МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВОЛИНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЛЕСІ УКРАЇНКИ
Кафедра комп’ютерних наук та кібербезпеки

На правах рукопису

СКОРУК ДЕНИС АНДРІЙОВИЧ

УДОСКОНАЛЕННЯ МЕТОДІВ РОЗРОБКИ ТА ОРГАНІЗАЦІЇ ФРЕЙМВОРКУ ДЛЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБОРІЄНТОВАНИХ КОРИСТУВАЦЬКИХ ІНТЕРФЕЙСІВ

Спеціальність: 122 Комп’ютерні науки
Освітньо-професійна програма: Комп’ютерні науки та інформаційні технології
Робота на здобуття освітнього ступеня “магістр”

Науковий керівник:
ГЛИНЧУК ЛЮДМИЛА ЯРОСЛАВІВНА
кандидат фізико-математичних наук, доцент
кафедри комп’ютерних наук та кібербезпеки

РЕКОМЕНДОВАНО ДО ЗАХИСТУ
Протокол №_____
засідання кафедри комп’ютерних наук
та кібербезпеки
від _____ 2024 р.
Завідувач кафедри
(_____) Гришанович Т. О.

ЛУЦЬК – 2024

ЗМІСТ

ВСТУП

Актуальність дослідження. Сучасні веборієнтовані користувацькі інтерфейси (UI) є важливою складовою цифрових продуктів, оскільки від їхньої зручності та функціональності залежить якість взаємодії користувача із системою. У зв’язку зі зростанням кількості вебдодатків та їхньою складністю виникає потреба в ефективних методах тестування для забезпечення якості програмного забезпечення. Автоматизоване тестування стає ключовим інструментом для скорочення часу тестування, мінімізації людських помилок та підвищення надійності продукту. Однак, розробка та організація фреймворків для автоматизації тестування потребує подальшого вдосконалення, щоб відповідати вимогам швидкої розробки та інтеграції.

Мета дослідження – удосконалення методів розробки та організації фреймворку для автоматизованого тестування веборієнтованих інтерфейсів з метою підвищення ефективності та якості тестування.

Завдання дослідження:

- розглянути і проаналізувати існуючі методи та технології для автоматизованого тестування вебінтерфейсів;
- визначити ключові проблеми при розробці фреймворків для автоматизованого тестування;
- розробити фреймворк для автоматизованого тестування користувацьких вебінтерфейсів;
- оцінити ефективність запропонованого підходу на практичних прикладах.

Об'єкт дослідження – процеси та підходи до тестування веборієнтованих користувацьких інтерфейсів.

Предмет дослідження – методи та принципи розробки фреймворків для автоматизованого тестування UI-інтерфейсів.

Практичне застосування одержаних результатів. Одержані в межах дослідження результати можуть бути впроваджені в різних сферах розробки вебдодатків. Розроблений фреймворк може бути інтегрований у конвеєри

безперервної інтеграції та доставки (CI/CD), що дозволить автоматично перевіряти працездатність користувацьких інтерфейсів на кожному етапі розробки. Це сприяє зменшенню ризику появи помилок у фінальній версії продукту та прискорює процес релізів.

Використання вдосконаленого фреймворку дозволить зменшити обсяг рутинних тестів, які виконуються вручну, і зосередити зусилля інженерів з якості на більш складних тестових сценаріях. Це підвищить ефективність роботи команди та зменшить ймовірність людських помилок. Запропонований підхід дозволить створювати автоматизовані тести для різних браузерів та пристроїв, що забезпечить коректне функціонування інтерфейсу на всіх платформах. Це особливо важливо для вебдодатків, які повинні працювати однаково добре на різних типах пристроїв.

Автоматизація ключових тестових сценаріїв зменшить час на перевірку функціоналу під час кожної ітерації розробки. Це дасть можливість швидше випускати оновлення та оперативно реагувати на потреби ринку або зміну вимог замовників. Розроблений фреймворк може бути адаптований до специфіки різних галузей, зокрема e-commerce, фінансових технологій, медіа та інших, де веборієнтовані інтерфейси відіграють ключову роль у взаємодії з користувачами.

Апробація.

Скорук Д.А., Глинчук Л.Я., Особливості розробки фреймворку для тестування UI частини вебдодатків. *Інформаційні технології і автоматизація – 2024*: матеріали XVII міжн. наук.-практ. конференції. (Одеса, 31 жовтня - 1 листопада 2024 р.). Одеса, Видавництво ОНТУ, 2024 р. С. 433-435

Скорук Д. А., Аналіз інструменту Selenium та його роль у автоматизації тестування. *Проблеми комп'ютерних наук, програмного моделювання та безпеки цифрових систем*: матеріали І міжн. наук-практ. конференції. (Табір "Гарт" Луцьк-Світязь, 13 червня - 16 червня 2024 р.). С. 67.

