Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота

за освітнім ступенем «магістр»

на тему:

РОЗРОБКА ФРЕЙМВОРКУ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Виконав:
здобувач 2 курсу
групи М-КН-21
спеціальності 122 «Комп’ютерні науки»
Гущук Віталій Ігорович

Науковий керівник:
д.т.н., професор Турбал Ю. В.

Рівне – 2024

ЗМІСТ

ВСТУП

У сучасних умовах швидкого розвитку технологій важливим аспектом ефективної розробки програмного забезпечення є забезпечення його якості. Одним із способів досягнення високої якості є автоматизація тестування, яка дозволяє значно скоротити час і зусилля, необхідні для перевірки функціональності програмного продукту.

Одним із головних завдань автоматизації процесу тестування є підвищення ефективності, збільшення варіантів роботи систем та прискорення тестування. Автотести дозволяють запускати тестові сценарії регулярно та у будь-який час, завдяки чому можна якнайшвише виявляти помилки після додавання чи змінення функціоналу системи. Надають розробникам звіт про стан продукту одразу після завершення тестування, забезпечують підтримку Agile, крім того здатні виявляти помилки, які були пропущені на стадії ручного тестування.

Такий вид тестування доцільно використовувати для регресійного тестування, тестування з різним набором конфігурацій та на різних платформах. Існує думка, що регресійне тестування є досить складним та заплутаним видом тестування програмного забезпечення. Але наявність сучасних інструментів для автоматизації регресійного тестування та коректне використання таких інструментів дозволяє автоматизувати проведення регресійних тестів.

Регресійне тестування дозволяє перевірити функціональність програми після внесення в неї змін. Це досить складна та кропітка робота, яка вимагає 8 детального аналізу. Автоматизація регресійних тестів дозволить суттєво полегшити роботу. Проте, найчастіше існуючі інструменти автоматизованого тестування відповідають лише декільком з наведених пунктів.

Тому виникає необхідність створення програмного забезпечення для

вирішення цієї проблеми.

В основі обраного підходу є створення інструментарію для автоматизованого тестування, що дозволило б інженерам уникнути запуску тест сценаріїв вручну, зконцентрувавшись на створенні складніших тест-кейсів для перевірки функціональності сайту.

Така система допомагає уникнути регресії та надає можливість проаналізувати виконані кроки тест сценаріїв.

Актуальність дослідження обумовлена необхідністю створення фреймворків для автоматизації тестування, що забезпечують зручність у написанні тестів, їх виконанні та інтеграції в процеси розробки програмного забезпечення. Створення такого фреймворку дозволяє скоротити час на виконання тестів і підвищити їх точність, що є критичним у великих та складних проєктах.

Метою дослідження є розробка фреймворку для автоматизації тестування веб-додатків, що дозволяє значно покращити процес тестування, зробити його більш ефективним та менш затратним за часом. Дослідження спрямоване на розробку універсальних інструментів для автоматизованого тестування, які можуть бути інтегровані в різні середовища розробки.

Завданнями дослідження є:

1. Вивчення існуючих фреймворків для автоматизації тестування.
2. Проєктування архітектури фреймворку для автоматизації тестування.
3. Розробка та реалізація інструментів для автоматизованого тестування.
4. Тестування розробленого фреймворку та оцінка його ефективності.

Об'єктом дослідження є процес автоматизації тестування програмного забезпечення, зокрема веб-додатків. Предметом дослідження

є фреймворки для автоматизації тестування, їх архітектура, інструменти та методи реалізації.

Методи дослідження включають аналіз існуючих фреймворків, проектування архітектури, розробку програмного забезпечення, а також тестування і оцінку ефективності розробленого продукту.

Практичне значення дослідження полягає у створенні інструментів для автоматизованого тестування, що дозволяють розробникам програмного забезпечення покращити якість та ефективність тестування веб-додатків, зменшити витрати часу та ресурсів на тестування, а також забезпечити стабільність і надійність програмних продуктів.

Апробація результатів дослідження проводилася на етапах тестування фреймворку та його використання в реальних проєктах, де розроблені інструменти були інтегровані в процеси розробки та тестування.

Структура роботи включає: вступ, огляд літератури, методологію розробки фреймворку, практичну частину, висновки та список використаних джерел.

У роботі також наведено додатки, які містять код та тестові приклади.

РОЗДІЛ І. ТЕОРЕТИЧНІ ОСНОВИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Основні поняття та визначення

Тестування програмного забезпечення є невід'ємною частиною створення
програмного продукту. Від того, наскільки досконало створені тести, залежить те, як скоро проект буде зданий остаточно, і чи буде необхідність згодом усувати помилки.

Тестування програмного забезпечення - це процес, що використовується для виміру якості розроблюваного програмного забезпечення. Зазвичай, поняття якості обмежується такими поняттями, як коректність, повнота, безпечність, але може містити більше технічних вимог. Тестування - це процес технічного дослідження, який виконується на вимогу замовників, і призначений для вияву інформації про якість продукту відносно контексту, в якому він має використовуватись. До цього процесу входить виконання програми з метою знайдення помилок.

Тестування так само можна описати як процес верифікації та валідації того чи іншого програмного продукту, щоб дізнатися на скільки точно він задовольняє всім встановленим вимогам.

Верифікація (Verification – узгодження) – це процес оцінки системи або її компонентів з метою визначення чи задовольняють результати поточного етапу розробки умовам, сформованим на початку цього етапу (чи виконуються наші цілі, терміни, завдання, по розробці проекту, визначені на початку поточної фази.)

Валідація (Validation – затвердження) – це визначення відповідності ПЗ очікуванням і потребам користувача, вимогам до системи.

Тестоване програмне забезпечення повинно проходити кожен з етапів тестування, обумовлених продуктовнерами, менеджментом компанії розробника та тест-дизайнерами для того, щоб вважати програмний продукт відносно якісним або придатним до використання.

На сьогодні розрізняють близько 150 видів тестування програмного забезпечення, щоб за необхідності протестувати програмний продукт від А до Я на усі 100%

1.2 Роль тестування в розробці програмного забезпечення

Під час користування будь-якою програмою ми навіть не замислюємося про те, чого та чи інша програма працює без помилок, зависань тощо. Саме в цьому аспекті тестування відіграє далеко не останню скрипку.

Чому ж тестування в рамках даного процесу настільки важливе? Ось кілька причин:

1) Тестування дозволяє перевірити, чи правильно реалізовано усі вимоги до програмного забезпечення, що розроблялось.

2) Тестування допомагає у виявленні помилок та забезпечує їх розпізнавання і вирішення до етапу розгортання програмного забезпечення.

3) Тестування пом’якшує наслідки та ризики втрат, якщо програмний продукт все ж випустили по неправильним вимогам. В такому випадку намагаються частково виправити, переоцінити та покращити програму.

4) Тестування допомагає перевірити належну інтеграцію та взаємодію програми з навколишнім середовищем.

Звідси основна мета тестування програмного забезпечення, яка полягає у намаганні виміряти рівень якості програмного продукту з точки зору основних вимог.

Люди схильні помилятися, людські помилки можуть призводити до порушення нормальної роботи програмного забезпечення на всіх стадіях

розробки, причому наслідки цього можуть бути найрізноманітнішими — від незначних до катастрофічних.

Для програмного забезпечення будь-якої складності неможливо вилучити всі проблеми. Проте тестування допомагає виявити більшість помилок, з якими може зіткнутися користувач, й потенційно зменшує ризик виникнення проблем з програмою у майбутньому.

1.3. Методології тестування

Наразі, існують три найпопулярніші методології тестування. В залежності від того, чи наявний доступ у тестувальника до безпосередньо самого коду програми, методології розділяють на три типи: • метод чорного ящика; • метод білого ящика;

• метод сірого ящика.

Метод чорного ящика.

Тестування "чорного ящика" також відомо як поведінкове тестування, тестування "непрозорого ящика", тестування "закритого ящика", тестування на основі специфікацій або тестування "очі в очі" [4]. Схематичне представлення методу на рис. 1.1

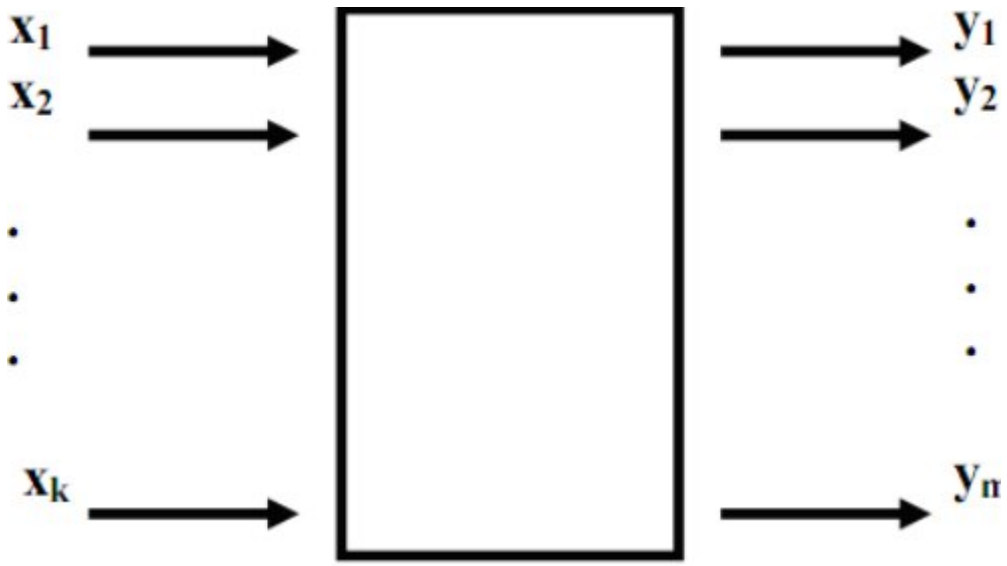


Рисунок 1.1 – схематичне представлення методу чорного ящика

Це метод тестування програмного забезпечення, який аналізує функціональність програмного забезпечення, або розроблюваного додатку, не знаючи нічого про внутрішню структуру, або дизайн тестованого елемента, і порівнює вхідне значення з вихідним.

Основна увага при тестуванні методом "чорного ящика" приділяється функціональності системи в цілому. Термін "поведінкове тестування" також використовується для позначення тестування "чорного ящика". Проектування поведінкових тестів дещо відрізняється від проектування тестів "чорного ящика", оскільки використання внутрішніх знань не є строго забороненим, але все ж не вітається.

Більшість додатків тестується методом "чорного ящика". Оскільки необхідно охопити більшість тестових випадків, щоб більшість помилок було виявлено.

Таке тестування проводиться протягом усього життєвого циклу розробки і тестування програмного забезпечення, тобто на етапах модульного, інтеграційного, системного, приймального і регресивного тестування.

Метод білого ящика.

Якщо йти за визначенням, то "тестування білого ящика" (також відоме як чисте, скляне або структурне тестування) - це метод тестування, який оцінює код і внутрішню структуру програми [4]. Схематичне представлення методу зображено на рис. 1.2.

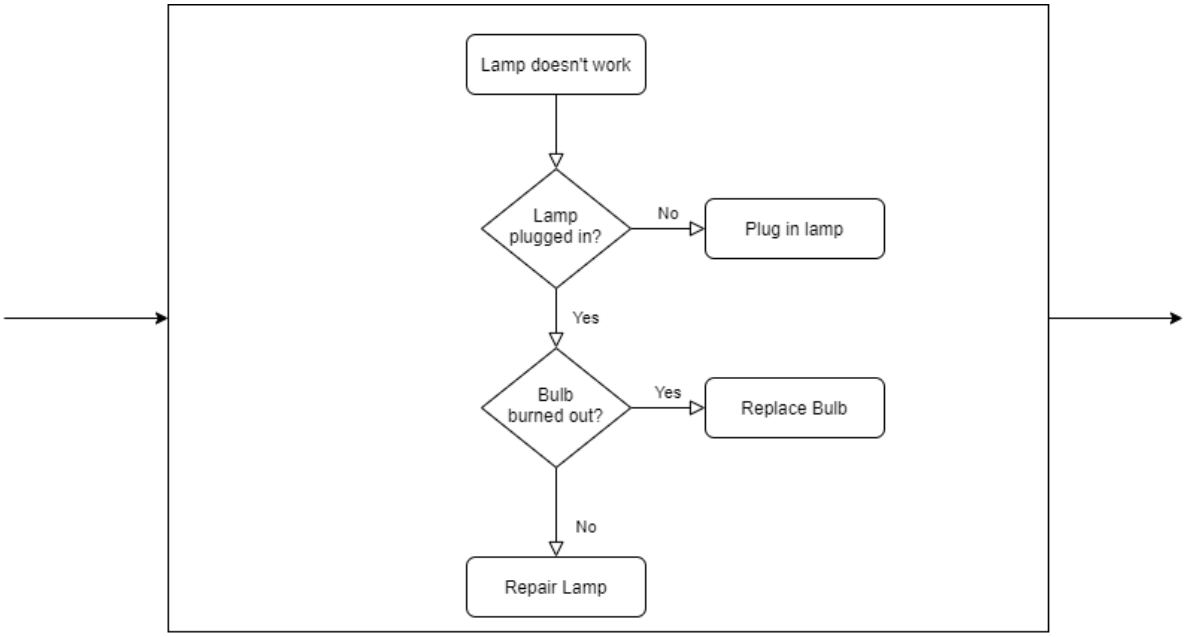


Рисунок 1.2 – схематичне представлення методу білого ящика

Тестування "білого ящика" передбачає вивчення структури коду. Коли тестувальник знає внутрішню структуру продукту, можна проводити тести, щоб переконатися, що внутрішні операції виконуються відповідно до специфікації. І всі внутрішні компоненти, або окремі модулі програми були правильно відпрацьовані.

Метод сірого ящика.

Тестування «сірого ящика» - це метод тестування програмного забезпечення, що дозволяє протестувати програмний продукт або додаток з частковим знанням внутрішньої структури програми. Метою тестування "сірого ящика" є пошук і виявлення дефектів, пов'язаних з неправильною структурою коду або неправильним використанням додатку. У цьому процесі, зазвичай, виявляються контекстно-специфічні помилки, пов'язані з веб-системами. Це збільшує охоплення тестування, концентруючись на всіх шарах будь-якої складної системи. Схематичне представлення методу зображене на рис. 1.3.

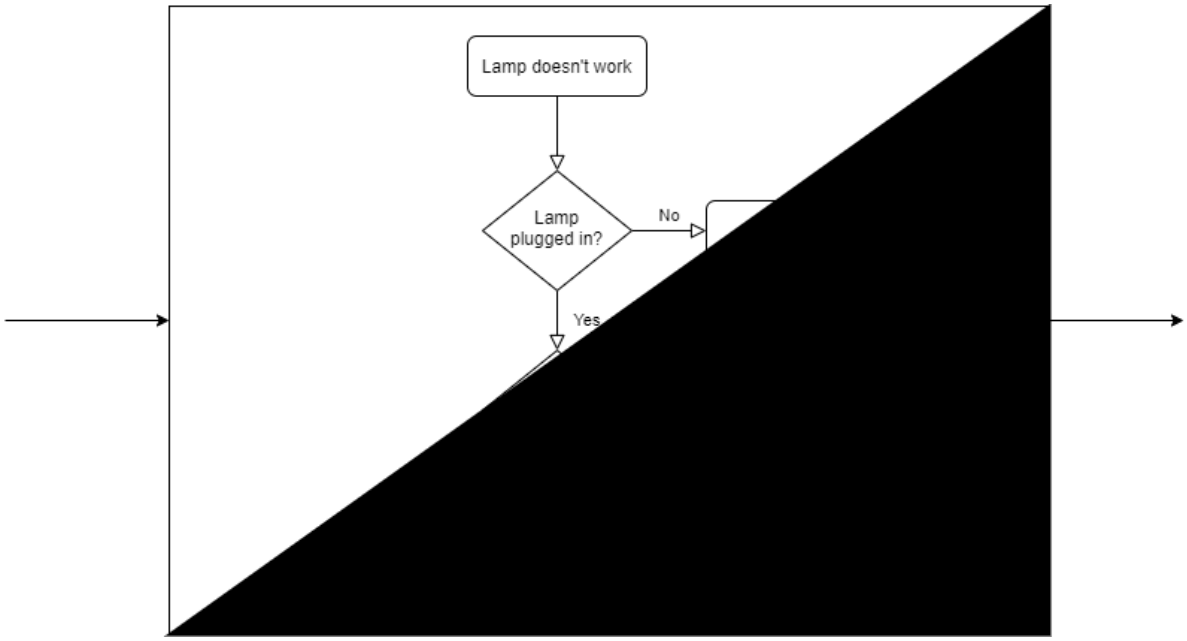


Рисунок 1.3 – Схематичне зображення методу сірого ящика

1.4 Класифікація видів тестування

За рівнем тестування:

Рівні тестування включають в себе різні методології, які можуть бути використані при проведенні тестування програмного забезпечення. Основними рівнями тестування програмного забезпечення є ***функціональне*** тестування та ***нефункціональне*** тестування.

Функціональне тестування – це підтип тестування "чорного ящика", який заснований на специфікаціях програмного забезпечення, що підлягають тестуванню. Додаток тестується шляхом введення вхідних даних, а потім вивчаються результати, які повинні відповідати функціональності, для якої воно призначене. Функціональне тестування програмного забезпечення проводиться на повній, інтегрованій системі для оцінки відповідності системи заданим вимогам.

