у програмному контексті віртуальної машини, ми можемо реалізувати синтаксис декларації змінних для новоствореної формальної мови без потреби використовувати ключові слова JavaScript [24].

Виконання вихідного коду сценарію тестування призводить до створення спрощеного абстрактного синтаксичного дерева. Це дерево представляє структуру

сценарію як набір вкладених об'єктів та масивів даних. Діаграма послідовності підготовки програмного середовища та виконання сценарію тестування фреймворком представлена на рисунку 3.2.

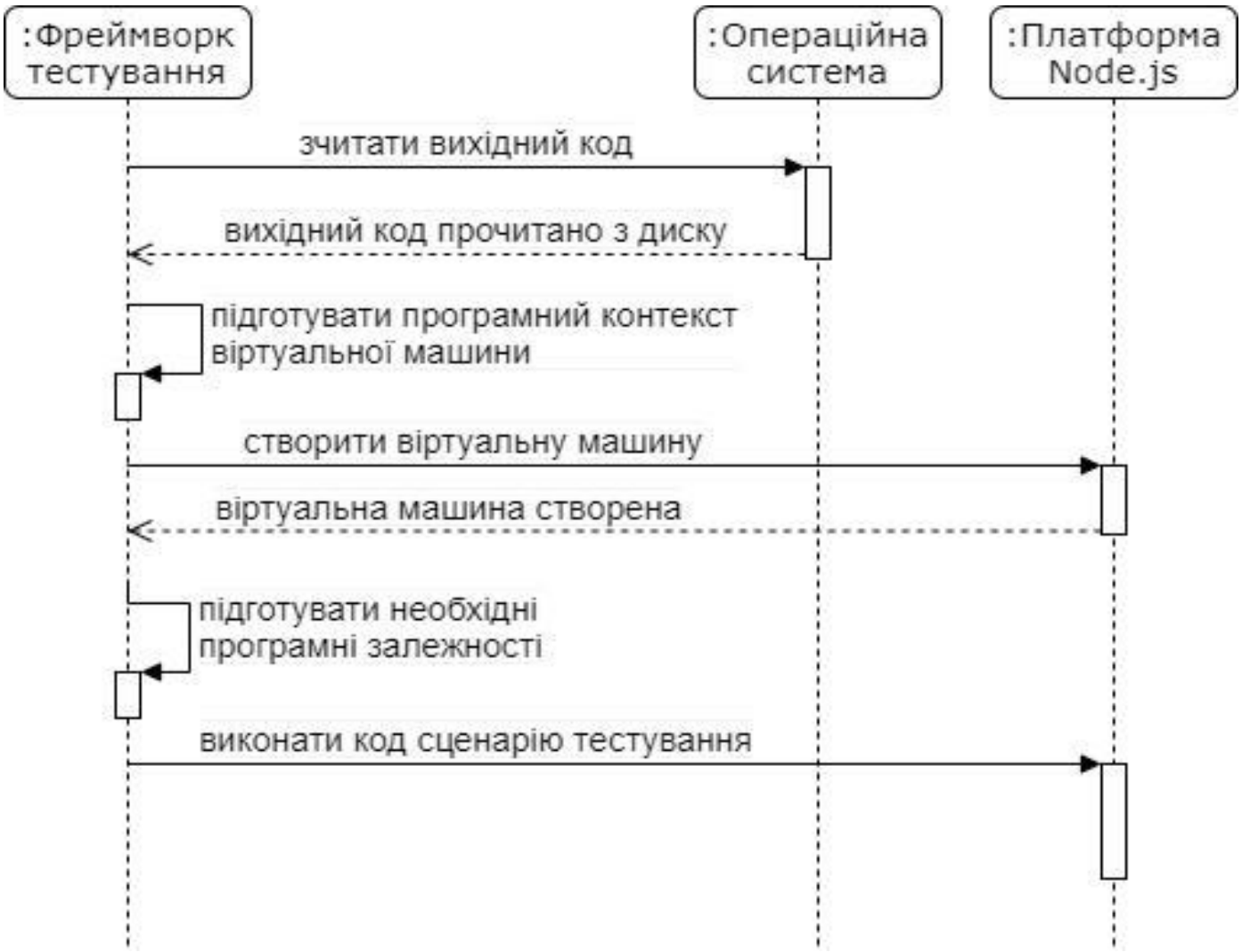


Рисунок 3.2 – Послідовність виконання вихідного коду сценарію тестування

Якщо таке посилання не було надано в тестовому сценарії, сценарій визнається недійсним, і фреймворк переходить до обробки наступного тестового сценарію. Якщо посилання було успішно отримано, фреймворк ініціює процес створення віртуальної машини, який аналогічний до процесу, описаного вище для віртуальної машини обробки тестового сценарію.