РОЗДІЛ 1.
ТЕОРЕТИЧНІ ОСНОВИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

1.1 Роль тестування в процесах забезпечення якості продукту

У сфері написання програмного забезпечення важливе значення мають стандарти якості продукту. Робота у сфері QA (Quality Assurance) спрямована на забезпечення цього стандарту шляхом планування, контролю та тестування на всіх етапах розробки програми. Метою QA є не лише виявлення помилок, а й їх попередження для створення надійного та функціонального програмного забезпечення відповідно до потреб клієнтів.

QA відрізняються від тестування тим, що це не просто перевірка готового продукту. Це охоплювання стандартизацій процесів, визначення критерій якості та методологій і взаємодії з розробниками і аналітиками для введення системного підходу до розвитку проекту.

Основний принцип QA полягає в ітеративному та безперервному поліпшенні. Це передбачає, що кожен етап розробки – від збору вимог до релізу – супроводжується ретельним аналізом та тестуванням. Тестування може бути ручним або автоматизованим, залежно від завдань та специфіки продукту. Ручне тестування актуальне там, де потрібно оцінити інтерфейс з точки зору користувача або відчути особливості взаємодії з продуктом. Але у великих проєктах, де швидкість і повторюваність тестів критично важливі, ручний підхід часто не може забезпечити необхідну ефективність. Тут на перший план виходить автоматизація тестування.

Основна ідея QA полягає в постійному та ітеративному покращенні процесу розробки продукту. Це визначається тим, що кожний етап від збору вимог до випуску супроводжується детальним аналізом і тестуванням якості продукту. Тестування може бути проведено вручну або автоматизовано залежно від характеристик завдань та самого продукту. Коли потрібно оцінити користувацький інтерфейс або особливості сприйняття продукту користувачем – доцільне ручне тестування. Але у

великому проєкті де швидкість та повторюваність тестів критично необхідна – ручне тестування часто не забезпечує досить ефективними результатами. При цих обставинах автоматизація тестів виходить на передній план. [8]

З розвитком Agile-методологій та DevOps-підходів, які передбачають швидкі релізи й безперервну інтеграцію нових компонентів, потреба в автоматизованому тестуванні стає очевидною. Коли код змінюється щодня або навіть щогодини, перевіряти стабільність усієї системи вручну стає занадто повільним і ресурсомістким процесом. Автоматизація дозволяє швидко запускати сотні тестів, перевіряючи не лише нові функції, але й їхню сумісність із наявними модулями. Це знижує ризик регресії – ситуації, коли новий код порушує вже працюючі функції.

Автоматизоване тестування інтегрується в процеси CI/CD (безперервної інтеграції та доставки), дозволяючи проводити тестування щоразу, коли розробники вносять зміни в код. Такий підхід сприяє виявленню помилок на ранніх стадіях і забезпечує стабільність продукту. Проте автоматизація – не є панацеєю. Вона вимагає інвестицій у розробку скриптів, налаштування інфраструктури та регулярного обслуговування тестів. Важливо також розуміти, що не всі аспекти тестування доцільно автоматизувати – складні або креативні сценарії, де важлива оцінка людського сприйняття, краще довірити ручному тестуванню.

Таким чином, автоматизація тестування є логічним етапом еволюції процесів QA. Вона дозволяє забезпечити високу швидкість розробки без втрати якості, що особливо важливо в умовах сучасного ринку програмного забезпечення. Правильне поєднання ручного та автоматизованого тестування дозволяє максимально ефективно досягати мети – створення продуктів, які не лише працюють, але й приносять користувачам задоволення та цінність. [9]

1.2 Поняття, види та особливості автоматизованого тестування

Автоматизоване тестування – це процес перевірки програмного забезпечення (ПЗ) за допомогою спеціальних інструментів і скриптів, які автоматично виконують

тестові сценарії. Мета такого підходу – підвищення ефективності тестування, зменшення часу на виявлення помилок та покращення якості продукту. На відміну від ручного тестування, де тестування здійснюється людиною, автоматизація дозволяє виконувати тести швидко та повторювано. [1]

Автоматизація тестування поступово стала ключовим інструментом у забезпеченні якості програмного забезпечення, особливо в умовах швидких релізів і складних проєктів. Її суть полягає у створенні тестових сценаріїв і використанні спеціальних програмних інструментів для їх виконання без участі людини. Це дозволяє перевіряти продукт ефективно й швидко, скорочуючи час тестування та підвищуючи його точність.