Існує п'ять етапів тестування функціональності додатку:

- визначення функціональності, яку має виконувати об’єкт для тестування;
- створення тестових даних на основі специфікацій додатку;

- висновок на основі тестових даних і специфікацій додатку;
- написання тестових сценаріїв і виконання тестових прикладів;
- порівняння фактичних і очікуваних результатів на основі виконаних тестових сценаріїв.

Ефективна практика тестування передбачає застосування вищевказаних кроків в політиці тестування кожної організації, і, це гарантує, що організація підтримує найсуворіші стандарти, коли мова йде про якість програмного забезпечення.

Нефункціональне тестування засноване на тестуванні програми з точки зору не функціональних атрибутів програми. Нефункціональне тестування включає в себе тестування програмного забезпечення на основі вимог, які не є функціональними за своєю природою, але важливі, наприклад, продуктивність, безпеку, призначений для користувача інтерфейс і т.д.

Деякі з важливих і часто використовуваних типів нефункціонального тестування обговорюються нижче.

До функціонального тестування відносять:

• ***Юніт-тестування*** – цей вид тестування виконується розробниками до того, як програма буде передана команді тестувальників для формального виконання тестових випадків. Юніт-тестування виконується відповідними розробниками на окремих ділянках вихідного коду для призначених модулів. Розробники використовують тестові дані, які відрізняються від тестових даних команди QA. Мета модульного тестування - ізолювати кожну частину програми і показати, що окремі частини програми працюють коректно з точки зору вимог і функціональності.

Обмеження юніт-тестування:

- тестування не може виявити всі помилки в програмі;
- неможливо протестувати кожен випадок виконання програми.

Існує обмеження на кількість сценаріїв і тестових даних, які розробник може використовувати для перевірки вихідного коду. Після вичерпання всіх можливостей не залишається іншого вибору, окрім як припинити модульне тестування і об'єднати ділянку коду з іншими модулями.

Інтеграційне тестування. Головна мета інтеграційного тестування полягає в тестуванні об'єднаних частин програми для визначення правильності їх функціонування. Інтеграційне тестування може бути виконано двома способами: інтеграційне тестування знизу вгору і інтеграційне тестування зверху вниз. Інтеграційне тестування знизу вгору починається з тестування нижчих модулів, за якими слідує тестування все більш високорівневих комбінацій модулів. Інтеграційне тестування зверху вниз полягає в тестуванні спочатку модулів найвищого рівня, а потім, поступово, модулі найнижчого рівня. У комплексному середовищі розробки програмного забезпечення, зазвичай, спочатку проводиться тестування знизу вгору, а потім тестування зверху вниз. Процес завершується багаторазовим тестуванням всієї програми, переважно в сценаріях, розроблених для імітації реальних ситуацій.

• ***Системне*** тестування. Системне тестування полягає в перевірці системи в цілому. Після інтеграції всіх компонентів програма в цілому піддається ретельному тестуванню, щоб переконатися в відповідності встановленим стандартам якості. Цей вид тестування виконується спеціалізованою командою тестувальників. Системне тестування важливо з наступних причин:

- системне тестування - це перший крок в життєвому циклі розробки програмного забезпечення, на якому тестується додаток в цілому;
- програма ретельно тестується, щоб переконатися, що вона відповідає функціональним і технічним специфікаціям;

- програма тестується в середовищі, яке дуже близьке до реального середовища, де програму буде розгорнуто;
- системне тестування дозволяє нам протестувати, перевірити і підтвердити як бізнес-вимоги, так і архітектуру програми.

• **Регресивне** тестування. При внесенні змін до частини функціоналу програми, цілком можливо, що ці зміни торкнулися і іншого функціоналу програми. Регресивне тестування проводиться для перевірки того, що виправлена помилка не призвела до порушення іншої функціональності або бізнес-логіки. Мета регресивного тестування – переконатися, що зміни, наприклад, виправлення помилки, не повинні привести до появи інших помилок в програмі. Регресивне тестування важливо з наступних причин:

- мінімізація непротестованих ділянок коду, коли необхідно протестувати програму з внесеними змінами;
- тестування нових змін для перевірки того, що внесені зміни не вплинули на будь-яку іншу область застосування;
- зниження ризиків появи помилок, про які не знають тестувальники;
- охоплення тестування збільшується без шкоди для термінів;
- підвищення швидкості виведення продукту на ринок.

• ***Приймальне*** тестування – це, мабуть, найважливіший вид тестування, оскільки його проводить команда тестувальників, яка оцінює, чи відповідає додаток заявленим специфікаціям і задовольняє вона вимогам
клієнта. У команди QA повинен бути набір заздалегідь написаних сценаріїв і тестових випадків, які будуть використовуватися для тестування програми. В ході тестування буде висловлено більше ідей щодо програми та проведено більше тестів, щоб оцінити її точність і причини, за якими було ініційовано проект. Приймальні тести призначені не тільки для виявлення простих орфографічних помилок, косметичних помилок або прогалин в інтерфейсі, але і для виявлення будь-яких помилок в програмі, які приведуть до збоїв системи або серйозних помилок в роботі програми.

Виконуючи приймальні випробування програми, команда тестувальників знижує ефективність роботи програми у виробництві. Існують також юридичні і договірні вимоги до приймання системи.

• ***Тестування зручності*** використання. Це спосіб тестування, що дозволяє оцінити ступінь зручності використання програми, швидкість навчання користувачів при роботі з програмою, а також наскільки користувачі продукту, що розробляється знаходять її зрозумілою і привабливою в контексті заданих вимог. Таке тестування необхідно для забезпечення максимально позитивного користувацького досвіду при роботі з додатком.

• ***Альфа-тестування*** – це тестування є першим етапом тестування і проводиться серед команд (команди розробників і QA). Юніт-тестування, інтеграційне тестування і системне тестування, об'єднані разом, відоме як альфа-тестування. На цьому етапі в додатку будуть перевірені наступні аспекти:

- орфографічні помилки;
- непрацюючі посилання;
- відповідність напрямків, за якими програма спілкується з сервером;
- додаток буде тестуватися на машинах з найнижчими технічними характеристиками для перевірки часу завантаження і будь-яких проблем із затримкою.

• ***Бета-тестування*** – це тестування проводиться після успішного проведення альфа-тестування. При бета-тестуванні додаток тестується на вибірці цільової аудиторії. Бета-тестування також відомо як предрелізне тестування. Бета-тестові версії програмного забезпечення в ідеалі поширюються серед широкої аудиторії в Інтернеті, частково для того, щоб дати програмі випробування в "реальному світі", а частково для попереднього перегляду наступного релізу. На цьому етапі аудиторія буде тестувати наступне:

- користувачі встановлять, запустять додаток і відправлять свої відгуки команді проекту;
- друкарські помилки, заплутана логіка роботи з програмою і навіть збої;
- отримавши зворотний зв'язок, команда проекту може виправити проблеми, перш ніж випустити програмне забезпечення для реальних користувачів;
- чим більше проблем, які ви виправили, вирішуючи реальні проблеми користувачів, тим вищою буде якість вашої аихуднох програми;
- більш якісна програма, коли ви випустите її для широкої публіки, підвищить задоволеність клієнтів.

До ***нефункціонального*** тестування відносяться:

• ***Тестування продуктивності***. Воно в основному використовується для виявлення вузьких місць або проблем з продуктивністю, а не для пошуку помилок в програмному забезпеченні. Існують різні причини, які сприяють зниженню продуктивності програмного забезпечення:
- затримка в мережі;
- обробка на стороні клієнта;
- обробка транзакцій в базі даних;
- балансування навантаження між серверами;
- візуалізація даних.

Тестування продуктивності вважається одним з важливих і обов'язкових видів тестування з точки зору наступних аспектів:
- швидкість (тобто час відгуку, рендеринг даних і доступ до них);
- продуктивність;
- стабільність;
- масштабованість.

Тестування продуктивності може бути як якісним, так і кількісним і може бути розділене на різні підтипи, такі як тестування навантаження і стрес-тестування.

• ***Тестування навантаження*** – це процес тестування поведінки програмного забезпечення шляхом застосування максимального навантаження з точки зору доступу програмного забезпечення до великих вхідних даних і маніпулювання ними. Воно може проводитися як за нормальних умов, так і при піковому навантаженні. Цей тип тестування дозволяє визначити максимальну потужність програмного забезпечення і його поведінку в піковий час. Найчастіше тестування навантаження проводиться за допомогою автоматизованих інструментів, таких як Load Runner, AppLoader, IBM Rational Performance Tester, Apache JMeter, Silk Performer, Visual Studio Load Test і т.д. Віртуальні користувачі (VUsers) визначаються в автоматизованому інструменті тестування і виконується сценарій для перевірки навантажувального тестування програмного забезпечення. Кількість користувачів може бути збільшена або зменшена одночасно або поступово в залежності від вимог.

• ***Стрес-тестування*** включає в себе перевірку поведінки програмного забезпечення в аномальних умовах. Наприклад, воно може включати вилучення деяких ресурсів або застосування навантаження, що перевищує фактичну межу навантаження [2]. Метою стрес-тестування є перевірка програмного забезпечення шляхом прикладання навантаження до системи і забору ресурсів, використовуваних програмним забезпеченням, для визначення точки відмови. Це тестування може бути виконано шляхом тестування різних сценаріїв, таких як:

- випадкове відключення або перезапуск мережевих портів;
- відключення, або включення баз даних;
- запуск різних процесів, які споживають ресурси, такі як процесор, пам'ять і т.д.

• ***Тестування зручності використання*** є технікою "чорного ящика" і використовується для виявлення будь-яких помилок і поліпшень в програмному забезпеченні шляхом спостереження за користувачами в процесі їх використання програми та роботи з нею. Зручність використання можна визначити з точки зору п’яти чинників:

- ефективність використання;
- здатність до навчання;
- здатність запам’ятовувати;
- помилки / безпека;
- задоволеність.

Зручність використання продукту буде на високому рівні, а система придатною для використання, якщо вона володіє перерахованими вище факторами.

Різниця між UI тестуванням та тестуванням зручності використання:

- UI тестування включає в себе тестування графічного інтерфейсу користувача програмного продукту;
- UI тестування гарантує, що графічний інтерфейс функціонує відповідно до вимог і перевіряється з точки зору кольору, вирівнювання, розміру та інших властивостей;
- тестування зручності використання забезпечує хороший і зручний графічний інтерфейс користувача, з яким легко працювати;
- UI тестування можна розглядати як частину тестування зручності використання.

• ***Тестування безпеки***. Тестування безпеки включає в себе тестування програмного забезпечення з метою виявлення будь-яких недоліків та прогалин з точки зору безпеки та вразливості. Нижче перераховані основні аспекти, які має забезпечувати тестування безпеки:

- конфіденційність;
- цілісність;

- аутентифікація;
- доступність;
- авторизація;
- програмне забезпечення захищене від відомих і невідомих вразливостей;
- дані програмного забезпечення надійно захищені;
- програмне забезпечення відповідає усім нормам безпеки;
- перевірка та валідація введених даних;
- атаки SQL ін’єкціями;
- проблеми управління сеансами;
- уразливість переповнення буфера.

• ***Тестування портативності***. Тестування портативності включає в себе тестування програмного забезпечення з метою забезпечення можливості його повторного використання і можливості перенесення з іншого програмного забезпечення. Нижче перераховані стратегії, які можуть бути використані для тестування портативності:

- перенесення встановленого програмного забезпечення з одного комп'ютера на інший;
- створення виконуваного файлу (.exe) для запуску програмного забезпечення на різних платформах;

Тестування портативності можна розглядати як один з розділів системного тестування, оскільки цей вид тестування включає в себе загальне тестування програмного забезпечення на предмет його використання в різних середовищах. Комп'ютерне обладнання, операційні системи і браузери є основними об'єктами тестування портативності. Умови для тестування портативності:

- програмне забезпечення повинно бути розроблено з урахуванням вимог портативності;
- проведено модульне тестування пов'язаних компонентів;

- проведено інтеграційне тестування;
- створене тестове середовище.

<u>За ступенем автоматизації:</u>

• ***Ручне тестування***. Ручне тестування включає в себе тестування програмного забезпечення вручну, тобто без використання будь-яких автоматизованих інструментів або сценаріїв. У цьому типі тестування тестувальник бере на себе роль кінцевого користувача і тестує програмне забезпечення, щоб виявити будь-яку неочікувану поведінку або помилку. Існують різні етапи ручного тестування, такі як модульне тестування, інтеграційне тестування, системне тестування і тестування прийняття користувачем. Тестувальники використовують тест плани, тестові випадки або тестові сценарії для тестування програмного забезпечення, щоб забезпечити повноту тестування. Ручне тестування також включає в себе дослідне тестування, коли тестувальники досліджують програмне забезпечення, щоб виявити в ньому помилки.

• ***Автоматичне тестування.*** Автоматичне тестування, яке також відоме як автоматизоване тестування, - це коли тестувальник пише сценарії і використовує інше програмне забезпечення для тестування продукту. Цей процес має за мету автоматизацію ручного процесу. Автоматизоване тестування використовується для повторного виконання сценаріїв тестування, які були виконані вручну, швидко і багаторазово [2]. Крім регресійного тестування, автоматизоване тестування також використовується для тестування програми з точки зору навантаження, продуктивності і стресостійкості. Воно збільшує покриття тестів, підвищує точність, економить час і гроші в порівнянні з ручним тестуванням.

1.5. Проблеми тестування програмного забезпечення.

Ручне і автоматизоване тестування сьогодні грають істотну роль в будь-якої технологічної компанії. Будь то мобільний або веб-додаток або сайт, перевірка коду вкрай важлива. Правильне планування, коли і яке

тестування використовувати, допомагає зберігати час і гроші. Обидві методики тестування мають свої переваги і недоліки, їх ми розглянемо нижче.

Ручне тестування може займати багато часу, зате в короткостроковій перспективі заощадить в рази більше грошей. Його вартість залежить тільки від тестувальника, а не інструментів для автоматизації. Ручне тестування можна розглядати як взаємодію професійного тестувальника і софта з метою пошуку багів. Таким чином, під час ручного тестування можна отримувати фідбек, що неможливо у зв'язку з автоматизованою перевірці. Іншими словами, взаємодіючи з додатком безпосередньо, тестувальник може порівнювати очікуваний результат з реальним і залишати рекомендації. Якщо у вас є QA- команда, ручне тестування не буде проблемою.

Плюси ручного тестування:

- Призначений для користувача фідбек. Весь звіт тестувальника може бути розглянутий як зворотний зв'язок від потенційного користувача.
- UI-фідбек. У наш час для користувача інтерфейс грає величезну роль, і тому повністю протестувати його можна тільки вручну. До речі, чи знаєте ви, які 7 елементів інтерфейсу вам краще прибрати з вашого сайту?
- Дешевизна. У короткостроковій перспективі ручне тестування дешевше, ніж інструменти автоматизованої перевірки.
- Тестування в реальному часі. Незначні зміни можуть бути досліджені відразу, без написання коду і його виконання.
- Можливість дослідного тестування. Його метою є перевірка різноманітних можливостей програми. Важливо, що використовуються не заздалегідь складені тест-кейси, а придумані на льоту сценарії.

Мінуси ручного тестування:

- Людський фактор. Хоча UI і може бути протестований тільки вручну,

люди часто схильні до неефективності. Деякі помилки можуть залишитися непоміченими.

- Трудомісткість повторного використання. Провести серію стандартних автоматичних тестів простіше, ніж протестувати проект вручну після внесення навіть невеликих змін.

- Неможливість навантажувального тестування. Не можна змоделювати велику кількість користувачів вручну.

- Плюси автоматизованого тестування:

- Можливість навантажувального тестування.

- Можна досить швидко змоделювати велику кількість користувачів.

- Економія часу.

- Ручне тестування великих додатків - довгий і трудомісткий процес, в той час як сценарії пишуться лише один раз.

- Можливість повторного використання.

- Тестовий сценарій, написаний один раз, може бути використаний і в майбутньому при черговому оновленні проекту.

Мінуси автоматизованого тестування:

- UI-тестування. Автоматизоване тестування не може в повній мірі покрити вимоги до призначеного для користувача інтерфейсу.

- Відсутність «людського погляду». Можливо існування помилок, які помітить тільки людина.

РОЗДІЛ ІІ. АНАЛІЗ ПІДХОДІВ ДО АВТОМАТИЗАЦІЇ ТЕСТУВІАННЯ

2.1 Життєвий цикл автоматизації тестування

Рисунок 2.1 – Схема життєвого циклу автоматизації тестування

Керівники проектів та розробники досить часто стикаються з проблемою створення додатків в умовах недостатньої кількості ресурсів та постійно скорочуваного графіка. Незалежно від того, наскільки досвідчені програмісти працюють над створенням певного ПЗ, все одно потрібно перевіряти чи дійсно воно відповідає вимогам потрібно. Таким чином, організації переходять до автоматизації тестування для ефективного досягнення цієї мети.

Коли ми говоримо про автоматизацію тестування, багато хто з нас вважає, що це лише частина SDLC (життєвого циклу розробки програмного забезпечення), але для досягнення найкращих результатів, потрібно дотримуватися саме повного циклу, відомого під назвою -життєвий цикл автоматизації тестування.

Реалізація життєвого циклу автоматизації тестування виконується паралельно з процесом життєвого циклу розробки програмного забезпечення.

Сам життєвий цикл є багатоетапним процесом, який підтримує діяльність, необхідну для використання та впровадження автоматизованого інструменту тестування, розробки та запуску тест кейсів, розробки тестового дизайну, побудови та обробки даних тесту та середовища.