Основна відмінність між створенням віртуальної машини для клієнтського програмного застосунку та віртуальної машини для тестового сценарію полягає в процесах створення програмного контексту та підготовки залежностей.

Основою модульного тестування є створення “дублікатів” компонентів системи користувача. Ці симульовані компоненти повинні бути вже готові до моменту запуску коду користувача.

Для цього, фреймворк тестування використовує метадані про “дублікати”, які він отримує зі спрощеного абстрактного синтаксичного дерева. Використовуючи ці метадані, створюються відповідні об’єкти, які потім розміщуються безпосередньо в програмному контексті віртуальної машини застосунку користувача або в реєстрі програмних залежностей, який передається програмному застосунку користувача перед запуском його вихідного коду.

Отже, відмінно від існуючих фреймворків автоматизованого тестування на програмній платформі Node.js, розроблений фреймворк виконує заміну залежностей компонентів системи користувача на стадії підготовки та налаштування системи, а не під час її роботи, оновлюючи посилання, що є ненадійним методом ізоляції залежностей.

Останнім кроком у роботі фреймворку тестування є відновлення та відмова від використаних системних ресурсів, завершення роботи програми користувача та завершення роботи самого фреймворку тестування. Завершення роботи фреймворку тестування передбачає передачу контролю назад до операційної системи користувача та відправку сигналу про статус виконання тестових сценаріїв.

Якщо не вдалося виконати хоча б один сценарій або виникла будь-яка непередбачена помилка, то процес завершується з кодом помилки «1». Якщо ж всі сценарії були успішно виконані, процес завершується з кодом «0», що свідчить про те, що фреймворк успішно виконав всі заплановані завдання і досягнув мети тестування [25].

Висновки до розділу 2

У цьому розділі наведено архітектурний підхід та реалізацію програмного забезпечення, що відповідає функціональним та нефункціональним вимогам, викладеним у розділі 3.1.

Створене програмне забезпечення надає можливість проводити автоматизоване модульне тестування програмного забезпечення за допомогою сценаріїв тестування, розроблених на мові програмування, описаній у 2 розділі даної роботи.

Фреймворк для тестування, який ми розробили, відрізняється від інших рішень, доступних на програмній платформі Node.js. Він виконує тестові сценарії та вихідний код клієнтського додатку в окремих ізольованих тестових середовищах. Це не тільки дозволяє контролювати процес тестування за допомогою інверсії управління, але й надійно ізолює компоненти системи, знижуючи їх взаємний вплив.

Додатково, наш фреймворк надає додаткові інструменти для модульного тестування, включаючи модуль для створення фіктивних даних та компонентів, модуль для звітування про результати тестування та інше.

РОЗДІЛ 3 РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

3.1 Мета та порядок досліджень

Для оцінки ефективності, стислості та виразності запропонованого методу опису сценаріїв автоматизованого тестування, а також швидкості відповідного програмного додатку для проведення автоматизованого тестування, пропонується провести ряд експериментальних досліджень у порівнянні з існуючими методами та інструментами автоматизації тестування, а саме:

- порівняння стислості синтаксису сценарію модульного тестування, написаного на розробленій формальній мові, з текстом опису сценарію тестування народною мовою за допомогою обчислення відстані Левенштейна;
- обчислення теоретичної та практичної виразності сценаріїв тестування, описаних на запропонованій формальній мові.
- аналіз сценарію модульного тестування, створеного на розробленій нами формальній мові, у порівнянні з подібними сценаріями тестування, описаними за допомогою існуючих фреймворків автоматизованого тестування на програмній платформі Node.js, за числом ключових слів і символів, що застосовують для їх опису;
- вимірювання швидкості роботи розробленого нами програмного додатку для виконання сценаріїв автоматизованого тестування у порівнянні з існуючими інструментами автоматизації тестування на програмній платформі Node.js.