Автоматизація стає особливо цінною у повторюваних завданнях, таких як регресійні тести. Кожна нова версія продукту потребує перевірки не лише змін, але й коректної роботи старих функцій. Там, де для цього знадобилися б години або дні ручної роботи, автоматизовані скрипти справляються за хвилини. Це підвищує продуктивність і зменшує ризик пропустити критичну помилку.

Однак автоматизація – це не просто запис і виконання тестів. Для її ефективної реалізації потрібно чітко розуміти, які саме тести мають бути автоматизовані. Найкраще підходять ті завдання, які повторюються багато разів, мають чіткі очікувані результати й не залежать від суб’єктивної оцінки. Це можуть бути функціональні тести, що перевіряють правильність роботи бізнес-логіки, або навантажувальні тести, які визначають, чи витримає система великі обсяги одночасних запитів.

Автоматизація вимагає інвестицій не лише у програмне забезпечення, але й у підготовку фахівців. Тестувальники повинні вміти програмувати, налаштовувати середовище для запуску тестів та інтегрувати їх у CI/CD конвеєри. Інструменти, такі як Selenium, Cypress або Appium, дозволяють створювати складні тестові сценарії для веб та мобільних додатків, тоді як Jenkins або GitLab CI допомагають автоматично запускати ці тести при кожній зміні в коді. [2]

Ще однією особливістю автоматизації є те, що тестові скрипти потрібно підтримувати. Програма розвивається, змінюються інтерфейси та логіка, через що

старі тести можуть перестати бути актуальними. Це потребує регулярного оновлення тестів і створення гнучкої архітектури для їх підтримки.

Автоматизоване тестування охоплює різні аспекти функціонування програмного забезпечення. Кожен вид автотестів має свою специфіку, завдання та підходи до реалізації. Далі розглянемо ключові види автоматизованого тестування та особливості їх використання. [3]

1. Модульне тестування (Unit Testing)

Цей рівень тестування фокусується на найменших частинах програми – модулях або функціях, перевіряючи їхню роботу ізольовано від решти системи. Модульні тести часто пишуть самі розробники, щоб переконатися, що кожна функція або клас працює саме так, як очікується.

Модульні тести виконуються швидко й служать основою для побудови інших рівнів автоматизації. Вони допомагають виявляти помилки в бізнес-логіці на ранніх етапах, що зменшує витрати на їх виправлення в майбутньому.

2. Інтеграційне тестування (Integration Testing)

На цьому рівні перевіряється взаємодія між різними модулями або компонентами. Модулі, які працюють коректно окремо, можуть зіткнутися з проблемами на етапі інтеграції, наприклад, через несумісність інтерфейсів або неочікувані залежності. Інтеграційне тестування гарантує, що всі частини системи працюють разом узгоджено.

Цей рівень часто використовується для перевірки взаємодії між різними сервісами або підсистемами, як-от між бекендом і базою даних, або між API і фронтендом.

3. Системне тестування (System Testing)

На цьому рівні перевіряється повний продукт як цілісна система. Метою системного тестування є перевірка того, що всі компоненти працюють разом і відповідають початковим вимогам. Це включає тестування функціональних, нефункціональних та технічних аспектів продукту.

Тут можуть використовуватися як ручні, так і автоматизовані тести для перевірки основних бізнес-сценаріїв у реальних умовах. [6]

1.3 Особливості розробки фреймворків для автоматизованого тестування

Фреймворк для автоматизованого тестування – це набір інструментів, бібліотек та методологій, які дозволяють розробникам і тестувальникам створювати, виконувати та підтримувати автоматизовані тести.

Перш ніж почати розробку фреймворку, важливо чітко визначити його цілі та завдання. Фреймворк може бути орієнтований на різні типи тестування: функціональне, регресійне, навантажувальне, безпеки тощо. Це визначає вибір інструментів і технологій, які будуть використані у фреймворку. Наприклад, якщо фреймворк призначений для функціонального тестування вебдодатків, необхідно включити інструменти для роботи з браузерами, такі як Selenium або Playwright.