У методології тестова конструкція побудована для відображення тестових зусиль та для надання проекту й тестовій групі розуміння щодо обсягу робіт з тестування.

Існує 6 основних етапів методології тестування життєвого циклу автоматизації:

- Визначення об'єму робіт з автоматизації тестування
- Вибір правильної технології для автоматизації
- Тест план + Тест Дизайн + Тест Стратегія
- Налаштування тестового середовища
- Розробка коду для запуску автоматизованих тестів і їх запуск
- Аналіз і генерація тестових звітів[6]

Визначення об'єму робіт з автоматизації тестування:

Це перший етап життєвого циклу АТ, і він спрямований на визначення наскільки взагалі доцільно застосовувати автоматизацію. Тут потрібно врахувати скільки часу є на побудову процесів та наскільки важко створити модель фреймворку автоматизації. Крім того, важливо провести аналіз техніко-економічного обґрунтування на ручному пакеті тестових кейсів, який дозволяє інженерам-автоматизаторам розробляти програмні сценарії.

У цьому конкретному етапі слід обережно ставитися до наступних речей:

1. Які модулі додатку можна автоматизувати, а які ні?

2. Які тест кейси можна автоматизувати та як їх автоматизувати?

3. Такі фактори, як вартість, чисельність команди та досвід також слід враховувати.

Перед початком автоматизації тестів слід виконати наступні перевірки техніко-економічного обґрунтування:

• Можливість автоматизації тестових випадків

• Автоматизація автоматичної автоматизованості

Згадаймо всі компоненти програмного інтерфейсу та виділимо найбільш важливі для користувача. В першу чергу слід звернути увагу на ті, несправність яких може призвести до втрати найбільшої кількості грошей. Також необхідно визначити відсоток компонентів користувальницького інтерфейсу, які потрібно автоматизувати за допомогою інструменту автоматичного тестування. Спробуймо знайти засоби тестування автоматизації, які допоможуть автоматизувати компоненти інтерфейсу з невеликими змінами. Це полегшить роботу на наступному етапі.

Вибір правильної технології для автоматизації:

АТ дуже залежить від інструментів. Ось чому пошук правильного інструменту тестування автоматизації є критичною фазою для ЖЦАТ. Коли ми шукаємо інструмент автоматизації, нам потрібно пам’ятати про бюджет, технології що використовуються в проекті, знання розробників та тестувальників всередині команди, гнучкість тощо. Потрібно обирати інструмент, який підтримується розробниками і має добре оформлену документацію.

Наприклад, якщо ми шукаєте автоматизований інструмент для перевірки сумісності браузера, нам потрібно чітко розуміти різноманітність підтримуємих браузерів. Можливість зйомки відео-звітів, метаданих скриптів автоматизації серед різних браузерів та пристроїв. Механізм

виділення та відстеження помилок, спосіб логування програмного коду та інші.

Тест план + Тест Дизайн + Тест Стратегія:

Це найважливіша фаза методології ЖЦАТ, яка визначає, як досягти основної мети АТ. Планування тестової автоматизації - це перше і головне, на що потрібно звернути увагу на цьому етапі.

Вибір інструменту залежить від технологій, які використовуються в додатку. Потрібно повністю розуміти свій продукт перед початком автоматизації.

Наприклад, якщо це настільний додаток - знайдіть на якій мові він побудований. Для вебдодатку, знайте про можливості тестового фреймворку, його особливості роботи з компонентами, які ви використовуєте.

На етапі планування тестування, команда тестування приймає рішення про стандарти та рекомендації щодо створення тестової процедури, обладнання, програмного забезпечення та мережі для підтримки тестового середовища, попередній графік випробувань, вимоги до тестових даних. Повинна бути чітка процедура відстеження дефектів та пов'язаний з цим інструмент відстеження контролю конфігурації тесту та середовищ інсценування.

Команда тестових інженерів розробляє тестову архітектуру для опису структури тестової програми та способів управління процедурами тестування після розроблення моделі тестової стратегії.

П ісля проектування створюється тестова архітектура, де описана структура всієї програми тестування, а також шляхи управління цією тестовою процедурою.

Не забудьте врахувати наступні речі при плануванні тестової стратегії:

Зберіть всі ручні тестові кейсі з інструменту управління тестом, щоб визначити, який тестовий випадок потрібно автоматизувати.

Визначте, які рамки слід використовувати після розуміння плюсів і мінусів інструменту тестування. Побудуйте тестовий набір для тестового випадку автоматизації в інструменті для управління тестами.

У плані тестування обов'язково зазначайте передумови, ризики та залежність між інструментом та програмою. Шукайте схвалення щодо стратегії тестування у клієнтів або зацікавлених сторін.

Налаштування тестового середовища:

Як видно з назви, цей етап життєвого циклу автоматизації тестування передбачає налаштування машини або віддаленої машини, де будуть виконуватися тестові кейси. Навіщо нам потрібні віддалені машини? Тому що, якщо ми не живемо в ідеальному світі, наші користувачі використовуватимуть різні машини для доступу до нашого веб-сайту чи вебдодатку в Інтернеті.

Відслідковувати коректність роботи вебдодатку на різних пристроях - одне, але нам також слід бути обережними щодо різних браузерів та версій браузера. Оскільки наш веб-сайт може відображатися по-різному від одного браузера до іншого. Тестування сумісності веб-переглядачів, також відоме як крос-браузерне тестування, - це процедура, коли ми перевіряємо веб-сайт або вебдодаток у кількох версіях браузера, щоб переконатися, що ми створили безперебійну роботу для усіх наших користувачів.

Етап налаштування навколишнього середовища потребує ретельного планування, ми повинні переконатися, що зможемо максимально охопити тестове покриття для якомога більшої кількості сценаріїв. Потрібно правильно налаштувати навколишнє середовище, встановити програмне забезпечення, мережеві ресурси та обладнання, вдосконалити бази даних тестів та розробити сценарії тестового покриття з неперервною інтеграцією у процес розробки.

Що стосується крос-браузерного тестування браузера, налаштування сотень браузерів та версій браузера на численних пристроях може бути дуже складним завданням, а придбання лабораторії пристроїв - не доступний варіант для всіх. Тут почали грати хмарні інструменти, такі як – Amazon Web Services, Azure Devops, Google Web Services які пропонують понад 2000+ браузерів та версій браузера та розміщують VM для численних настільних та мобільних пристроїв.

Основні сфери для налаштування тестового середовища:

• Тестові данні, – не потрібно постійно створювати велику кількість тестових даних для перевірки функціональності. Якщо є можливість – потрібно заздалегідь створити все необхідне для
прогону тестових випадків. Це може заощадити велику кількість часу на виконання тестів і зберегти від невірних результатів.

• Середовище для тестування навантаження, – важливо мати окреме середовище для навантаження й аналізу можливостей обробки веб-трафіку, підтримки великої кількості користувачів і обробки достатньої кількості запитів одночасно.

• Контрольний список усіх систем, модулів та додатків, які підлягають тестуванню.

• Ізольований сервер баз даних

• Тестування на різних клієнтських операційних системах.

• Тестування на максимальній кількості браузерів з різними версіями браузера.

Переконайтесь, що ви протестуєте свій веб-сайт у низькій та високій мережі, щоб зрозуміти різницю між часом надання та загальним виглядом веб-сайту чи вебдодатка.

Документація є ключовою. Переконайтеся, що ви охоплюєте всі посібники по конфігурації / посібники з установки / посібники користувача в центральному репозиторії.

Налаштування тестового середовища включає такі завдання:

1. Ліцензії на інструменти
2. Інструменти налаштування, такі як сучасні текстові редактори та засоби їх порівняння.
3. Впровадження фреймворку автоматизації
4. Доступ AUT і дійсні облікові дані
5. Ліцензії на надбудови

Більшість організацій має окреме тестове середовище для тестування програмного забезпечення. Найкращий підхід - скопіювати дані з користувацького середовища і використовувати їх для тестування. Це допоможе інженеру-випробувачу розкрити проблеми, не псуючи виробничі дані.

Кращими практиками для налаштування управлінням тестовим довкіллям, можна зазначити:

• Знання тестового середовища та навчіть членів групи тестування.
• Необхідне програмне забезпечення, ліцензії та обладнання.
• Контрольний список засобів автоматизації та їх конфігурації.
• Крос-браузерне тестування, щоб забезпечити покриття тестів на численних браузерах та версіях щодо пріоритету та частки ринку.
• Використання трафіку в режимі реального часу, щоб переконатися, що ваші зміни є більш стійкими.
• Планування використання тестового середовища.

Розробка коду для запуску автоматизованих тестів і їх запуск:

Після встановлення тестового середовища настає час виконати тестовий скрипт. Отже, ця фаза життєвого циклу автоматизації тестування присвячена виконанню всіх тестових сценаріїв.

Для виконання сценаріїв, підписані та перевірені тестові кейси доставляються команді тестування автоматизації. Важливо переконатися що всі тестові сценарії працюють правильно.

Отже, перед розробкою тестового сценарію потрібно подбати про наступні речі:

• Створення тестових сценаріїв повинно бути на основі фактичних вимог. Розробити загальні методи функцій, які можна використовувати протягом усього процесу тестування.

• Створіть багаторазовий, структурований та простий сценарій, щоб будь яка людина могла це зрозуміти.

• Проведіть перегляд коду іншою людиною з команди, для кращої якості продукту і покращення розуміння всього фреймворку членами команди.

• Використовуйте звітність для більш чіткого зображення результатів і загального прогресу у АТ.

Після успішної розробки тестових сценаріїв слід пам'ятати про наступні речі:

1. Тестовий сценарій повинен включати всі функціональні аспекти відповідно до тестового випадку.

2. Запускати тестові сценарії в різних середовищах та на різних платформах.

3. Якщо можливо, паралельне виконання може бути здійснено для економії часу та зусиль.

4. Якщо збій стався через певну функціональність, напишіть звіт про помилку.

Для виконання тестових сценаріїв, команда тестування повинна відповідати графіку, прийнятому для виконання процедури. На цьому етапі готується оцінка результатів тестів та документація про результати випробувань.

Плани, розроблені для тестування модулів, системи, прийняття користувача та інтеграції, виконуються для тестування системи в цілому. Профілювання коду проводиться під час тестування одиниць. Профілювання виявляє випадки, коли нераціональне масштабування алгоритмів, використання ресурсів та інстанцій.

Аналіз і генерація тестових звітів:

Після того, як всі типи тестувань виконались, команда тестування проводить аналіз для виявлення конкретної функціональності або компонентів, в яких виявлені певні проблеми.

Результати, отримані під час аналізу, можуть підтвердити, чи можуть виконані тестові сценарії / процедури виявити помилки.

Це остання фаза життєвого циклу тестування з автоматизації, і на цьому етапі створюються звіти про випробування. Ось чому тестові звіти мають вирішальне значення для аналізу того, наскільки добре ваш вебдодаток реагує на негаразди. Ви можете використовувати старий аркуш програми Excel, однак, сучасні тестові генератори надають звіти про додаток та всі тестові випадки, виконані за допомогою сценарію автоматизації на хмарній сітці Selenium Grid.

На рисунку 2.2 зображена так звана ідеальна піраміда автоматизації тестування.

Почати слід з того, що взагалі робить тести автоматичними. Який саме тест можна назвати автоматичним? Насамперед той, виконання якого не потребує присутності людини. Способів для запуску та налагодження існує велика кількість. Є можливість налаштувати запуск тестів після окремих дій програміста, таких як новий коміт чи просто за певним розкладом.

2.2 Піраміда тестування у автоматизації

Рисунок 2.2 – Піраміда ідеального тестування

На нижньому рівні знаходяться автоматичні модульні тести, ціллю яких являється перевірити що окремі кусочки коду, частіше за все функції, коректно працюють окремо одна від одної. Головна ідея в тому, щоб подальші зміни вже існуючого коду не призвели до поломки вже протестованої частини програми. Як правило, написанням модульних тестів займається сам розробник коду або колега з команди. Час витрачений на один юніт-тест в рази менше ніж на будь який рівнем вище. Таким чином, юніт-тестування - це перший рівень для боротьби з багами. При розробці великих додатків модульні тести відіграють важливу роль. Якщо над

продуктом починає працювати більше ніж одна людина, стає дуже важко вносити зміни у вже існуючий код, не зламавши жодного компоненту системи. На допомогу й приходять автоматичні юніт-тести, які можуть бігти у системі неперервної інтеграції після будь якої зміни у коді.

Але й слід звернути увагу на те, що у випадку коли ви самостійно розроблюєте невеликий додаток, написання модульних тестів може тільки сповільнити і поскладнити сам процес розробки.

Основна різниця між компонентним і модульним тестуванням полягає у тому, що компонентне тестування у якості параметрів функцій використовує об'єкти і драйвери, а у модульному – конкретні значення. [7] В інтеграційних тестах перевіряються програмні модулі, які інтегровані між собою локально і перевіряються як група. Типове ПЗ складається з окремих модулів, створених різними програмістами. Основною ціллю цього рівня є знаходження дефектів під час взаємодії цих частин. Потрібно впевнитись, що навіть у випадку коли кожен модуль ідеально працює окремо, нічого не пошкоджується при їх взаємодії. Можна перевіряти як взаємодію компонентів так і різних систем. Для відокремлення окремих частин системи використовують так звані заглушки, які можуть емулювати або просто припиняти роботу окремих частин вебдодатку. Розділяють основні 3 підходи інтеграційного тестування:

- Знизу вгору (Bottom Up Integration)

Всі низькорівневі процедури, модулі та функції збираються разом і тільки потім тестуються. Після цього береться наступний рівень модулів з метою проведення інтеграційного тестування. Даний підхід буде корисним, якщо практично всі модулі, які повинні бути створені – вже готові. За результатами тестування, такий підхід, може допомогти поступово знаходити вразливості на кожному рівні додатку.

- Зверху вниз (Top Down Integration)

Спочатку тестуються усі високорівневі модулі та поступово один за одним додаються низькорівневі. Всі модулі низького рівня симулюються за допомогою заглушок з аналогічною функціональністю, потім по мірі готовності вони замінюються реальними активними компонентами. Таким чином відбувається тестування зверху вниз.

- Великий вибух ("Big Bang" Integration)

Всі модулі збираються разом у вигляді закінченої системи і вже потім проводиться інтеграційне тестування. Такий підхід дуже добре підходить для збереження часу. Але, якщо тест кейси та їх результати будуть записані не вірно, то сам процес інтеграції може сильно ускладнитись, що стане перешкодою при досягненні основної мети інтеграційного тестування командою тестування.[8]

Прикладний програмний інтерфейс (API – англ. Application Programming Interface) надає можливість здійснювати обмін і зв'язок даними у двох програмних системах. Система ПЗ, що реалізовує API, повинна містити функції, які можуть бути виконані за допомогою іншої системи ПЗ. Щоб програми могли спілкуватися між собою, їх API повинен бути побудований по єдиному принципу. Самий популярний на сьогоднішній день - REST. Це стандарт архітектури взаємодії додатків, який використовує HTTP. Тестування API виконують опираючись на бізнес-логіку програмного продукту. Для його тестування використовують спеціальні інструменти, які дозволяють робити запити і перевіряти достовірність вихідних даних.

Чим вище ми просуваємося по піраміді тестування, тим більших навичок і знань потрібно для реалізації тестування. Тестування API потребує розуміння системи і того як в ній влаштовані запити. Помилки HTTP сервера розподіляють на діапазони:

• 100-199 Інформаційні. Повідомляють що запит прийнятий і обробляється.

• 200-299 Запит оброблений успішно, сервер відправив клієнтові потрібний документ.

• 300-399 Запит змінений і агентові потрібно почати впроваджувати дії, з метою задоволення зміненого запиту.

• 400-499 Проблеми на стороні клієнту.

• 500-599 Серверні помилки

Різниця між модульним тестуванням і тестуванням API наведена у таблиці 2.1

Таблиця 2.1 Порівняльна таблиця API і модульного тестування

Модульне тестування	Тестування API
Зазвичай тести пишуть розробники	Тести пишуть тестувальники, менше розробники
Тестуються окремі методи, функції	Тестується функціональність системи
Розробник може звернутися до програмного коду	Тестувальник, як правило, не може звернутися до програмного коду
Включене тестування з урахуванням графічного інтерфейсу	Тестуються лише функції з API
Тестування лише для базових функцій	Тестуються всі функціональні частини
Застосування обмежене	Широка сфера застосування
Зазвичай проходить під час написання програмного коду	Проходить після створення основного функціоналу

Довгий час SOAP був основним протоколом для обміну даними між веб-службами, але оскільки в наші дні розробникам все більше необхідно створювати компактні та швидкі додатки, більш гнучка REST архітектура швидко завоювала популярність. Розглянемо основні відмінності, двох

архітектур, які зображені у таблиці 2.2. та спробуємо визначити основні переваги та недоліки. [8]

Таблиця 2.2 Порівняльна таблиця REST і SOAP архітектур

	SOAP	REST
Значення	Simple Object Access Protocol	Representational State Transfer
Дизайн	Стандартизований протокол з чіткими заданими правилами оформлення	Архітектурний стил, який просто має свої рекомендації до оформлення
Підхід	Function-driven	Data-driven
Безпечність	WS-Security з підтримкою SSL. Вбудована ACID compliance.	Підтримує HTTPS and SSL.
Формат повідомлень	Тільки XML.	Plain text, HTML, XML, JSON, YAML та інші
Трансферні протоколи	HTTP, SMTP, UDP, and others.	Тільки HTTP
Затратність ресурсів	Велика	Меньша

Обидві архітектури дають можливість створювати власний API. SOAP, ймовірно, продовжить застосовуватися підприємствами, які потребують великого рівня безпеки та мають велику кількість складних транзакцій. Найгарнішими прикладами є банківські установи, системи управління ідентифікацією, платіжні шлюзи, компанії які надають

телекомунікаційні послуги. Навіть одна з найбільших платіжних систем – PayPal, повністю побудована на API-інтерфейсах SOAP. Слід також не забувати і про підтримку вже існуючих систем. У відомих вебдодатків вже може існувати велика кількість користувачів, які все ще підключаються до своїх сервісів через SOAP API. Наприклад, деякі вебдодатки можуть мати частину своєї веб-сервісної архітектори на SOAP, іншу - на REST.