Експериментальні дослідження виконуються за допомогою апаратного забезпечення, яке має такі специфікації:

Центральний процесор: Intel® Core™ i7-7500U, двоядерний, з основною частотою 2.90 ГГц;

ОЗП: 16 ГБ SODIMM з частотою 2133 МГц;

Відеокарта: NVIDIA GeForce GTX 960M з 4 ГБ відеопам'яті;

ОС: 64-бітна версія Windows 10 Home [26].

3.2 Порівняння шляхом розрахунку відстані Левенштейна

Щоб порівняти лаконічність запропонованої формальної мови для опису сценаріїв тестування з лаконічністю мови людського опису, пропонується використати розрахунок відстані Левенштейна. Це міра різниці між послідовностями символів, яку запропонував радянський математик Володимир Левенштейн [27]. Вона визначається як найменша кількість односимвольних операцій вставки, видалення або заміни, необхідних для перетворення однієї послідовності символів на іншу. Менше число необхідних операцій свідчить про більшу схожість двох послідовностей символів. Таким чином, якщо сценарій автоматизованого тестування, створений за допомогою запропонованої формальної мови, має невелику відстань Левенштейна в порівнянні з аналогічним описом сценарію тестування на людській мові, то можна стверджувати, що запропонована формальна мова має лаконічність, що наближена до людської мови.

Сценарій модульного тестування функції сумування чисел був створений для проведення експериментального дослідження за допомогою запропонованої формальної мови (див. рис. 4.1). Також було розроблено відповідний сценарій модульного тестування (6), який був описаний на людській мові, використовуючи латинську абетку:

Unit test scenario "Sum two numbers" should call function "sum" from source file "./sum.js" with arguments 3 and 4 to test if correct answer 7 is returned as a result (6) of the call.

```
scenario `Sum two numbers` (
  sourceFile `./sum.js`,
  entity `sum`,
  cases (
    test `if correct answer 7 is returned` (
      steps (call `entity`, 3, 4)
            (expect `entity`, { returns: 7 })
    ),
  )
)
```

Рисунок 4.1 – Сценарій модульного тестування функції додавання на запропонованій формальній мові

Для розрахунку відстані Левенштейна застосовується метод матричного розрахунку з розміром матриці:

де n та m – довжини порівнюваних рядків A та B відповідно. Для заповнення матриці застосовуються наступні формули:

$$D(i,j) = \begin{cases} 0, & \text{якщо } i = 0, j = 0 \\ i, & \text{якщо } j = 0, i > 0 \\ j, & \text{якщо } i = 0, j > 0 \\ \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+1 \end{cases}, & \text{якщо } i > 0, j > 0. \end{cases}$$

Отже, Левенштейнова відстань між описом сценарію тестування, виконаного на запропонованій формальній мові, та описом на людській мові становить 124 односимвольних операції. На рисунку 4.2 для порівняння представлено сценарій модульного тестування, розроблений за допомогою фреймворку Jest, Левенштейнова відстань якого від сценарію на людській мові становить 128.

```
1 const sum = require('./sum.js');
2
3 describe('Sum two numbers', () => {
4   test('if correct answer 7 is returned', () => {
5     expect(sum(3, 4)).toBe(7);
6   });
7 });
```

Рисунок 4.2 – Сценарій модульного тестування функції додавання із застосуванням фреймворку Jest

Для детальнішого аналізу лаконічності синтаксису тестових сценаріїв пропонується вивчити більш складний приклад модульного тестування: створення масиву, який наповнений результатами виконання визначеної функції (див. рисунок 4.3).

```
function generateArray(length, generateFunction) {
  return Array.from({ length }, generateFunction);
}
```

Рисунок 4.3 – Вихідний код функції генерації масивів

Модульне тестування вказаної функції вимагає використання мок-функції як аргументу під час тестування та перевірки, що виконання цієї функції дає масив визначеної довжини, заповнений результатами виклику мок-функції. Сценарій модульного тестування цієї функції, описаний за допомогою запропонованої формальної мови, представлено на рисунку 4.4, а відповідний сценарій тестування, розроблений з використанням фреймворку Jest, - на рисунку 4.5. Текстовий опис подібного сценарію модульного тестування (7) може бути представлений на людській мові, використовуючи латинську абетку:

Unit test scenario "Generate prefilled array" should call function "generateFunction" from source file "./array.js" with 2 arguments: number 10 and a fake function that (7) always returns number 1. Function "generateFunction" should return an array of size 10 that has each element equal number 1 as a result of the call.

```
scenario `Generate prefilled array` (
  sourceFile `./array.js`,
  entity `generateFunction`,
  doubles (
    fakeFunction `fakeGenerateFn` ({
      implementation: () ⇒ 1
    })
  ),
  cases (
    test `function should return array` (
      steps (call `entity`, 10, fakeGenerateFn)
            (expect `entity`, {
              returns: {
                type: Array,
                length: 10,
                items: { equal: 1 }
              }
            })
    )
  )
)
```

Рисунок 4.4 – Сценарій модульного тестування функції "generateFunction" на запропонованій формальній мові

```
const generateFunction = require('./array.js');
const fakeGenerateFn = jest.fn(() => 1);

describe('Generate prefilled array', () => {
  test('function should return array', () => {
    const result = generateFunction(10, fakeGenerateFn);
    expect(result).toBeInstanceOf(Array);
    expect(result).toHaveLength(10);
    result.forEach(item => expect(item).toStrictEqual(1));
  });
});
```

Рисунок 4.5 – Сценарій модульного тестування функції "generateFunction" із застосуванням фреймворку Jest

Обчислення відстані Левенштейна між описом сценарію тестування, виконаного на запропонованій формальній мові, та описом на людській мові вказує на необхідність 232 односимвольних операцій для перетворення однієї послідовності символів на іншу. У той час як для сценарію тестування, описаного за допомогою фреймворку Jest, потрібно 280 таких операцій.

Отже, на основі експериментальних досліджень, синтаксис нашої формальної мови виявився більш стислим, ніж сценарій тестування, створений за допомогою Jest. Враховуючи, що синтаксис тестових сценаріїв, описаних у цій роботі для фреймворків автоматизованого тестування, є подібним або ідентичним, можна стверджувати, що наша предметно-орієнтована мова програмування краще підходить для опису сценаріїв автоматизованого тестування, ніж існуючі аналоги [27].

3.3 Порівняння шляхом підрахунку слів та символів

Одним з ключових аспектів стислості та виразності нашої формальної мови є кількість слів та символів, які потрібні для опису тестового сценарію. Наприклад, ми розглядаємо вищенаведені тестові сценарії, а в таблиці 4.1 ми надаємо обрахунок необхідної кількості слів та символів для опису відповідних сценаріїв модульного тестування. В таблиці 4.2 ми показуємо кількість допоміжних символів (перенесення

рядка та пробіли), які використовуються для опису цих сценаріїв автоматизованого тестування, а в таблиці 4.3 ми надаємо кінцевий обрахунок необхідних слів та символів без урахування допоміжних символів [28].

Таблиця 4.1 – Підрахунок кількості необхідних слів та символів враховуючи символи перенесення рядка та пробіли

	Кількість необхідних слів			Кількість необхідних символів		
	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest
Сценарій (6)	33	33	23	179	207	154
Сценарій (7)	50	52	35	317	457	375

Таблиця 4.2 – Кількість допоміжних символів для опису наведених сценаріїв автоматизованого тестування

	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest
Сценарій (6)	32	57	25
Сценарій (7)	49	171	45

Таблиця 4.3 – Результати аналізу кількості необхідних слів та символів для опису сценаріїв тестування різними методами

	Кількість необхідних слів			Кількість необхідних символів		
	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest
Сценарій (6)	33	33	23	179	150	129
Сценарій (7)	50	52	35	317	286	330

Враховуючи, що пробіли та перенесення рядків мають семантичне та синтаксичне значення лише для людської мови, можна знехтувати їх кількістю при використанні запропонованої формальної мови, а також мови опису сценаріїв тестування за допомогою Jest. Незважаючи на те, що формальна мова опису сценаріїв тестування, яку ми пропонуємо, в більшості випадків вимагає більше слів та символів, ніж аналогічні сценарії, описані за допомогою Jest, вона є найближчою до людської мови опису [29].

3.4 Порівняння швидкодії фреймворків тестування

Для оцінки продуктивності розробленого програмного додатку та його порівняння з існуючими аналогами на платформі Node.js, ми використали програму GNU time версії 1.9 [28] з відкритим вихідним кодом. Для проведення дослідження ми використали сценарії модульного тестування з розділу 4.2 (див. рисунки 4.1, 4.2, 4.4 та 4.5), які були запущені на фреймворках в паралельному режимі. Результати експерименту представлені у вигляді діаграми на рисунку 4.6.