В свою чергу у 2018 році REST став найпопулярнішим вибором розробників для побудови загальнодоступного API. Існує велика кількість соціальних мереж, які мають відкритий REST API-інтерфейс, що дає можливість розробникам дуже легко інтегрувати свої вебдодатки з будь якою платформою. Ці публічні API мають добре описану документацію, звідки можна взяти всю необхідну інформацію.

Підсумовуючи, основними перевагами SOAP є великий рівень безпеки, стандартизованість та розширюваність. З мінусів можна виділити – великий час на виконання, складність і менша гнучкість.

REST, в свою чергу, має кращі можливості для розширення, швидше виконується, більш гнучкий та краще працює з сучасними браузерами. З мінусів можна відзначити – менший рівень безпеки та погану роботу з розподіленими середовищами.

На верхньому рівні піраміди знаходяться UI End-to-End тести. З їх допомогою можна повністю перевірити систему чи сам продукт і бути впевненим, що весь інтерфейс користувача працює коректно. Вони займають найбільшу частину часу і можуть одночасно перевіряти велику кількість компонентів. В нас є можливість перевірити що кожен елемент на сторінці відображається відповідно до вимог, відбуваються переходи між сторінками, розмітка не пливе. Такий тест повністю емульє дії звичайного користувача і може одночасно перевіряти роботу декількох сервісів. Існує велика кількість інструментів, написаних різними мовами програмування, які надають можливість налагодити процес автоматичної перевірки

вебдодатків. Вони запускаються на останньому етапі і можуть гарантувати найбільше покриття і найкращу якість для кінцевого продукту.

E2E тестування – це складна інженерна робота, проведення мостів з датчиками і проведення всіх необхідних перевірок і ситуацій – чи хоча б опис цих сценаріїв. [9]

2.3. Порівняльна характеристика інструментів для програмної реалізації автоматизованого тестування вебдодатку

Приступаючи до автоматизації, кожна команда приймає рішення керуючись потребами замовника або на основі своїх навичок та знань. Зараз з'являється все більше можливостей обирати мову автоматизації виходячи з стека технологій конкретного проекту. Як правило, намагаються обрати мову програмування, яка використовується на Back-End чи Front-End. Це дає можливість залучати більшу кількість членів команди у процес автоматизації, що покращує якість написання коду і надає можливість всім приймати участь у процесах написання та перевірки коду.

Різниця між мовами все більше зменшується і стає непомітною. Тому, переглядаючи сучасні тренди в автоматизації, можна побачити, що все більша кількість проектів використовують JavaScript для різних рівнів і видів тестування. Як приклад можна привести статистику Github за 2019 рік за популярністю мов програмування, рисунок 2.3.[10]. Оскільки ми знаємо що Front-End завжди пишеться за допомогою Javascript / TypeScript, а все більша кількість проектів обирає Node.js як мову для Back-End, у нас є можливість побудувати весь проект лише використовуючи Javascript. Тому для нас вибір на користь Javascript має бути найбільш оптимальним.

Рисунок 2.3 Популярність мов програмування у 2019 році.

Як ми бачимо JavaScript більше ніж в 2 рази попереджає своїх найближчих конкурентів Java та Pyhton, але й слід звернути увагу на вже існуючі рішення на ринку автоматизації. Це теж відіграє важливу роль при виборі відповідної мови.

Jest

Ймовірно, один з найпопулярніших фреймворків тестування javascript, який має понад 22 000 зірок на github, був побудований та постійно підтримується командою з Facebook. Не потребує налаштування конфігурацій, рекомендований від React, і найпростіший у використанні. Jest має дуже вражаючий рівень прийняття у 2019 році спільнотою javascript.

Він має велику швидкість роботи та зручний інтерфейс користувача. Також передбачає тестування знімків і постачається із вбудованим інструментом покриття коду. Це неймовірно швидко і один з найкращих

варіантів для початківців, які хочуть пройти тестування свого коду JavaScript. В Інтернеті також є багато ресурсів та документації [11].

2.3.1 Порівняння JS-фреймворків для Unit-тестування

Рисунок 2.4 Найпопулярніші фреймворки для Unit-тестування, написанні на Javascript / TypeScript

Основні переваги JEST:

▪ Сумісний з NodeJS, React, Angular, VueJS та іншими проектами на основі Babel

Стандартний синтаксис із підтримкою документації

▪ Дуже швидко та високоефективно

▪ Управління тестами з більшими об'єктами можливо за допомогою Live Snapshots

Mocha

MochaJS - це найпопулярніший тестовий фреймворк, який підтримує тестування бекенда та фронтенда. MochaJS - це гнучка база для розробки тестів у міру необхідності. Він запускає тести асинхронно на двигуні Chrome v8 або будь-якому іншому браузері.

Пропонує розробникам лише базову структуру тестування. Функціональність для тверджень, шпигунів, макетів потім додається за допомогою інших бібліотек / плагінів.

Якщо ви хочете гнучку конфігурацію, включаючи бібліотеки, які вам особливо потрібні, то додаткове налаштування та конфігурація, необхідні для Mocha, - це те, що ви обов'язково потрібно перевірити

На жаль, вищезазначений момент має і зворотний бік, який повинен включати додаткові бібліотеки для перевірок. Це означає, що налаштувати трохи складніше, якщо не довше, ніж інші.

Mocha включає структуру тесту як глобальну, що дозволяє економити ваш час.

Гнучкість у твердженнях та заглушках дуже корисна. В цілому Mocha охоплює основи і дозволяє розробникам розширювати їх.

До основних переваг Mocha можна віднести:

• Працює як для фронтенду, так і для бекенда

• Підтримка відладчика NodeJS

Забезпечує чисту базу для розробки тестів відповідно до зручності розробника

• Підтримується будь-який веб-переглядач, включаючи бібліотеку для Google Chrome

• Підтримує заглушку з об’єктів для виконання запуску гнучких тестів

Jasmine

Jasmine - надає вам все, що вам потрібно з коробки. Це імітатор поведінки користувачів, який дозволяє виконувати тестові випадки, подібні до поведінки користувачів на вашому веб-сайті. Jasmine корисний для тестового інтернету на предмет видимості, чіткості натискань, а також чуйності інтерфейсу користувача в різних режимах дозволу. Jasmine дозволяє автоматизувати поведінку користувачів з мінімальними затримками та зачекат, щоб імітувати фактичну поведінку користувача.

Основні переваги використання Jasmine:

• Нижні накладні витрати через майже нульові зовнішні залежності

• Поставляється майже з кожним необхідним інструментом поза коробкою

• Підтримує тести написані для фронтенду, а також тести бекенду

• Написання коду досить схоже на написання натуральною мовою

• Обширна документація по використанню з іншими фреймворками

• Велика кількість ознайомчих онлайн курсів

AVA

Це мінімалістичний фреймворк тестування легкої ваги, яка використовує асинхронний характер Javascript. Він в основному зосереджений на запущених тестах для коду на основі NodeJS. AVA може виконувати тести паралельно.

Це дозволяє вам майже повний контроль над тим, що ви робите. Він в основному зосереджений на запущених тестах для коду на основі NodeJS. Деякі з переваг включають:

• Тести виконуються асинхронно та одночасно

• Швидше, ніж більшість інших тестових рамок

• Простіший синтаксис тестів Javascript

• Очисник стеку відстежує будь-які виявлені потенційні помилки

Karma

Це продуктивне середовище для тестування, яке підтримує всі популярні фреймворки опису тестів всередині себе. Він надає вашій програмі підтримку виконання тестів у різних середовищах. Він має широку підтримку виконання тестів на різних пристроях та додатках.

Основний чинник вибору Karma полягає в його підтримці інтеграції з двигунами CI / CD та наступними особливостями:

• Можна використовувати для запуску тестів на браузерах, безголових середовищах, таких як PhantomJS, а також на пристроях

• Підтримує тести, написані в більшості популярних фреймворків

• Дозволяє дистанційно запускати тести на інших пристроях, просто надсилаючи файли

Підтримує налагодження тестових випадків за допомогою Chrome, а також WebStorm

2.3.2. Порівняння JS-фреймворків для End-To-End-тестування

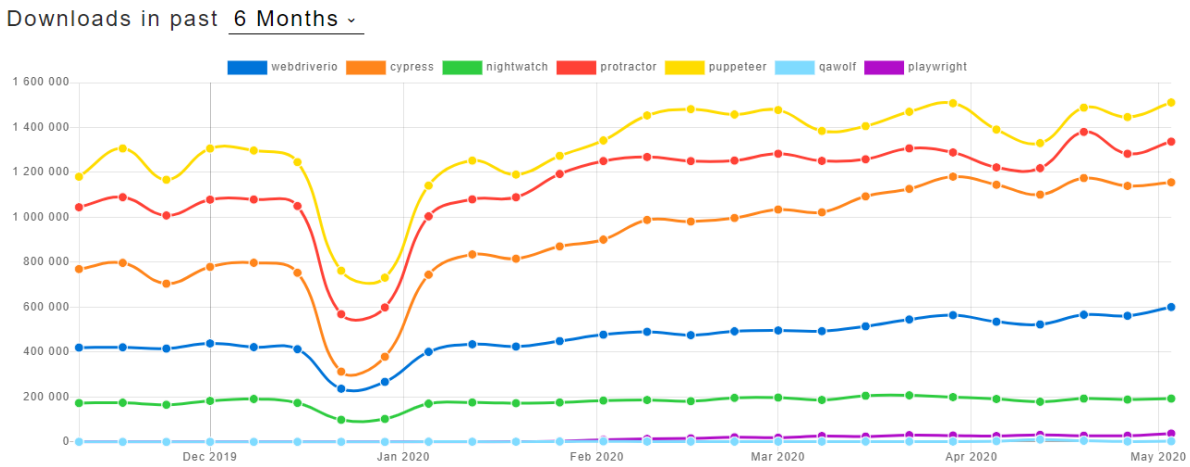


Рисунок 2.5 Статистика скачувань фреймворків для End-To-End-тестування, написаних на Javascript / TypeScript за грудень 2019 – травень 2020 років

При виборі фреймворка для тестування, слід звернути увагу на його популярність на GitHub, підтримку від товариства, кількість відкритих проблем і сумісність з технологіями, які використовуються у вас в системах, а також на особливостях кожної програми, використовуваної для автоматизації. [13]

Protractor

Добре підходить для проектів, написаних на Angular, для яких є встроєні функції для конкретних завантажувачів Angular-елементів

Плюси:

• Protractor - це єдиний фреймворк, який із коробки підтримує кастомні визначення AngularJS-елементів. Якщо у вас Angular, використовуйте Protractor.

• Зручна підтримка TypeScript та різних фреймворків для тестування одиниць (Jasmine, Mocha, Cucumber та інші).

• Велика кількість допоміжних функцій

	stars	forks	issues	updated	created	size
webdriverio	5 675	1 663	119	May 15, 2020	Aug 30, 2011	minzipped size 238.4 KB
cypress	20 103	1 205	1 249	May 15, 2020	Mar 4, 2015	minzipped size 365.2 KB
nightwatch	10 194	999	120	May 11, 2020	Mar 17, 2012	
protractor	8 542	2 370	610	May 14, 2020	Jan 16, 2013	bundlephobia timeout
puppeteer	61 323	6 323	1 032	May 14, 2020	May 10, 2017	minzipped size 50.8 KB
qawolf	1 324	35	15	May 14, 2020	Sep 17, 2019	bundlephobia timeout
playwright	12 631	365	119	May 15, 2020	Nov 15, 2019	minzipped size 117.3 KB

Рисунок 2.6 Інформація щодо актуальності і підтримки фреймворків для End-To-End-тестування, написаних на Javascript / TypeScript

Мінуси:

• Нема підтримки мобільного тестування.

• Написаний як обертка над WebDriverJS. Якщо у вас будуть проблеми з WebDriverJS, вони автоматично будуть в ProtractorJS.

• Зупинилася підтримка від розробників

Базовий приклад тесту на ProtractorJS:

```
// spec.js
describe('Protractor Simple test', function() {
it('should enter two numbers', function() {
browser.get('https://www.madewithangular.com/');
element(by.model('one')).sendKeys(1);
element(by.model('two')).sendKeys(2);
element(by.id('enter')).click();
expect(element(by.binding('latest')).getText()).
toEqual('10'); // That is wrong!
```

```
});
});
```

Nightwatch.js

Плюси:

• Схожий з WebdriverIO, і він є кастомною імплементацією W3C WebDriver API.

• Легко додати нову функцію.

• Не потрібно вибирати між Jasmine і Mocha.

Мінуси:

• Нема підтримки Mocha та мобільного тестування.

• Менше підтримують, ніж у WebdriverIO та Cypress.

Базовий приклад тесту на Nightwatch.js:

```
module.exports = {
after: function(browser) {
console.log(Browser loaded.')
},
'Nightwatch simple test': function (browser) {
browser
.url('https://nightwatch.netlify.com/')
.waitForElementVisible('[data-nw=name-input]')
.setValue('[data-nw=name-input]', 'Text you want to enter')
.pause(1000)
.assert.containsText('[data-nw=welcome-message]', 'Welcome your text')
.end()
}
```

Cypress

Плюси:

• На відмінність від більшості E2E - фреймворків, не використовує Selenium. Архітектурна роботи з браузером написана повністю та Cypress не використовує віддалених команд через мережу, а працює напряму з програмою.

• Швидке і зручне налаштування. Необхідно залишити лише одну команду для встановлення всіх пакетів, і немає необхідності ставити все окремо.

• Будь-яка документація в одному місці, не потрібно окремо шукати, як правильно робити перевірку або як використовувати Chai.

• Необхідний інтерфейс використовується з описом, які команди виконуються, що краще показує, що відбувається на сторінках у відповідний момент.

Мінуси:

• Плюс, який в той же час є і мінусом, - це той факт, що Cypress не використовує Selenium для end-to-end-тестування. Що породжує багато проблем в роботі. Робота з браузером не завжди стабільна, і на сьогоднішній день на GitHub є відкритими більш ніж 900 проблем.

• Немає підтримки мобільного тестування, і, судячи з коментарів творців нативной підтримки, ніколи і не буде.

• Підтримка обмеженої кількості браузерів: Canary, Chrome, Chromium, Electron.

• Платна паралелізація тестів. Хочете запускати в кілька потоків - доведеться платити.

```
describe(Cypress simple Test', () => {
it('clicking "type" navigates to a new url', () => {
cy.visit('https://example.cypress.io')
cy.contains('button').click()
// should be redirected to a new url
cy.url().should('include', '/commands/actions')
})
```

```
})
```

WebdriverIO

Плюси:

• WebdriverIO - це кастомна імплементація W3C WebDriver API. Не потрібно прив'язуватися до імплементації WebDriverJS.

• Підтримка синхронного коду. Можна забути про асинхронний JavaScript.

• Зручне базове налаштування за допомогою вбудованого інтерфейсу для командного рядка wdio.

• Підтримує майже всі тестові фреймворки BDD і TDD.

• Підтримує зручну бібліотеку 'webdrivercss' для порівняння CSS-стилів елементів на сторінці.

Мінуси:

• Велика відмінність між останніми версіями (WebdriverIO 4 і 5).

• Гірше для автоматизації AngularJS, ніж Protractor.

```
describe('Simple WebdriverIO Test', () => {
beforeEach(async(() => {
TestBed.configureTestingModule({
declarations: [
AppComponent
],
}).compileComponents();
}));
it('should add the app', () => {
const myFixture = TestBed.createComponent(AppComponent);
const application = myFixture.componentInstance;
expect(application).toBeTruthy();
});
```

Отже, перелічимо переваги і недоліки використання JavaScript-фреймворків для автоматизації тестування, які ми відзначили вище.

Переваги:

▪ Швидкість написання тестів значно вище, ніж на Java або C #.

▪ Нижче поріг входу для старту проекту.

▪ Більше взаємодії всередині, якщо члени команди володіють однією мовою програмування.

▪ Велика кількість готових рішень дуже різних проблем, які виникають.

Недоліки:

▪ Менш стабільні рішення. Іноді може вийти так, що проблема є на стороні фреймворку.

▪ Для написання стабільних тестів потрібно розуміння, як працює JavaScript.

У кожного рішення / мови / технології є свої переваги і свої недоліки. З вибором технологічного стека визначатися вам. Цей вибір залежить від складу вашої команди, від знань і досвіду її учасників, а також від завдань, які ви хочете вирішити. Якщо умови дозволяють, обов'язково дивіться в бік технологій, які активно розвиваються.

РОЗДІЛ ІІІ. РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ ТА АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ

3.1. Вибір мови програмування та бібліотек

Автоматизоване тестування (АТ) найбільше ефективно, коли реалізовано за допомогою фреймворку. Незважаючи на те, що в АТ термін фреймворк найчастіше використовується для опису сукупності об'єктів, що формує інструмент модульного тестування, тут буде здебільшого сфокусована на фреймворках іншого роду.