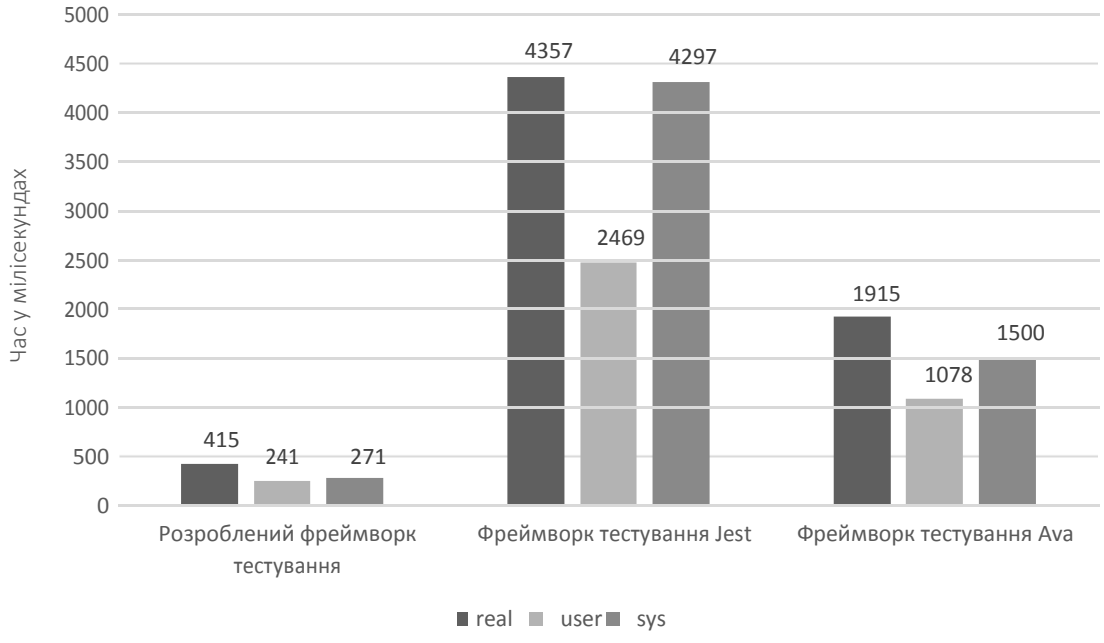


Рисунок 4.6 – Результати порівняння швидкодії програмних застосунків

Результати порівняння включають три компоненти:

- real time – це час, який програма потребує від моменту запуску до завершення;
- user time – це час, який програма проводить на центральному процесорі;
- sys time – це час очікування на виконання завдань операційною системою.

На рисунку 4.6 видно, що продуктивність розробленого програмного продукту значно перевищує Jest та Ava, фреймворки для автоматизованого тестування. Отже, можна стверджувати, що створений програмний продукт є більш ефективним рішенням для автоматизованого модульного тестування, ніж існуючі аналоги [30].

Висновки до розділу 3

У цьому розділі було проведено декілька експериментальних тестів характеристик запропонованого методу для створення сценаріїв автоматизованого тестування та інструменту для їх впровадження.

На основі результатів цих експериментальних досліджень можна стверджувати, що на даний момент запропонована предметно-орієнтована мова програмування є найбільш схожою на звичайну розмовну мову в порівнянні з іншими аналогами і найкраще підходить для розв’язання задачі створення сценаріїв автоматизованого тестування.

У порівнянні з існуючими аналогами, розроблена формальна мова не вимагає спеціальних навичок програмування для створення сценаріїв автоматизованого тестування. Вона проста, лаконічна, має виразний синтаксис, який легко читається і інтуїтивно зрозумілий при написанні. Створений програмний продукт для виконання сценаріїв автоматизованого тестування відрізняється високою швидкістю роботи на платформі Node.js, що в середньому в 10 разів перевищує швидкість роботи аналогів.

ВИСНОВКИ

У цій дипломній роботі було створено методики та інструменти для автоматизованого тестування програмного забезпечення. Вони покращують ефективність процесу автоматизації тестування за допомогою використання предметно-орієнтованої мови для опису сценаріїв тестування. Ця мова відрізняється своєю лаконічністю та виразністю, а також схожістю зі звичайною розмовною мовою.