Розглянемо типи фреймворків, які можуть бути визначені як сукупність абстрактних понять, процесів, процедур та середовищ, за допомогою яких автоматичні тести проектуються, створюються та реалізуються. Крім того, це визначення фреймворку включає фізичні об'єкти, що використовуються для створення тестів та їх реалізації, а також для організації логічної взаємодії між компонентами.

Автоматизоване тестування (і, отже, фреймворки) розвивалося роками, формуючи та ускладнюючись з кожною новою фазою еволюції. Ці фази можуть бути описані в термінах трьох поколінь, кожне з яких має набір недоліків і переваг, завдяки яким кожне з них залишається актуальним, незважаючи на нові розробки.[9]

Сервіси, які необхідно протестувати можна написати на будь-якій мові програмування(МП), наприклад на Java, Python, C#, Assembly, Scala або будь-якою іншою доступною мовою програмування.

Фреймворки для автоматизованого тестування на Java автоматизуватимуть ручні тести нічим не гірші за фреймворки на Python. Коли мова заходить про автоматизацію веб-сервісів, немає різниці на якому фреймворку буде проводитись тестування. Яка МП підійде залежить лише від ситуації.

Розберемо плюси та мінуси тих та інших фреймворків.

Плюси Python:

важко стати на шлях тестувальника, не знаючи Python. Якщо і є така мова, яку потрібно знати, щоб вміти автоматизувати все, то це безумовно Python. Ви можете автоматизувати розгортання оточення, використовувати його для сканування портів або проводити тестування на безпеку;

у порівнянні з навантаженим синтаксисом Java його до дуже просто використовувати та читати. Крім того, відомий факт, що на один рядок на Python припадає 10 рядків на Java;

багато інших людей використовують Python з тієї ж причини, що і ви, з цього можна зробити висновок, що хтось вже написав код, який вам потрібен, а ви можете його імпортувати;

навчання та підтримка. В Інтернеті можна знайти купу навчальних та корисних матеріалів. В цілому, люди прийшли до думки, що матеріали з Python зрозуміліші, ніж з будь-якої іншої мови.

Мінуси Python:

python створювався, щоб бути простим, універсальним і давати можливість писати скрипти прямо з інтерпретатора, тому він не так добре працює з IDE, як та ж Java. Він настільки простий та універсальний, що IDE не може зрозуміти, що ви робите, коли починаєте створювати об'єкти та передавати їх між методами. Це неприємна особливість, яка може зіграти свою роль, якщо ви дійсно захочете використати IDE для створення свого фреймворку;

підтримка в офісі. Іноді краще мати локальну підтримку. Якщо у вас в колективі ніхто не знає Python, то отримати невідкладну допомогу у вирішенні питань буде не від кого;

Плюси Java:

чудові IDE. Одне задоволення писати на Java у такому середовищі, як, наприклад, IDE від IntelliJ. IDE виконує за вас більшу частину роботи, навіть беручи проблеми від складного синтаксису. Функції авто доповнення

коду зроблять за вас величезну кількість роботи, поки вам буде здаватися, що ви набрали на клавіатурі всього пару символів;

pageFactory. PageFactory Java спрощує код для автоматизації на Selenium і дозволяє писати прості для розуміння тести; більшість тестувальників працює з Java-розробниками. Якщо ви з чимось застрягнете, через пару столів від вас завжди буде хтось, хто простягне вам руку допомоги. Це дуже допомагає на лінії навчання, і дає вам переваги знань та досвіду колег. Не встигнете озирнутися, як станете професіоналом;

Мінуси Java:

непросто читати код Java в порівнянні з простим англійським Python. А ще у Java дуже крута крива навчання, і документація виявляється не завжди корисною. Однак допомогу з багатьох питань можна знайти онлайн (наприклад, StackOverflow);

проблема з null pointer. Коли Java видає вам повідомлення про помилку і виводить stack trace не завжди просто зрозуміти, в чому справа, і часом ця інформація виявляється зайвою. IntelliJ допомагає там, де може, але незрозумілі повідомлення про помилки можуть перетворити дебаг на розлад.[10]

Основною мовою для розроблення додатку було обрано Java. Об’єктноорієнтована мова Java, розроблена в компанії Sun Microsystems в 1995 році для відображення графіки на стороні клієнта за допомогою аплетів, в даний час використовується для створення незалежних від операційних систем програм.

Мова Java знайшла широке застосування в Інтернет-додатках, додавши на статичні і клієнтські веб-сторінки динамічну графіку, поліпшивши інтерфейси і реалізувавши обчислювальні можливості. Але об’єктно-орієнтована парадигма і незалежність від операційної системи привели до того, що вже буквально через кілька років після створення мову практично повністю покинула клієнтські сторінки і перебралась на сервери.

На стороні клієнта його місце зайняли мови JavaScript, Adobe Flash і ін.При створенні мови Java, розробники намагалися зробити її більш простою, ніж її предок C ++. Сьогодні з появою нових версій можливості мови Java істотно розширилися і багато в чому перекривають функціональність C ++. Java вже не поступається за складністю попередникам і називати її простою мовою неможливо. Відсутність вказівників (найбільш небезпечного засобу мови C ++) не можна вважати звуженням можливостей, а тим більше - недоліком, це просто потреба безпеки. Можливість роботи з довільними адресами пам'яті через ігнорування типів дозволяє ігнорувати захист пам'яті.

Відсутність в Java множинного успадкування легко замінюється на більш зрозумілі конструкції із застосуванням інтерфейсів. Системна бібліотека класів мови Java містить класи і пакети, реалізуючи і розширюючи базові можливості мови, мережева взаємодія, взаємодія з базами даних, графічні інтерфейси і багато іншого.

Методи класів, включених в ці бібліотеки, викликаються JVM (Java Virtual Machine) під час інтерпретації програми. 56 У Java всі об'єкти програми розташовані в динамічній пам'яті - купі даних (heap) і доступні за об'єктним посиланнях, які, в свою чергу, зберігаються в стеку (stack). Це рішення виключило безпосередній доступ до пам'яті, але ускладнило роботу з елементами масивів і зробило її менш ефективною в порівнянні з програмами на C ++. У свою чергу, в Java запропоновано удосконалений механізм роботи з колекціями, реалізовані основні динамічні структури даних.

Необхідно відзначити, що об'єктна посилання мови Java містять інформацію про клас об'єкту, на котрий вона посилається, так що об'єкт на посилання - це не покажчик, а дескриптор (опис) об'єкта. Наявність дескрипторів дозволяє JVM виконувати перевірку сумісності типів на фазі інтерпретації коду, генеруючи виключення в разі помилки.

В Java змінена концепція організації динамічного розподілення пам’яті: відсутні способи програмного очищення динамічно виділеної пам’яті. Замість цього реалізована система автоматичного звільнення пам’яті (збирач сміття), виділеної за допомогою оператора new. Програміст може тільки рекомендувати системі звільнити виділену динамічну пам’ять. На відміну від C ++, Java не підтримує множинне успадкування, перегрузку операторів, беззнакові цілі, пряме індексування пам’яті і, як наслідок, покажчики.

В Java існують конструктори, але відсутні деструктори (застосовується автоматичне прибирання сміття), не використовується оператор goto і слово const, хоча вони є зарезервованими словами мови.

3.2. Середовище розробки IntelliJ IDEA

Для розробки додатку було використано платформу IntelliJ IDEA - це інтегроване середовище розробки Java (IDE) для розробки комп'ютерного програмного забезпечення. Редактор IntelliJ IDEA є особливим.

Найбільш помітним є те, що ви можете запускати практично будь-яку функцію IDE, не залишаючи її, що дозволяє вам організувати макет, де ви маєте більше місця на екрані, оскільки приховані допоміжні елементи керування, такі як панелі інструментів та вікна.

IntelliJ IDEA підтримує пробні тести на платформі JUnit з версії 2016.2. Однак слід зазначити, що рекомендується використовувати IDEA 2017.3 або новішу версію, так як ці нові версії IDEA буде завантажити наступні JARs автоматично на основі версії API , що використовується в проекті: junit-platform-launcher, junitjupiter-engine і junit-vintage-engine.

IntelliJ IDEA випускає до випуску специфічних версій JUnit 5 для пакету IDEA 2017.3.

Отже, якщо ви хочете використовувати нову версію JUnit Jupiter, виконання випробувань в середовищі IDE може виникнути з-за конфліктів версій. У таких випадках, будь ласка, дотримуйтесь наведених нижче

інструкцій, щоб використовувати нову версію JUnit 5, ніж таку, що входить до комплекту IntelliJ IDEA.

Для того , щоб використовувати іншу версію JUnit 5 (наприклад, 5.3.2), буде потрібно включити відповідні версії junit-platform-launcher, junit-jupiter-enginei junitvintage-engine JARs в шляху до класів.[7]

Розробка великих програмних продуктів майже неможлива без застосування спеціальних інтегрованих середовищ розробки (IDE), що значно спрощують роботу з проектом. Окрім можливостей звичайного текстового редактору, IDE спрощують роботу з системами контролю версій, комунікацію з базою даних, інтеграцію з 54 фреймворками та інструментами для збірки та тестування систем.

Для Java існує три основні IDE: IntelliJ IDEA, NetBeans та Eclipse. NetBeans та Eclipse є безкоштовними та підтримують майже всі основні фреймворки та бібліотеки.

Середовище IntelliJ IDEA(розширене) є платним та, окрім базових можливостей інших IDE, є більш стабільним, має кращі механізми пошуку у проекті, широкий набір додаткових якісних плагінів та зручні методи для рефакторингу коду. Для студентів є можливість отримати IntelliJ IDEA абсолютно безкоштовно.

Саме тому весь код був написаний у інтегрованому середовище розробки IntelliJ IDEA для різних мов програмування від компанії JetBrains.

До основних можливостей IntelliJ IDEA відносяться:

— розумне автодоповнення, інструменти для аналізу якості коду, зручна навігація, розширені рефакторингом і форматування для Java, Groovy, Scala, HTML, CSS, JavaScript, CoffeeScript, ActionScript, LESS, XML і багатьох інших мов;

— підтримка всіх популярних фреймворків і платформ, включаючи Java EE, Spring Framework, Grails, Play Framework, GWT, Struts, Node.js, AngularJS, Android, Flex, AIR Mobile і багатьох інших;

— інтеграція з серверами додатків, включаючи Tomcat, TomEE, GlassFish, JBoss, WebLogic, WebSphere, Geronimo, Resin, Jetty і Virgo;

— інструменти для роботи з базами даних і SQL файлами, включаючи зручний клієнт і редактор для схеми бази даних;

— інтеграція з комерційними системами управління версіями Perforce, Team Foundation Server, ClearCase, Visual SourceSafe;

— інструменти для запуску тестів і аналізу покриття коду, включаючи підтримку всіх популярних фреймворків для тестування.

До складу IntelliJ IDEA входить модуль візуального проектування програм GUI-інтерфейсу Swing UI Designer, XML-редактор, редактор регулярних виразів, система перевірки коректності коду, система контролю за виконанням завдань і доповнення для імпорту та експорту проектів з Eclipse.

Доступні засоби інтеграції з системами відстеження помилок JIRA, Trac, Redmine, Pivotal Tracker, GitHub, 55 YouTrack, Lighthouse. Під час розробки даного програмного забезпечення було використано 2018.1 версію продукту.

Бібліотека JUnit

JUnit 5 - це наступне покоління JUnit. Метою є створення новітньої основи тестування на JVM на стороні розробників. Це включає в себе зосередження уваги на Java 8 і вище, а також увімкнення багатьох різних стилів тестування.

На відміну від попередніх версій JUnit, JUnit 5 складається з декількох різних модулів з трьох різних підпроектів.

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

Junit Platform платформа служить основою для запуску механізмів тестування на JVM. Він також визначає TestEngine API для розробки тестової структури, що працює на платформі.

Крім того, платформа надає консольний запуск, щоб запустити платформу з командного рядка та створювати плагіни для Gradle і Maven , а також на базі JUnit 4 Runner для роботи TestEngine на платформі.

JUnit Jupiter - це комбінація нової моделі програмування та моделі розширення для написання тестів і розширень у JUnit 5.

Підпроект Jupiter передбачає TestEngine тестування на платформі на базі Jupiter. JUnit Vintage забезпечує TestEngine тестування JUnit 3 та JUnit 4 на платформі. JUnit 5 вимагає Java 8 (або вище) під час виконання. Тим не менш, ви все ще можете перевірити код, який був скомпільований з попередніми версіями JDK.

Платформа JUnit (JUnit Platform) має наступні ідентифікатори артефактів:

• junit-platform-commons - внутрішня спільна бібліотека / утиліти JUnit. Ці утиліти призначені виключно для використання в рамках самого JUnit. Проте будь-яке використання зовнішніх сторін не підтримується.

• junit-platform-console - підтримка відкриття та виконання тестів на платформі JUnit з консолі.

• junit-platform-console-standalone - виконуваний JAR файл з усіма включеними залежностями надається у Maven Central у каталозі junitplatform-console-standalone.

• junit-platform-engine - публічний API для тестових движків(TestEngine).

• junit-platform-launcher - Загальнодоступний API для налаштування та запуску планів тестування - зазвичай використовуються IDE та інструментами для створення.

• junit-platform-runner - Runner для виконання тестів і тестових наборів на платформі JUnit у середовищі JUnit 4.

• junit-platform-suite-api - анотації для налаштування тестових наборів на платформі JUnit. Підтримується Engine'ом JUnitPlatform і сторонніми TestEngine реалізаціями. 58

• junit-platform-surefire-provider - підтримка відкриття та виконання тестів на платформі JUnit за допомогою Maven Surefire А такі артефакти має в собі JUnit Jupiter:

• junit-jupiter-api - API JUnit Jupiter для написання тестів та розширень.

• junit-jupiter-engine - JUnit Jupiter - реалізація тестового движка TestEngine, потрібна лише під час виконання.

• junit-jupiter-params - підтримка параметризованих тестів у JUnit Jupiter.

• junit-jupiter-migration-support - підтримка міграції від JUnit 4 до JUnit Jupiter потрібна лише для роботи вибраних правил JUnit 4.

JUnit Vintage має єдиний ідентифікатор - junit-vintage-engine - реалізація тестового двигуна JUnit Vintage, яка дозволяє запускати тести Junit, тобто тести, написані у стилі JUnit 3 або JUnit 4, на новій платформі JUnit. JUnit Jupiter підтримує наступні анотації для налаштування тестів та розширення основ.

Всі основні анотації розташовуються в org.junit.jupiter.api пакеті в junit-jupiter-api модулі.

Анотація	Опис
@Test	Позначає той метод, що є методом для перевірки. На відміну від @Test анотації JUnit 4 , ця примітка не оголошує жодних атрибутів, оскільки тестові розширення в JUnit Jupiter працюють на основі власних анотацій. Такі методи успадковуються, якщо вони не перевизначені.

@ParameterizedTest	Позначає, що метод є параметризованим тестом . Такі методи успадковуються, якщо вони не перевизначені.
@RepeatedTest	Позначає, що метод є тестовим шаблоном для повторного тесту. Такі методи успадковуються, якщо вони не перевизначені.
@TestFactory	Позначає, що метод є тестовою “фабрикою” для динамічних тестів. Такі методи успадковуються, якщо вони не перевизначені.
@TestInstance	Використовується для налаштування життєвого циклу тестового інстансу для анотованого тестового класу. Такі анотації успадковуються.
@TestTemplate	Позначає, що метод являє собою шаблон для тестових випадків, призначених для виклику кілька разів залежно від кількості контекстів викликів, що повертаються зареєстрованими провайдерами. Такі методи успадковуються, якщо вони не перевизначені.
@DisplayName	Оголошує відображуване ім'я для класу тесту або методу перевірки. Такі анотації не успадковуються.
@BeforeEach	Означає , що анотований метод повинен бути виконаний перед кожним @Test , @RepeatedTest, @ParameterizedTest, або @TestFactoryуметодом, в поточному класі; аналогічний до анотації у JUnit 4 - @Before. Такі методи успадковуються, якщо вони не перевизначені.
@AfterEach	Означає, що анотований метод повинен бути виконаний після кожного @Test , @RepeatedTest, @ParameterizedTest,

	або @TestFactoryметод , в поточному класі; аналогічний до JUnit 4 @After. Такі методи успадковуються, якщо вони не перевизначені.
@BeforeAll	Означає, що анотований метод повинен бути виконаний перед усіма @Test , @RepeatedTest, @ParameterizedTest і @TestFactory методами в поточному класі; аналогічно до JUnit 4 @BeforeClass. Такі способи успадковуються (якщо вони не приховані або не змінені), і вони повинні бути static (за винятком використання життєвого циклу тесту "per-class").
@AfterAll	Означає, що анотований метод повинен бути виконаний після всіх інших, що мають @Test , @RepeatedTest, @ParameterizedTesti @TestFactoryметоди в поточному класі; аналогічно до JUnit 4 @AfterClass. Такі способи успадковуються (якщо вони не приховані або не змінені), і вони повинні бути static (за винятком використання життєвого циклу тесту "на клас").
@Nested	Позначає, що клас анотованих є вкладеним, нестатичним тестовим класом. @BeforeAll і @AfterAll методи не можуть бути використані безпосередньо в @Nested класі тесту. Такі анотації не успадковуються.
@Tag	Використовується для оголошення тегів для тестування фільтрів на рівні класу або методу; аналогічні тестовим групам в TestNG або категорії в JUnit 4. Такі анотації успадковуються на рівні класу, але не на рівні методу.
@Disabled	Використовується для вимкнення класу тестів або методу

	тестування; аналогічно в JUnit 4 є @Ignore. Такі анотації не успадковуються .
@ExtendWith	Використовується для реєстрації власних розширень. Такі анотації успадковуються.

Методи анотовані @Test, @TestTemplate, @RepeatedTest, @BeforeAll, @AfterAll, @BeforeEach, @AfterEach анотаціями не повинні повертати значення. 61 Навіть якщо твердження можливості , що надаються JUnit Юпітеру досить для багатьох сценаріїв тестування, бувають випадки , коли більше потужності і додаткові функціональні можливості, такі як matchers бажані або необхідні. У таких випадках команда JUnit рекомендує використовувати сторонні бібліотеки затвердження, такі як AssertJ , Hamcrest , Truth та ін. Розробники можуть вільно використовувати бібліотеку затвердження за їх вибором. Наприклад, комбінацію збірок і вільного API можна використовувати для того, щоб зробити твердження більш описовими та зрозумілими. Тим не менше, org.junit.jupiter.api.Assertions клас JUnit Jupiter не надає такого assertThat() методу, як у org.junit.Assert класі JUnit 4, який приймає Hamcrest Matcher. Замість цього розробникам пропонується використовувати вбудовану підтримку збірок, що постачаються сторонніми бібліотеками затвердження.

Наступний приклад демонструє, як використовувати assertThat() підтримку Hamcrest в тесті JUnit Jupiter. Поки бібліотека Hamcrest була додана в шлях до класів, ви можете статично імпортувати такі методи, як assertThat(), is(), і , equalTo(), а потім використовувати їх в тестах.

```
class HamcrestAssertionDemo {
@Test
void assertWithHamcrestMatcher() {
        assertThat(2 + 1, is(equalTo(3)));
```

}

}

3.3 Побудова фреймворку для автоматизації тестування веб-сервісу

Вибір інструменту для управління та складання проекту У світі Java є три основні інструменти складання: Ant, Maven та Gradle. Після декількох років розробки Ant майже зник, Maven також занепадає, а розробка Gradle бурхливо розвивається. Зараз відбувається спад популярності Maven та hike Gradle.

Основні функції Maven в основному розділені на 5 пунктів: система управління залежностями, багатомодульна побудова, узгоджена структура проекту, узгоджена модель побудови та механізм модулів, що підключаються. Maven та Gradle мають свої переваги у використанні та використовують їх відповідно до сценарію використання. Maven необхідно запровадити залежність pom.xml (див. рис. 3.1).

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.5.RELEASE</version>
</dependency>
```

Рисунок 3.1 Залежності pom.xml Для роботи з Gradle необхідно запровадити build.gradle (див. рис. 3.2).

```
implementation 'org.springframework.boot:spring-boot-starter-web'
```

Рисунок 3.2 Залежності build.gradle

Серед переваг Gradle можна виділити те, шо він має еквівалентні комбінації Maven та Ant. А серед недоліків те, що для посилань на підкласи мікросервісів та мультипроектів він не так гарно структурований, як Maven.

Архітектура проекту з Maven:

структура/залежність проекту визначається файлом pom.xml;

виробничий код зберігається в src/main/java;

тестовий код зберігається в src/test/java;

процес складання та життєвий цикл:

о три стандартні життєві цикли (життєвий цикл);

о найменша одиниця операції – це мета;

о плагіни можуть пов'язувати свої цілі із певною фазою життєвого циклу;

управління пакетами та транзитивні залежності:

о унікальні координати пакета визначаються groupId/artifactId/version;

о посилання надходить із центрального складу.;

о транзитивна залежність;

о коли використання пакета залежить від інших пакетів, Maven автоматично імпортує всі залежні пакети; коли є конфлікт залежностей Maven покладається на демодуляцію, щоб слідувати двом принципам:

о Найближчий шлях;

порядок визначення;

Рисунок 3.3 Конфлікт залежностей Maven

Архітектура проекту з Gradle:

- структура / залежність проекту визначаються build.gradle;
- виробничий код зберігається в src/main/java;
- тестовий код зберігається в src/test/java;

процес складання та життєвий цикл:

о життєвий цикл не показаний;

о найменша діюча одиниця - це завдання, і завдання можуть залежати один від одного;

о може динамічно створювати завдання;

управління пакетами та транзитивні залежності:

о використовуйте компонентну систему Ivy, яка є розширеним набором компонентної системи Maven;

о ant ivy - більший склад, ніж склад Maven;

о сумісний з репозиторіями Maven;

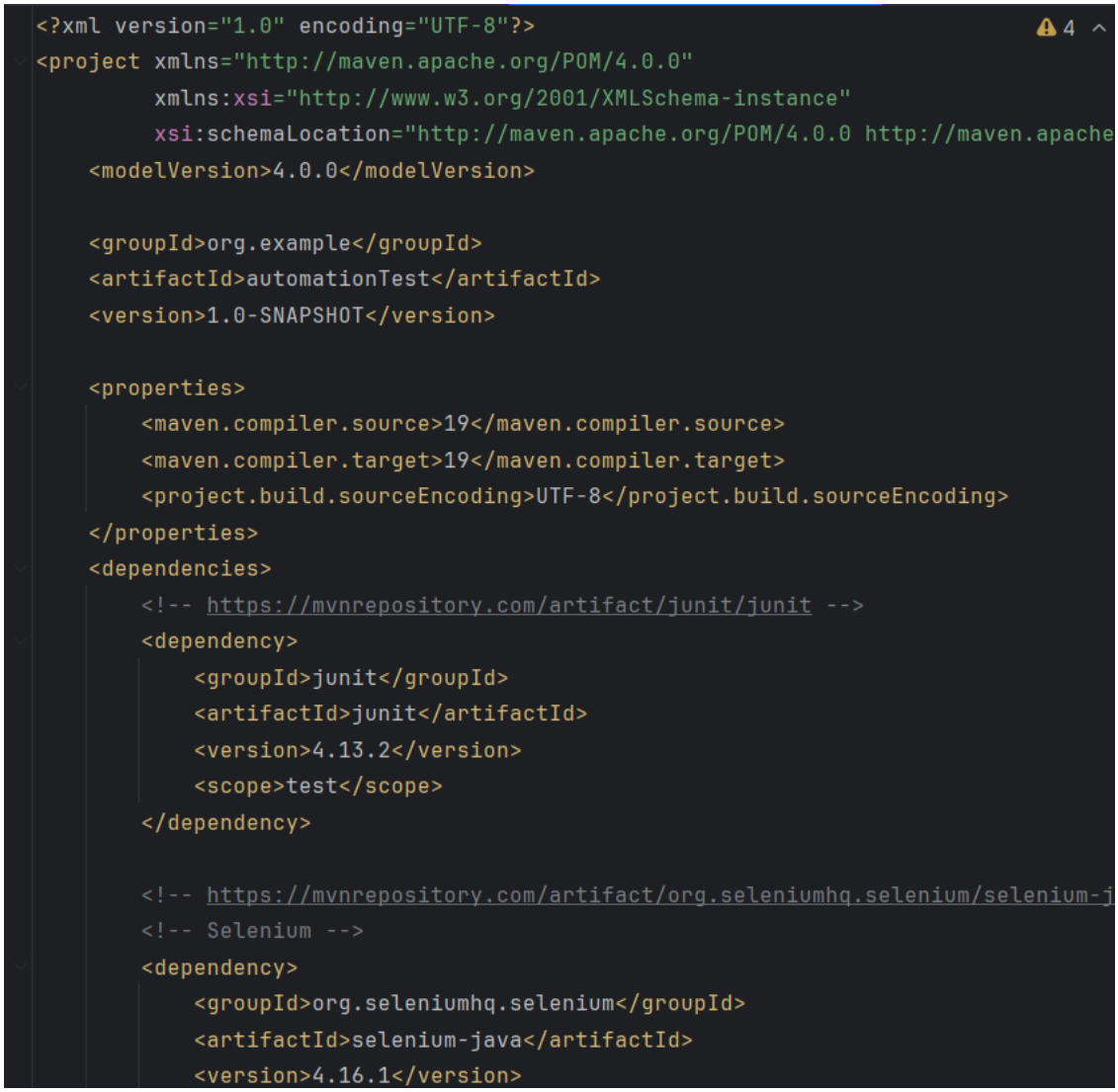
коли є конфлікт залежностей:

о вирішення конфліктів Gradle не є обов'язковим Нова версія (Нові версії зазвичай сумісні);

Підбиваючи підсумки, доцільно дати коротку характеристику фреймворкам.

Maven - стабільний та надійний, з безліччю плагінів. (Протягом багатьох років версія підтримувалась на рівні 3.XX, і лише невелике оновлення було випущено протягом тривалого часу, що вказує на її стабільність та меншу кількість помилок);

Трохи багатослівний, логіка, що настроюється, більш проблематична (Maven використовує xml для конфігурації, недоліки xml громіздкі і будуть відображені в Maven(див. рис. 3.4));



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>automationTest</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>19</maven.compiler.source>
        <maven.compiler.target>19</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-j
        <!-- Selenium -->
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>4.16.1</version>
```

Рисунок 3.4 Xml файл

У великомасштабних проектах поступово виникатимуть проблеми із продуктивністю; проекти, створені за допомогою Maven, відбуваються через кілька життєвих процесів. Механізму внутрішнього хешування немає, і що довше проект буде перекомпонований, то більше часу це займе;

Оскільки розробка Maven в основному залежить від підтримки спільноти, більше немає засобів для продовження розробки та підтримки Maven, що призводить до припинення розробки;

Gradle: gradle використовує логіку коду для побудови, що робить його більш гнучким;

Всередині Gradle є механізм хешування (коли введення та виведення файлу не змінилися, вважається, що це незмінний код, а висновок виконується безпосередньо. Але коли ви змінюєте залежну версію пакета, вона іноді не оновлюється, що також є проблемою з механізмом хешування), який буде працювати швидше;

Фктивна розробка, надто багато версій;

Отже, в даній роботі буде використовуватись Maven через свою надійність та гнучкість.

3.3.1 Підключення бібліотек до фреймворку на основі Maven

Бібліотека в розумінні Maven є артефактом, необхідним для складання програми. У кожного артефакту, як зазначалося вище, є groupId, artifactId та version. Потрібно лише вказати maven, що цей артефакт є залежністю проекту.

Список залежностей повинен бути обернутий тегом залежностей. Кожна окрема залежність повинна бути обгорнута тегом.

Всередині тегу dependency в тегах groupId, artifactId і version необхідно вказати значення відповідних параметрів.

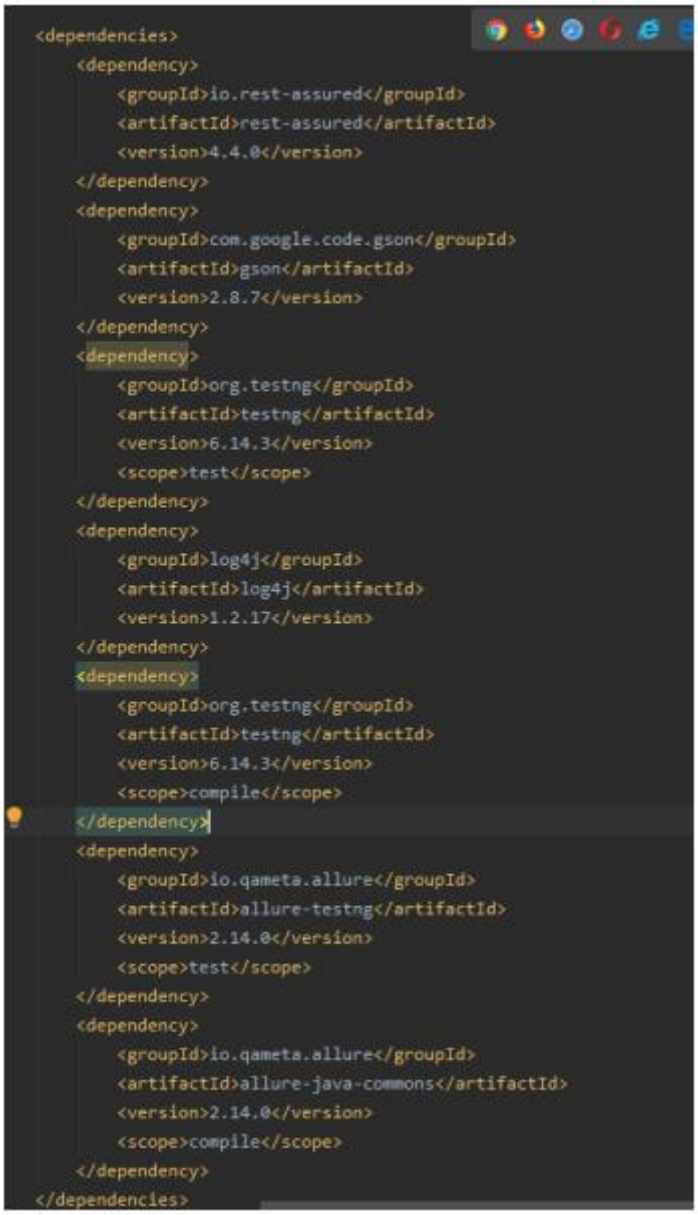


Рисунок 3.5 Список підключених бібліотек

Вижче на рисунку 3.5 показано як і які були підключені бібліотеки для проекту. Залежності підключеного до проекту артефакту maven знайде сам, сам складе список всіх бібліотек, необхідних проекту, включаючи залежності залежностей та залежності залежностей залежностей, сам скачає їх на пристрій, на якому відбувається складання і сам додасть їх до classpath. І це все дуже сильно полегшує роботу в подальшому.[11]

РОЗДІЛ IV. РЕЗУЛЬТАТИ ДОСЛІДНОГО ТЕСТУВАННЯ ВЕБ РЕСУРСУ ЗА ДОПОМОГОЮ АВТОМАТИЗОВАНОГО ФРЕЙМВОРКУ

4.1. Інструкція для розгортаня та запуску проєкту

Перед початком роботи необхідно встановити необхідні інструменти та програмне забезпечення.

Інструменти для розробки

Java Development Kit (JDK):

Переконайтеся, що у вас встановлена **Java 17** або новіша версія.Щоб перевірити, виконайте:

```
Shell Script
1    java -version
```

Якщо не встановлено, завантажте й встановіть JDK або OpenJDK.

Apache Maven:

Maven використовується для управління залежностями і збірки проєкту.

Щоб перевірити, виконайте:

```
Shell Script
1    mvn -version
```

Якщо не встановлено, завантажте й встановіть Maven.

Selenium WebDriver:

У проєкті використовується Selenium для автоматизації браузера. Перевірте, чи додано залежності Selenium у *pom.xml*.

Інтеграція з браузером:

Завантажте відповідний **WebDriver** для браузера (наприклад, ChromeDriver для Chrome, GeckoDriver для Firefox).

Упевніться, що версія WebDriver відповідає версії браузера.

JUnit:

У проєкті використовується JUnit для створення тестів. Залежність JUnit додається у *pom.xml*.

IDE (IntelliJ IDEA):

Використовується для розробки. Можна завантажити останню версію IntelliJ IDEA.

Розгортання проєкту

Процес розгортання включає завантаження та підготовку проекту до роботи.

Клонуйте репозиторій

Якщо проєкт розміщено у GitHub або іншому контрольному сховищі:

```
Shell Script
1 git clone <URL_репозиторія>
```

Імпорт у IntelliJ IDEA

1. Відкрийте файл *pom.xml* у корені проєкту через IntelliJ IDEA.
2. IntelliJ автоматично запропонує імпортувати залежності Maven. Натисніть **Enable Auto-Import**.

3. Переконайтеся, що всі залежності завантажено та відображаються у вкладці **Maven**.

Перевірка правильності налаштувань

1. Впевніться, що:
    - Драйвер Selenium для вашого браузера вказаний у шляхах системи (або змінній *webdriver.chrome.driver* для Chrome).
    - Сервер/web-адреса тестового середовища зазначений у тестах (за замовчуванням: *http://web.neobank.one/*).
2. Перевірте файл *pom.xml*, щоб переконатися, що всі залежності присутні. Наприклад:

```xml
<dependencies>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>4.6.0</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
    </dependency>
</dependencies>
```

Рисунок 4.1 - Залежності бібліотек

4.2 Опис проекту

Проєкт є набором автоматизованих тестів, написаних на **Java**, використовуючи бібліотеки **JUnit** для тестування та **Selenium WebDriver** для автоматизації веб-додатків. Тести спрямовані на перевірку функціональності, продуктивності та стабільності роботи веб-додатку. Додатково перевіряються API та їхня інтеграція з клієнтським застосунком.

Ось розгорнутий опис головних компонентів та структури проєкту:

Мета проєкту

Забезпечити якість веб-застосунку, автоматизуючи перевірку його основних функцій. Це включає тестування користувацького інтерфейсу, відповідності дизайну, продуктивності завантаження, коректності API, навігації сторінок, форм та інших аспектів веб-застосунку.