У ході виконання роботи були вивчені існуючі методики опису сценаріїв автоматизованого тестування. Встановлено, що велика кількість з них передбачає формулювання сценарію тестування безпосередньо на мові програмування, якою написано програмне забезпечення, що підлягає тестуванню. Це ставить перед тестувальниками вимогу володіння конкретною мовою програмування та навичками програмування на ній.

Для вирішення цієї проблеми було запропоновано вдосконалений підхід до створення сценаріїв автоматизованого модульного тестування, який включає:

− організацію сценарію тестування у вигляді документу з ієрархічною структурою полів, які відображають етапи тестування відповідно до стандартів тестування;

− використання спеціальної предметно-орієнтованої мови для створення такого сценарію, яка максимально наближена до розмовної мови, але має словниковий запас базової мови програмування.

У межах впровадження цього методу була створена предметно-орієнтована мова для створення структурованого сценарію тестування, поля і тестові випадки якого представлені металінгвістичними абстракціями, визначеними за допомогою словникового запасу базової мови програмування. Ця формальна мова подібна до звичайної розмовної мови, але залишається мовою програмування в контексті базової мови.

Для виконання тестового сценарію, створеного на розробленій предметно-орієнтованій мові, було запропоновано метод трансляції. Замість звичайних етапів, цей метод передбачає перетворення металінгвістичних абстракцій, які використовуються для визначення полів та тестових випадків тестового сценарію, в ієрархію пов’язаних об’єктів та масивів за допомогою методів метапрограмування.

Для виконання запропонованих рішень було створено фреймворк, що автоматизує модульне тестування для програмної платформи Node.js. Цей фреймворк дозволяє реалізувати сценарії тестування програмного забезпечення на JavaScript і відрізняється високою швидкістю тестування порівняно з існуючими аналогами.

Щоб забезпечити можливість модульного тестування, було запропоновано архітектурний підхід до ізоляції компонентів програмної системи під час тестування. Суть цього підходу полягає у використанні віртуальної машини JavaScript-рушія Chrome V8 та спеціально сконфігурованого контексту виконання для ізоляції компонентів системи. Цей підхід дозволяє замінювати компоненти системи до початку виконання програмного коду без зміни вихідного коду системи або оновлення глобального контексту виконання. У порівнянні з існуючими підходами в фреймворках автоматизованого тестування програмної платформи Node.js, це рішення не впливає на роботу програмного додатку та на результати тестування, оскільки кожний модуль програмного додатку виконується в окремому незалежному середовищі віртуальної машини.

У процесі роботи було виконано аналіз характеристик запропонованих рішень. Результати експериментів свідчать, що ефективність використання розроблених методів та інструментів для автоматизованого тестування відрізняється високою швидкістю у порівнянні з існуючими методами та інструментами на програмній платформі Node.js. Запропонована предметно-орієнтована мова, хоча і не має аналогів, дозволяє більш стисло формулювати сценарії тестування.

Наукова новизна отриманих результатів відображається у вдосконаленні методики формування сценаріїв автоматизованого тестування через структуризацію

такого сценарію та представлення окремих полів сценарію та тестових випадків у форматі модифікованих S-виразів. Для створення такого сценарію було розроблено спеціалізовану предметно-орієнтовану мову програмування, яка має схожість зі звичайною розмовною мовою, тому є простим та зрозумілим інструментом для формування сценаріїв автоматизованого тестування.

Практична цінність полягає у розробці фреймворку для модульного тестування на програмній платформі Node.js, який дозволяє проводити тестування за допомогою запропонованої предметно-орієнтованої мови і відрізняється високою швидкістю порівняно з існуючими аналогами. Таким образом, всі завдання бакалаврської роботи були виконані в повному обсязі, а мета була досягнута. Розроблені методи та інструменти для автоматизованого тестування є зручними засобами для впровадження модульного тестування на програмній платформі Node.js.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Abelson H. Structure and Interpretation of Computer Programs / H. Abelson, G. Sussman., 2019. – 883 с. – (Second edition).

2. An all-in-one test automation solution [Електронний ресурс] – Режим доступу до ресурсу: https://www.katalon.com/

3. Automation Testing Tutorial: What is Automated Testing. [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/automation-testing.html

4. Beizer B. Software Testing Techniques / Boris Beizer., 2019. – 580 с. – (2n Edition).

5. Brooks F. The Mythical Man-Month: Essays on Software Engineering / Frederick Brooks., 2019. – 336 с. – (Anniversary edition; 2).

6. Burns D. Selenium 2 Testing Tools: Beginner's Guide / Burns D. – Birmingham: Packt Publishing, 2019. – 437 с.

7. Chomsky N. Syntactic Structures / Noam Chomsky. – Berlin, New York: Mouton de Chomsky N. Aspects of the Theory of Syntax / Noam Chomsky., 2019. – 251 с.

8. Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects [Електронний ресурс] – Режим доступу до ресурсу: http://tisten.ir/wp-content/uploads/2019/01/Complete-Guide-to-TestAutomation-Techniques-Practices-and-Patterns-for-Building-andMaintaining-Effective-Software-Projects-Apress-2018-Arnon-Axelrod.pdf

9. Don't Block the Event Loop (or the Worker Pool) [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/en/docs/guides/dont-block-the-event-loop/.

10. ECMAScript 2021 Language Specification [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://tc39.es/ecma262/.

11. Fields J. Working Effectively with Unit Tests // Jay Fields. Leanpub. 2019. – 347 с.

12. Fowler M. Inversion of Control Containers and the Dependency Injection pattern [Електронний ресурс] / Martin Fowler. – 2019. – Режим доступу до ресурсу: https://www.martinfowler.com/articles/injection.html.

13. Fowler M. Integration Testing [Електронний ресурс] / Martin Fowler. – 2019. – Режим доступу до ресурсу: https://martinfowler.com/bliki/IntegrationTest.html.

14. GNU Time [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.gnu.org/software/time/.

15. ISO/IEC 14977:1996 (Information technology — Syntactic metalanguage — Extended BNF) [Електронний ресурс] – Режим доступу до ресурсу: https://www.iso.org/standard/26153.html.

16. McCarthy J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I [Електронний ресурс] / John McCarthy // MIT Press. – 2019. - Режим доступу до ресурсу: http://wwwformal.stanford.edu/jmc/recursive/recursive.html.

17. Meszaros G. xUnit Test Patterns: Refactoring Test Code // Gerard Meszaros. Addison-Wesley. 2019. – 833 с.

18. Mike Cohn. Succeeding with Agile. Software Development Using Scrum: книга // Mike Cohn. Addison-Wesley Professional. 2019 – 512 с.

19. National vulnerability database [Електронний ресурс] // The National Institute of Standards and Technology – Режим доступу до ресурсу: https://nvd.nist.gov/.

20. Node.js - Market Share & Web Usage Statistics [Електронний ресурс] // SimilarTech – Режим доступу до ресурсу: https://www.similartech.com/technologies/nodejs.

21. OWASP Zed Attack Proxy Project [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

22. Pulse of the Profession 2019 [Електронний ресурс] // Project Management Institute, Inc. – Березень 2019. Режим доступу до ресурсу: https://www.pmi.org/learning/thought-leadership/pulse/pulse-of-the-profession-2019.

23. Robinson I. Consumer-Driven Contracts: A Service Evolution Pattern [Електронний ресурс] / Ian Robinson // Martin Fowler Blog. – 2019. – Режим доступу до ресурсу: https://martinfowler.com/articles/consumerDrivenContracts.html.

24. Standard Glossary of Terms Used in Software Testing [Електронний ресурс] International Software Testing Qualifications Board. – Грудень 2019. ред. ISTQB Glossary Working Group, Matthias Hamburg, Gary Mogyorodi. Версія 3.3. Режим доступу до ресурсу: https://glossary.istqb.org/.

25. Strategies for Testing Web Applications from the Client Side [Електронний ресурс] – Режим доступу до ресурсу: https://cs.fit.edu/media/TechnicalReports/cs-2003-11.pdf

26. The State of JavaScript [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://2019.stateofjs.com/testing/.

27. The State of the Octoverse [Електронний ресурс] – Режим доступу до ресурсу: https://octoverse.github.com/.

28. Top 10 Automation Testing Tools in 2021 [Електронний ресурс] – Режим доступу до ресурсу: https://www.netsolutions.com/insights/top-10-automation-testing-tools/

29. Vocke H. The Practical Test Pyramid [Електронний ресурс] / Ham Vocke // Martin Fowler Blog. – 2019. Режим доступу до ресурсу: https://martinfowler.com/articles/practical-test-pyramid.html.