Структура проєкту

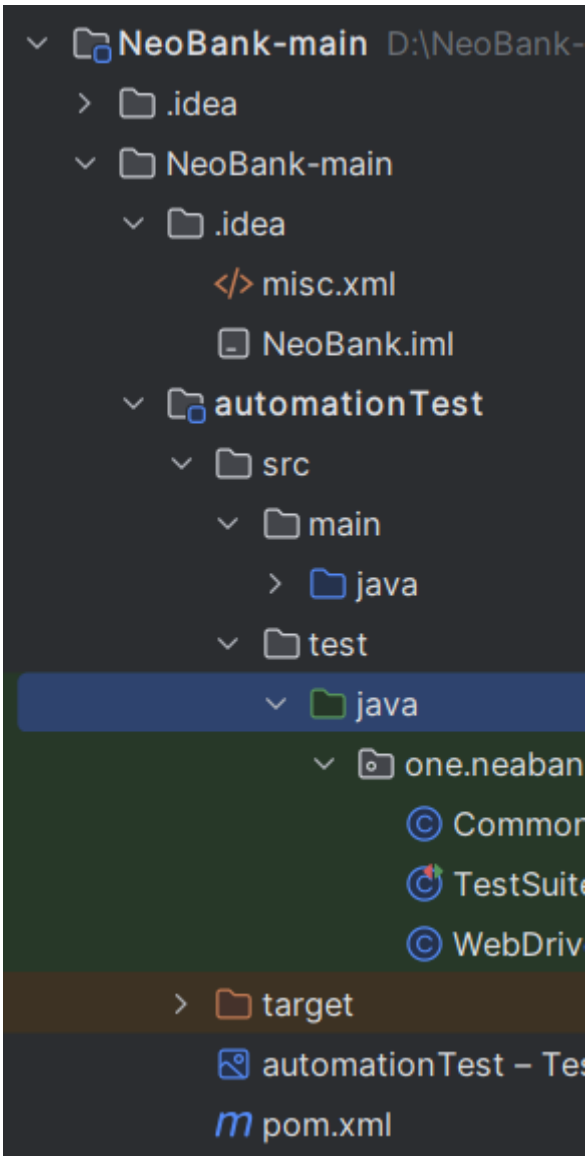
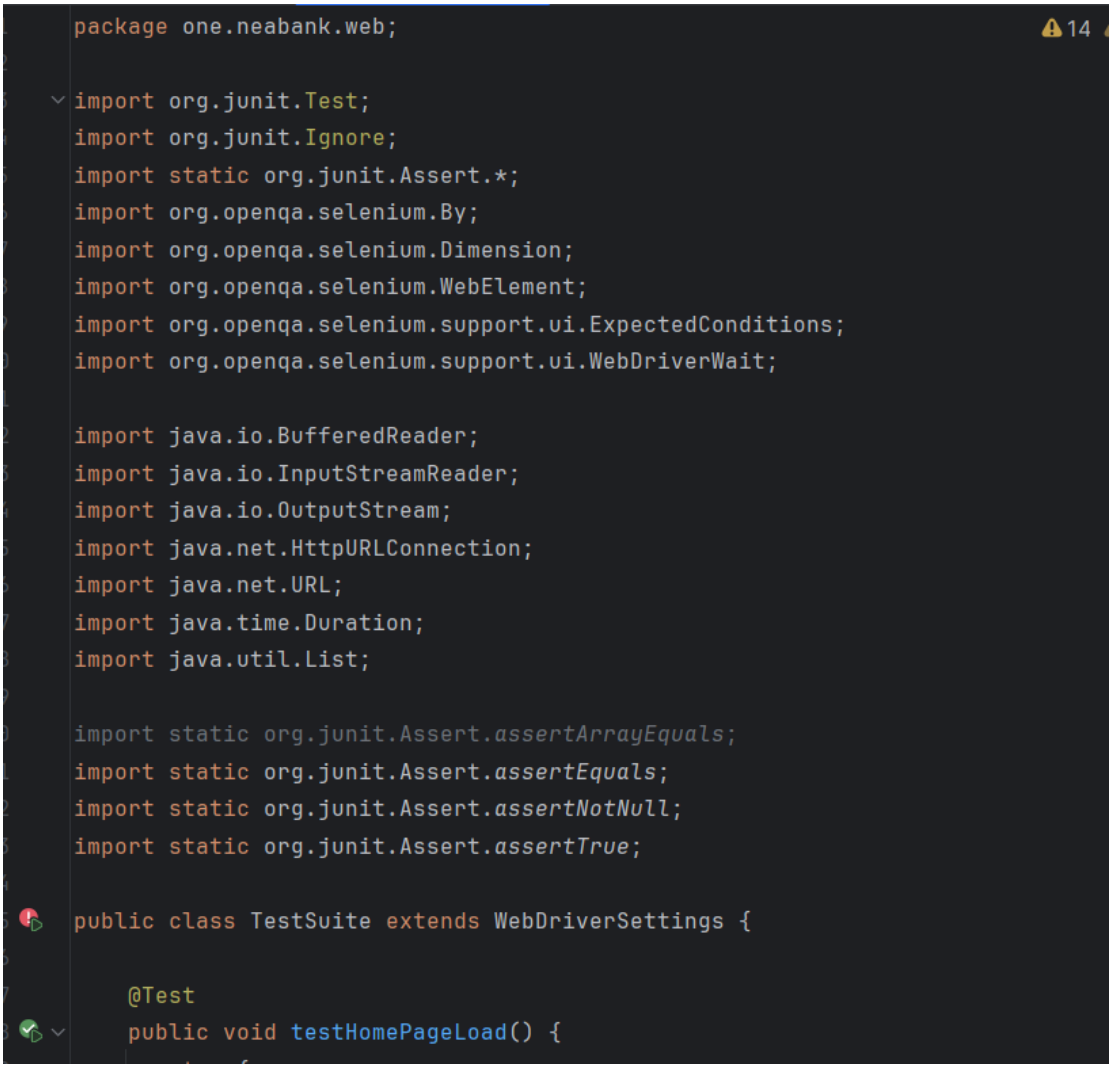


Рисунок 4.2 Загальна структура проєкту

Основний тестовий клас: *TestSuite*

Цей клас містить усі тести, поділені на методи з анотацією ***@Test.*** Тести організовані за функціональними напрямками для забезпечення повного огляду веб-застосунку.



```
package one.neabank.web;                                          ⚠14

import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.*;
import org.openqa.selenium.By;
import org.openqa.selenium.Dimension;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.time.Duration;
import java.util.List;

import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

public class TestSuite extends WebDriverSettings {

    @Test
    public void testHomePageLoad() {
```

Рисунок 4.3 - Структура тестового класу TestSuite

```java
@Test
public void testHomePageLoad() {
    try {
        // Navigate to the website
        driver.get("http://web.neobank.one/");

        // Add implicit wait to ensure elements are loaded
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));

        // Maximize window for better visibility
        driver.manage().window().maximize();

        // Handle potential security warning page
        if (driver.getTitle().contains("Privacy error") || driver.getTitle().con
            // Try to find and click "Advanced" button if present
            try {
                WebElement advanced = driver.findElement(By.id("details-button")
                if (advanced != null) {
                    advanced.click();
                    WebElement proceedLink = driver.findElement(By.id("proceed-l
                    if (proceedLink != null) {
                        proceedLink.click();
                    }
                }
            } catch (Exception e) {
                System.out.println("Security warning page not found or already h
            }
```

Рисунок 4.4. - Тест у структурі класу TestSuite

Вспадкування від базового класу

TestSuite розширює ***WebDriverSettings***.

Це означає, що проєкт має попередньо налаштований базовий клас, який:

- Створює інстанс Selenium WebDriver для обраного браузера.
- Містить конфігурації для виконання тестів.

```java
package one.neabank.web;

import io.github.bonigarcia.wdm.WebDriverManager;  // Add this import
import org.junit.After;
import org.junit.Before;
import org.openqa.selenium.chrome.ChromeDriver;

public class WebDriverSettings {  1 usage  1 inheritor

    public ChromeDriver driver;  29 usages

    @Before
    public void setUp() {
        WebDriverManager.chromedriver().setup();  // Add this line
        driver = new ChromeDriver();
        System.out.println("test start");
    }

    @After
    public void setOut() {
        System.out.println("test close");
        driver.quit();
    }
}
```

Рисунок 4.5 - Базовий клас WebDriverSettings

Тестові групи

Методи використовують бібліотеку JUnit та позначаються за допомогою:

- ***@Test*** — основні автоматизовані тести.
- ***@Ignore*** — тимчасово відключені тести (наприклад, "testNavigationElements", "testLoginForm").

Використані технології

1. **Мови та бібліотеки**:
    - Java
    - JUnit (тестування)
    - Selenium WebDriver (UI-автоматизація)

2. **HTTP-клієнт**:
    - Використовується клас ***HttpURLConnection*** для роботи з API.
3. **Локатори та селектори:**
    - XPath, CSS, ID.

4.2.1 Основні функції та напрямки тестування

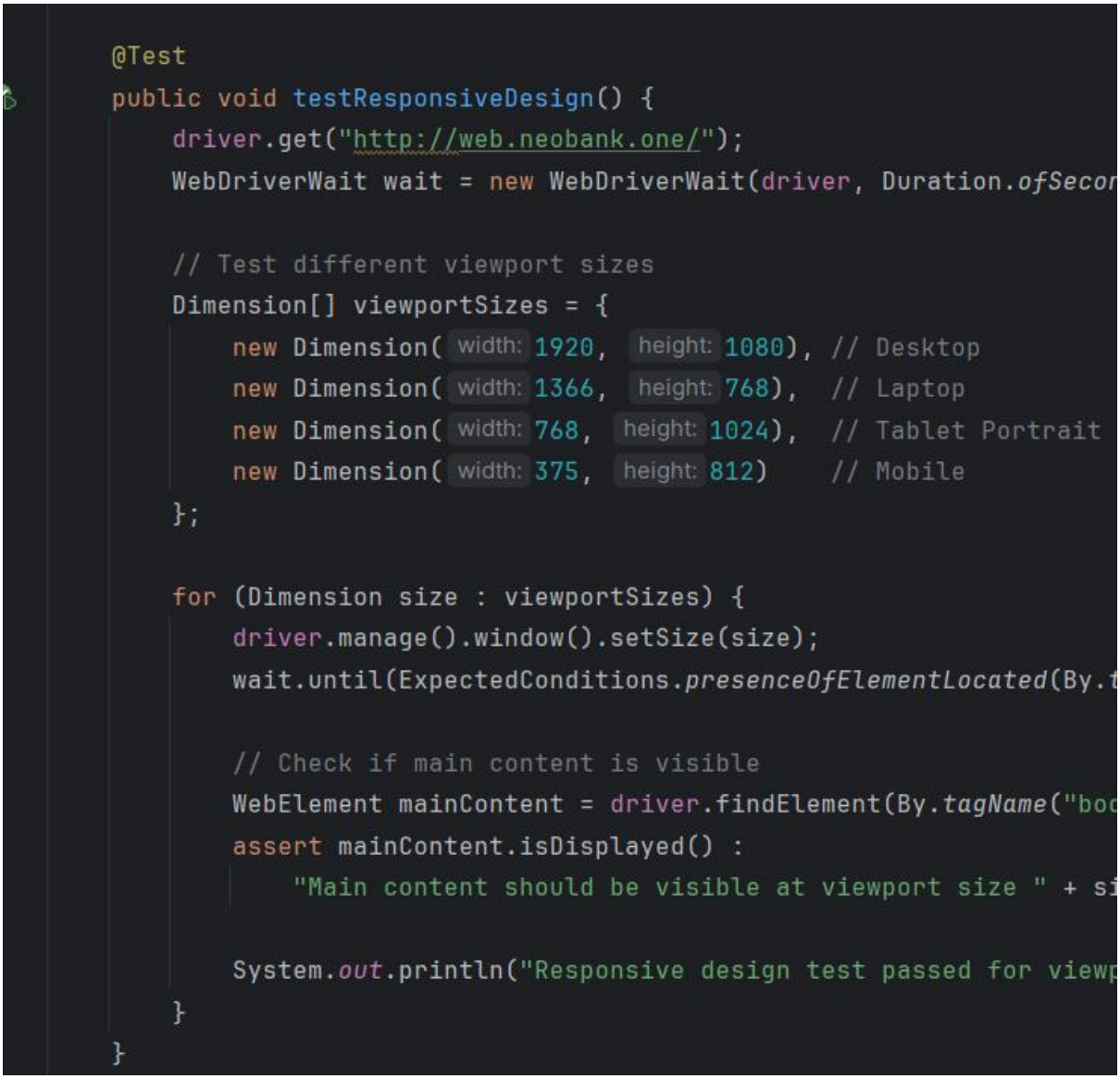
Тестування інтерфейсу користувача (Frontend testing):

- **Перевірка завантаження головної сторінки** Перевіряється коректність завантаження головної сторінки за умов різних сценаріїв, таких як попередження про безпеку чи час завантаження.

```
@Test
public void testHomePageLoad() {
    try {
        // Navigate to the website
        driver.get("http://web.neobank.one/");

        // Add implicit wait to ensure elements are loaded
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10))

        // Maximize window for better visibility
        driver.manage().window().maximize();

        // Handle potential security warning page
        if (driver.getTitle().contains("Privacy error") || driver.getTitl
            // Try to find and click "Advanced" button if present
            try {
                WebElement advanced = driver.findElement(By.id("details-b
                if (advanced != null) {
                    advanced.click();
                    WebElement proceedLink = driver.findElement(By.id("pr
                    if (proceedLink != null) {
                        proceedLink.click();
                    }
                }
            } catch (Exception e) {
                System.out.println("Security warning page not found or al
            }
```

Рисунок 4.6 - Тест кейс перевірки завантаження головної сторінки

- **Респонсивний дизайн** Тест перевіряє, чи коректно веб-додаток адаптується до різних розмірів екрану, зокрема на телефонах, планшетах, ноутбуках та моніторах.



```
@Test
public void testResponsiveDesign() {
    driver.get("http://web.neobank.one/");
    WebDriverWait wait = new WebDriverWait(driver, Duration.ofSecor

    // Test different viewport sizes
    Dimension[] viewportSizes = {
        new Dimension( width: 1920,  height: 1080), // Desktop
        new Dimension( width: 1366,  height: 768),  // Laptop
        new Dimension( width: 768,  height: 1024),  // Tablet Portrait
        new Dimension( width: 375,  height: 812)    // Mobile
    };

    for (Dimension size : viewportSizes) {
        driver.manage().window().setSize(size);
        wait.until(ExpectedConditions.presenceOfElementLocated(By.t

        // Check if main content is visible
        WebElement mainContent = driver.findElement(By.tagName("boc
        assert mainContent.isDisplayed() :
            "Main content should be visible at viewport size " + si

        System.out.println("Responsive design test passed for viewp
    }
}
```

Рисунок 4.7 - Тест перевірки дизайну

- **Теги *meta*** Тести наявності мета-тегів, як-от ***viewport*** чи ***charset***. Вони гарантують, що сторінка відповідає основним стандартам веб-дизайну.
- **Тестування атрибуту lang у HTML-тегу** Перевіряється, чи зазначено мовний атрибут для підтримки міжнародної локалізації.

Тестування функціональності (Functional testing):

- **Навігація між сторінками** Включає перевірку роботи кнопок "назад"/"вперед" у браузері, коректності переходів між сторінками.
- **Форми (наприклад, авторизація)** Тестуються елементи форм, такі як кнопки, поля введення, обробка помилок і повідомлення валідації.
- **Завантаження елементів** Модуль перевіряє наявність ключових елементів, наприклад, шапки або футера, та їхню функціональність.
- **Куки** Повна або часткова перевірка роботи з куками: створення, видалення й очищення.
- **Кнопки оновлення сторінки** Перевіряється, чи дані на сторінці залишаються доступними після оновлення.

Тестування API (Backend testing):

- **Стан здоров'я системи** Тести API ***/api/health*** і ***/api/status*** перевіряють, чи сервер відповідає успішним статус-кодом **200 OK**.

```
@Test
public void testAPIEndpoints() {
    try {
        // Test endpoints
        String[] endpoints = {
            "/api/health",
            "/api/status"
        };

        for (String endpoint : endpoints) {
            URL url = new URL( spec: "http://web.neobank.one" + endpoi
            HttpURLConnection conn = (HttpURLConnection) url.openConn
            conn.setRequestMethod("GET");

            int statusCode = conn.getResponseCode();
            assertTrue( message: "API endpoint " + endpoint + " should
                 condition: statusCode == 200);

            // Read response
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(conn.getInputStream()));
            StringBuilder response = new StringBuilder();
            String line;

            while ((line = reader.readLine()) != null) {
                response.append(line);
```

Рисунок 4.8. - Тест кейс перевірки API

- **Аутентифікація** Перевіряється захист API за допомогою авторизації. Використовуються POST-запити з фіктивними даними авторизації, щоб перевірити відхилення неавторизованих запитів (401 Unauthorized).

```java
@Test
public void testAPIAuthentication() {
    try {
        URL url = new URL( spec: "http://web.neobank.one/api/auth");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-Type", "application/json");
        conn.setDoOutput(true);

        // Add test credentials
        String jsonInputString = "{\"username\":\"test@example.com\",\"password\":\"
        try (OutputStream os = conn.getOutputStream()) {
            byte[] input = jsonInputString.getBytes( charsetName: "utf-8");
            os.write(input,  off: 0, input.length);
        }

        int statusCode = conn.getResponseCode();
        assertEquals( message: "Authentication with invalid credentials should return
             expected: 401, statusCode);

        System.out.println("API authentication test passed");
    } catch (Exception e) {
        System.err.println("Error testing API authentication: " + e.getMessage());
        throw new RuntimeException(e);
    }
```

Рисунок 4.9. - Тест кейс перевірки Аутентифікації

- **Перевірка даних (валідність і формат)** Дані у відповідях API перевіряються на відповідність (наприклад, щоб відповіді не були порожніми).

4. Продуктивність (Performance testing):

- **Час завантаження сторінки** Вимірюється час, необхідний для повного завантаження сторінки. Обмеження: не повинно перевищувати 5 секунд.

```java
@Test
public void testLoadTime() {
    try {
        long startTime = System.currentTimeMillis();
        driver.get("http://web.neobank.one/");

        // Wait for page load
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(30));
        wait.until(ExpectedConditions.presenceOfElementLocated(By.tagName("body")

        // Calculate load time
        long loadTime = System.currentTimeMillis() - startTime;

        // Assert load time is under 5 seconds
        assert loadTime < 5000 :
            "Page load time should be under 5 seconds, actual: " + loadTime + "ms

        System.out.println("Load time test passed. Page loaded in " + loadTime +
    } catch (Exception e) {
        System.err.println("Error testing load time: " + e.getMessage());
        throw e;
    }
```

Рисунок 4.10 - Тест кейс перевірик навантаження

- **Тестування документа *readyState*** Тест перевіряє стан документа після завантаження, щоб переконатися в доступності елементів.

Налаштування браузера та вікна:

- Розмір і масштаб вікна браузера перевіряються для різних сценаріїв використання.
- Встановлюються часи очікування (implicit, explicit), що дозволяє адаптувати поведінку тестів.

4.2.2 Проектні патерни

1. AAA (Arrange-Act-Assert)

Цей патерн є базовим і застосовується майже в кожному тестовому сценарії:

- **Arrange (Підготовка):** Налаштування початкового середовища, запуск драйвера, відкриття сторінки або налаштування часу очікування.
- **Act (Дія):** Виконується конкретна дія, наприклад, відправка запиту, взаємодія з елементом інтерфейсу, переключення вікна тощо.
- **Assert (Перевірка):** Виконується перевірка результатів дії – наприклад, наявність елемента, відповідь API чи зміна стану сторінки.

Приклад з проєкту:

```
@Test
public void testLoadTime() {
   long startTime = System.currentTimeMillis(); // Arrange
   driver.get("http://web.neobank.one/"); // Act
   long loadTime = System.currentTimeMillis() - startTime;
   assert loadTime < 5000 : "Page load time too long"; // Assert
}
```

Data-Driven Testing (Тестування на основі даних)

Цей патерн частково реалізований у таких тестах, як ***testResponsiveDesign:***

- Для перевірки респонсивного дизайну застосовуються різні розміри екрану, записані у вигляді масиву (*viewportSizes*).
- Можливе розширення цього підходу шляхом застосування зовнішніх джерел даних (файли .csv, бази даних, JSON тощо).

Приклад з проєкту:

```
@Test
```

```
public void testResponsiveDesign() {
    Dimension[] viewportSizes = {
        new Dimension(1920, 1080),
        new Dimension(1366, 768),
        new Dimension(768, 1024),
        new Dimension(375, 812)
    };

    for (Dimension size : viewportSizes) {
        driver.manage().window().setSize(size);
        WebElement mainContent = driver.findElement(By.tagName("body"));
        assert mainContent.isDisplayed();
    }
}
```

Smoke Testing (Димове тестування)

Певні тести, як-от ***testHomePageLoad()*** або ***testBasicPageNavigation()***, працюють за принципом **Smoke Tests**. Вони перевіряють основну працездатність компонентів, наприклад:

- Завантаження сторінки.
- Відсутність критичних помилок.
- Результат основної функції (отримання **HTTP 200 OK** у тестах API).

Test Isolation (Ізоляція тестів)

Тести в цьому проєкті ізолюються завдяки:

- Використанню окремих інстансів драйвера та запуску кожного тесту в своїй сесії.

- Видаленню куків перед виконанням тесту testCookieManagement() для уникнення стороннього впливу.
- Необхідність дотримання принципу ізоляції дозволяє запобігти "зависанню" тестів або станів.

5. Behavioral Testing (Тестування поведінки)

Ряд тестів спрямований на перевірку, як застосунок поводить себе в різних сценаріях:

- Робота кнопки "Назад" в історії браузера (*testBrowserHistoryNavigation()*).
- Перевірка поведінки сторінки при оновленні (*testPageReloadFunctionality()*).
- Поведінка форми при введенні некоректних даних у тесті *testLoginForm()*.

6. Negative Testing (Негативне тестування)

У кількох сценаріях застосовується підхід **негативного тестування** – перевірка здатності застосунку коректно обробляти помилкові ситуації:

- В API тестах аутентифікації (*testAPIAuthentication*) відправляються некоректні дані, і перевіряється, чи сервер повертає статус 401 *Unauthorized*.
- Аналогічно, під час тестування форми логіну можна перевіряти обробку як порожніх форм, так і неправильних даних.

4.3. Аналіз результатів тестового рану

Запуск проєкту

1. Відкрийте клас TestSuite.
2. Клацніть правою кнопкою миші у будь-якому місці класу та виберіть **Run 'TestSuite'**.

   Це запустить усі тести в цьому класі.

3. Щоб запустити один конкретний тест:

   Клацніть правою кнопкою по назві методу з анотацією ***@Test*** і виберіть **Run '<Назва_теста>'**.

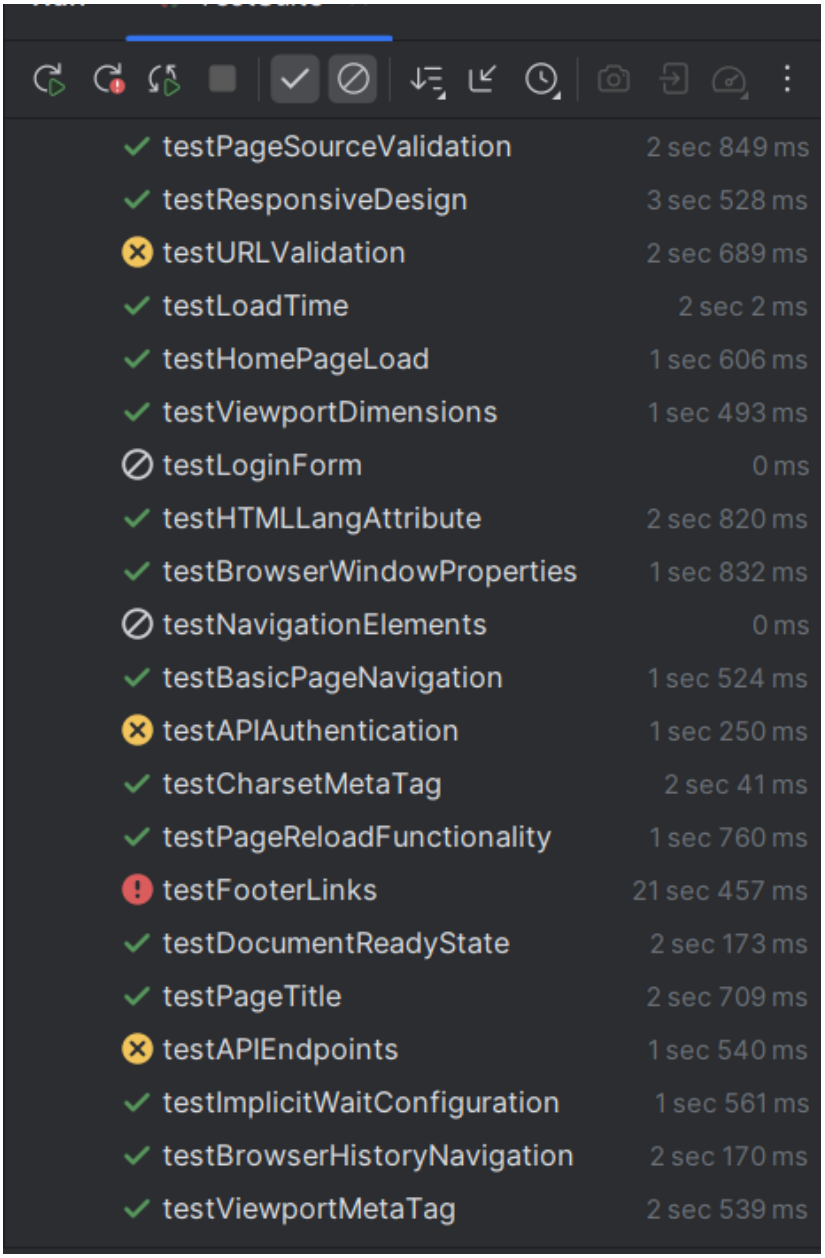


Рисунок 4.11 - Результати запуску автотестів

Аналіз результатів виконання тестів

Усього було написано 21 тест кейс, які покривали різноманітні частини функціоналу веб застосунку. Від UI, Frontend до API та Performance тестів.

Усі **зелені тести (успішно виконані)** свідчать, що ці функції працюють правильно. Загалом з 21 одного тесту 15 є успішними що становить 77% від загальної кількості тестів. Такий відсоток успішних

тестів може свідчити про незначні проблеми з функціоналом проекту, або з якимось конкретним модулем.

Один тест кейс, що становить 4%, не пройшов перевірку. У цьому тесті очікувані результати не збігається з фактичним.
{test start

Error testing footer links: Expected condition failed: waiting for presence of element located by: By.cssSelector: footer, .footer (tried for 20 second(s) with 500 milliseconds interval)

test close

}

Причинами можуть бути:

1. Зміни в оточенні (серверна адреса, версія ПЗ, або тестова сторінка).
2. Некоректний тест (застарілий асерт або хибне очікування).
3. Збої в DOM або мережі (елемент не відобразився через повільне завантаження).

В таких випадках необхідно виконати одну з наступних дій:

- Перевірити очікувані значення в тесті.
- Відкрити веб-ресурс вручну і перевірити, яка фактична поведінка.
- Перевірити налаштування оточення (мережу, дані).

Три тест кейси 14% завершилися через “помилка під час виконання тесту”

- Причина: Тест зламався через виняток або некоректні вхідні дані.

1. Наприклад, NullPointerException, TimeoutException, помилка підключення до API.

- **Можливі причини:**

1. Елемент не знайдений на сторінці (застарілий селектор/зміни на сторінці).
2. Веб-драйвер не зміг відкрити сторінку (помилка мережі, неправильно вказаний URL).
3. Типові мережева помилка в API тестах, наприклад, HTTP 500 (внутрішня помилка сервера).

Лог-повідомлення:

```
org.openqa.selenium.NoSuchElementException: No element found using locator By.id("submit-btn")
```

Це означає, що локатор більше не працює або елемент взагалі відсутній.

Дії:

- Перевірити локацію та доступність елементів (через Selenium DevTools або перезапуск у режимі DEBUG).
- Якщо проблеми виникають у тестах API:
  - Перевірте статус відповідей API через Postman чи curl.
  - Оновіть URL або дані автентифікації для сервісів у тест-кейсі.

Два тест кейси 8% не виконалися та були пропущені (Skipped). Тести пропускаються через анотацію @Ignore.

- **Причини:**
  - Тест очікує розробки функціональності.

- Функція тимчасово відключена.

- **Дії:**
  - Зняти анотацію @Ignore після виправлення функціональності.
  - В ході регулярного тестування перевірити, чи не стали пропущені тести критично важливими.

Оцінка загальної картини та ключові висновки

Загалом тести покривають усі ключові аспекти веб застосунку. У даному тест сюті представлені як UI, FrontEnd так і API, Performance тести. В подальшуму тести можуть доповнюватися щоб покривати більшу кількість юзкейсів.

В той же час, кількість успішних тестів не є оптимальною. За загальноприйнятими стандартами частка невдалих тестів повинна складати не більше 5%. У нашому випадку це 14%. Що говорить про те що частину тестів потрідно проаналізувати та доопрацювати.

Щодо тривалості виконання, то всі тести виконуються в межах норми. На даний момент тести оптимізації швидкості виконання не потребують.

В наступних тестових ранах потрібно звіряти результати із вже отриманими, та виявляти тенденції зміни. Якщо кількість невдач збільшується з часом, можливо, є регресія в проєкті (зміни коду, які ламають раніше працюючий функціонал). Якщо помилки незмінні між кількома тестовими ранами, це може свідчити про проблеми в тестах, а не в ПЗ.

ВИСНОВКИ

Розробка фреймворку для автоматизації тестування дозволяє значно знизити витрати часу та ресурсів на перевірку якості програмного продукту. Використання сучасних інструментів, таких як JUnit для тестування та Selenium для автоматизації веб-інтерфейсів, забезпечує високу ефективність і надійність тестових процесів. Автоматизація тестування дозволяє не тільки скоротити час на перевірку програмного забезпечення, але й підвищити точність результатів тестів, зменшивши ймовірність помилок, які можуть бути не виявлені при ручному тестуванні.

Однак для успішної реалізації автоматизованих тестів необхідно враховувати специфіку проекту, вибір відповідних фреймворків та інструментів, а також постійну підтримку тестових середовищ і сценаріїв. Оцінка результатів тестування на кожному етапі проекту є важливою для запобігання регресій та забезпечення стабільної роботи продукту в умовах змін.

Під час аналізу були розглянуті найпопулярніші на сьогоднішній день інструменти для автоматизації тестування вебдодатків. Досліджена їх продуктивність, заміряна швидкодія та характеристики, та побудовані графіки залежностей характеристик. Були виділені конкретні інструменти і бібліотеки, які необхідні для реалізації програмної системи автоматизації тестування вебдодатків.

В результаті дослідження було запропоновано ефективну стратегію автоматизації тестування для веб-додатків, що включає використання сучасних інструментів та методів, що забезпечують високу якість програмного продукту на всіх етапах його розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Clean Code: A Handbook of Agile Software Craftsmanship/Robert C. Martin. Pearson 2008. 464p.
2. Effective Java, 3rd Edition/ Joshua Bloch. Upper Saddle River, NJ : AddisonWesley, 2017. 416p.
3. Fowler M. Monolith First / Martin Fowler [Електронний ресурс] – режим доступу: https://martinfowler.com/bliki/MonolithFirst.html#footnotetypical-monolith.
4. Hands-On Software Architecture with Java: Learn key architectural techniques and strategies to design efficient and elegant Java applications/ Giuseppe Bonocore. Packt Publishing 2021. 510p.
5. Introduction to Algorithms, fourth edition 3th Edition/ Thomas H. Cormen. MIT Press 2009. 1292p.
6. Java Design Patterns: A Hands-On Experience with Real-World Examples 2nd ed. Edition/ Vaskaran Sarcar. Apress 2018. 533p.
7. Java: The Complete Reference, Eleventh Edition/Herbert Schildt. McGraw Hill Education 2018. 1248p.
8. Modern Java in Action: Lambdas, streams, functional and reactive programming 2nd Edition/ Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft. Manning – 2018. 592p.
9. RESTful Java Web Services: A pragmatic guide to designing and building RESTful APIs using Java, 3rd Edition 3rd Revised edition/ Bogunuva Mohanram Balachandar. Packt Publishing 2017. 420p.
10. Spring in Action, 5th edition/Craig Walls. Manning Publications 2018. 520p.
11. Test Driven Development: By Example 1st Edition/ Kent Beck. Addison-Wesley Professional 2002. 240p.

12. Web Engineering: Modelling and Implementing Web Applications / G. Rossi, P. Oscar, S. Daniel. – New York: Springer-Verlag, 2008. 476p.

Вебдодаток і його характеристики [Електронний ресурс] https://www.centum-d.com/uk/veb-dodatok-yogo-harakteristiki

13. Introduction to DOM elements [Електронний ресурс] https://frontender.info/an-introduction-to-dom-events/

14. Сучасний веброзробник [Електронний ресурс] https://medium.com/nuances-of-programming/%D1%81%D0%BE%D0%B2%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9backend%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%87%D0%B8%D0%BA-2018-a43d51a7bcd1

15. Фронтенд і Бекенд розробка [Електронний ресурс] https://skillbox.ru/media/code/frontend_i_backend_razrabotka/

16. 6 Different Types of Web Application Development [Електронний ресурс] https://www.clustox.com/6-different-types-of-web-application-development

17. Automation Testing Life Cycle [Електронний ресурс] https://www.lambdatest.com/blog/all-you-have-to-know-about-automation-testing-life-cycle

18. Компонентне тестування [Електронний ресурс] http://www.protesting.ru/testing/levels/component.html

19. Інтеграціонне тестування [Електронний ресурс] http://www.protesting.ru/testing/levels/integration.html

20. End-to-End тестування [Електронний ресурс] https://habr.com/ru/post/417395

21. Ranking Programming Languages by GitHub Users [Електронний ресурс] https://www.benfrederickson.com/ranking-programming-languages-by-github-users/

22. Дослідження переваг і недоліків методології розробки програмного забезпечення через тестування / Павловський В.І., Разсказов М.П.; Прикладна математика та комп'ютинг 2023, 2023. – С. 285-289

23. Ушакова І. О. Підходи до забезпечення якості програмного забезпечення / І. О. Ушакова // Сучасні інформаційні технології і системи : монографія / за заг. ред . В. С. Пономаренка. - Харків : «Стильіздат», 2021. – С. 125-140.

24. Difference Between Quality Assurance And Quality Control (QA Vs QC) [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://www.softwaretestinghelp.com/quality-assurance-vs-quality-control/.

25. What Is the International Organization for Standardization (ISO)? [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://www.investopedia.com/terms/i/international-organization-for-standardizationiso.asp.

26. 5 reasons why Java is still the best programming language [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://www.theserverside.com/feature/5-reasons-why-Java-is-still-the-bestprogramming-language.

27. Why Java is the best Programming language to Learn Coding? [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://medium.com/javarevisited/why-java-is-the-best-programming-language-tolearn-coding-for-beginners-cba79aed1271