30. When and what to automate in software testing? A multi-vocal literature review [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.sciencedirect.com/science/article/abs/pii/S0950584916300702.

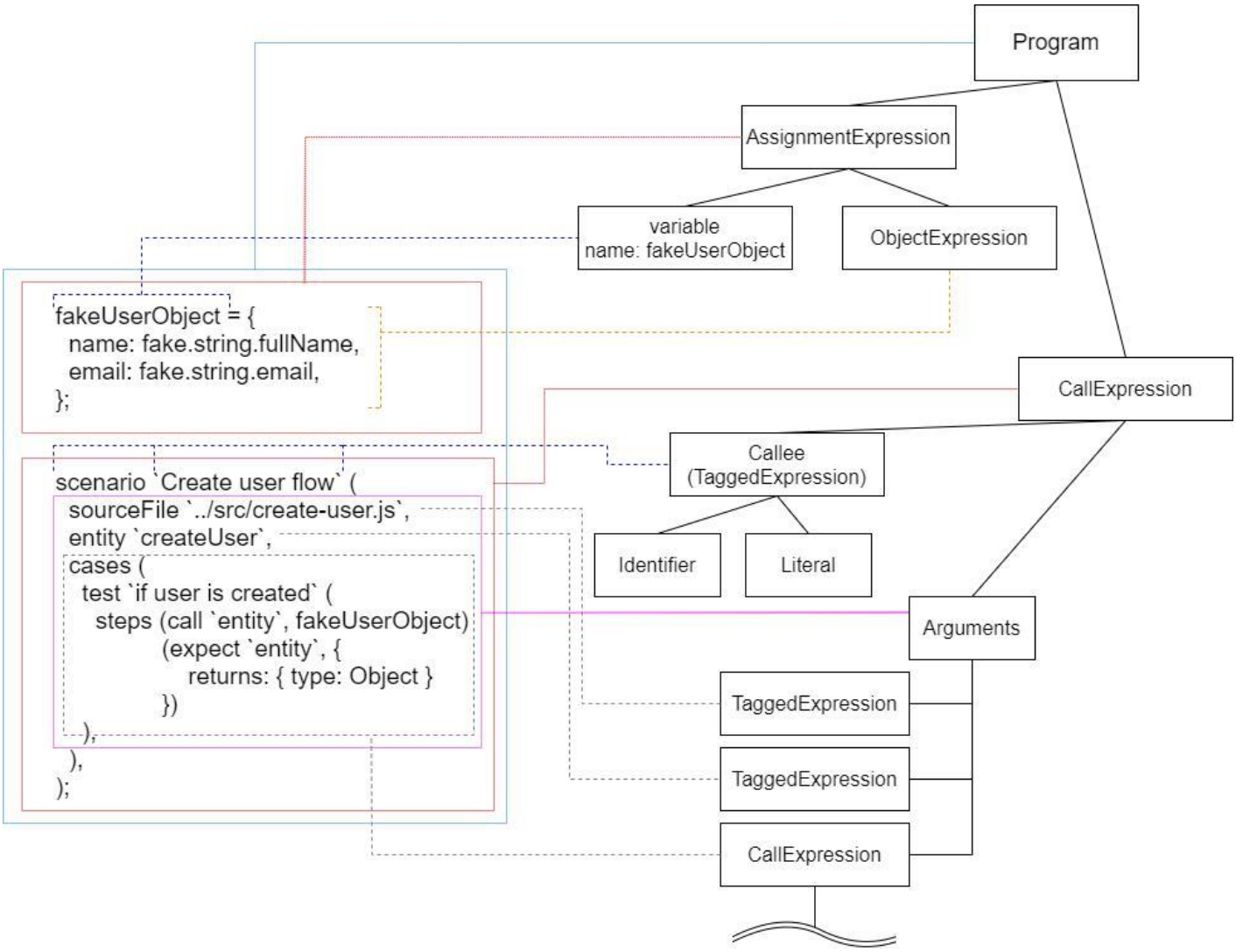
ДОДАТКИ

ДОДАТОК А

Формалізація створеної предметно-орієнтованої мови у розширеній нотації Бекуса-Наура

ДОДАТОК Б

Абстрактне синтаксичне дерево сценарію автоматизованого тестування в загальному випадку



ДОДАТОК В

Схема-структурна послідовності роботи зі змінними у розробленому фреймворку автоматизованого тестування