Основним аспектом під час написання фреймворку є планування його архітектури. Розробники повинні створити ієрархію папок і файлів, яка відображає різні аспекти тестування, такі як:

- основні тестові сценарії, які описують поведінку системи в різних умовах, при цьому кожен тест має бути зрозумілим і лаконічним, щоб легко відстежувати, які функціональні можливості вони перевіряють;
- класи або модулі, що описують елементи інтерфейсу користувача;
- допоміжні функції та бібліотеки, які полегшують роботу з тестами (це можуть бути функції для роботи з файлами, обробки даних, логування, генерування звітів тощо (утиліти також можуть включати в себе інструменти для роботи з арі, якщо ваше тестування передбачає взаємодію з бекендом));
- файли конфігурації для налаштувань середовища тестування (це можуть бути файли з параметрами для тестування (наприклад, url-адреси, дані для входу), а також налаштування для інтеграції з різними сервісами (такими як бази даних,

системи безперервної інтеграції тощо), використання конфігураційних файлів дозволяє легко змінювати параметри тестування без потреби в змінах в коді).

Наступним кроком потрібно визначити набір технологій які будуть використовуватися для організації та проходження тестів.

Основою для написання тестових сценаріїв є мова програмування. Вибір мови впливає на доступність бібліотек та інструментів для автоматизації. Найпопулярнішими мовами для автоматизованого тестування є Java, Python, JavaScript, C#. Важливо вибрати мову, яка буде зрозумілою для команди і підтримуватиме обраний фреймворк.

Інструменти для побудови проєкту, такі як Maven, Gradle (для Java) або npm (для JavaScript), дозволяють автоматизувати управління залежностями, компіляцію коду та запуск тестів. Вони спрощують процес налаштування проєкту, дозволяючи легко додавати нові бібліотеки та інструменти, а також автоматизувати виконання завдань, таких як тестування і збірка артефактів. Ці інструменти також забезпечують зручність у налаштуванні середовищ, з якими працюють тести.

Інструменти такі як TestNG (для Java) або pytest (для Python), дозволяють структурувати тести, управляти їх виконанням, налаштовувати залежності та проводити паралельне тестування. Вони забезпечують можливість групувати тести в тести, параметризувати їх, а також організовувати звітування про результати. Це дозволяє командам легко управляти великою кількістю тестових сценаріїв і підвищує продуктивність процесу тестування.

Для автоматизації тестування вебдодатків використовуються інструменти для взаємодії з браузером, такі як Selenium, Cypress або Playwright. Ці інструменти дозволяють автоматично виконувати дії в браузері, такі як заповнення форм, натискання кнопок і перевірка вмісту сторінок. Вони забезпечують можливість тестувати вебдодатки на різних браузерах і платформах, що є важливим для забезпечення крос-браузерної сумісності.

Інструменти для створення звітності, такі як Allure, ExtentReports або ReportNG, допомагають генерувати зрозумілі та візуально привабливі звіти про виконання

тестів. Вони збирають дані про результати тестування, час виконання, логування та інші метрики, що дозволяє командам легко аналізувати результати і швидко реагувати на проблеми. Звіти допомагають зацікавленим сторонам, таким як керівники проектів і замовники, отримувати зрозумілу інформацію про якість продукту.

Варто зазначити, що фреймворк повинен бути спроектований з урахуванням розширюваності та гнучкості. Це означає, що нові модулі або функції можуть бути легко додані без потреби в значних змінах в основному коді. Використання патернів проектування, таких як “Page Object Model”, може допомогти досягти цього.

Фреймворк для автоматизованого тестування потребує постійного обслуговування та оновлення. Це включає в себе виправлення помилок, оновлення бібліотек, адаптацію до нових версій продукту, що тестується, і врахування зворотного зв'язку від команди тестувальників. [5]

1.4 Практики організації коду автотестів

Для ефективного тестування ПЗ не достатньо просто написати код для автоматичного проходження тестових сценаріїв. Щоб фреймворк для тестування працював ефективно, були сформовані певні принципи та патерни, які дозволяють покращити організацію коду. Вони допомагають уникати дублювання, зменшувати складність та уникати перевантаження непотрібним функціоналом. Серед найбільш важливих із них виділяють DRY (Don’t Repeat Yourself), KISS (Keep It Simple, Stupid) та YAGNI (You Aren’t Gonna Need It). Ці підходи не лише спрощують процес написання й підтримки тестів, але й роблять код більш надійним та зрозумілим для команди [18].

Принцип DRY застерігає від дублювання коду й логіки в межах фреймворку для тестування. Якщо одна й та сама логіка використовується в кількох тестах або модулях, це призводить до збільшення обсягу коду, ускладнює підтримку та підвищує

ймовірність помилок. Будь-які зміни в логіці потребують внесення правок у кількох місцях, що підвищує ризик несинхронізованих змін.

Реалізація DRY передбачає створення спільних методів чи функцій для багаторазового використання. Наприклад, якщо багато тестів потребують авторизації користувача, краще винести цю логіку у функцію

KISS наголошує на важливості простоти. Тести мають бути легкими для читання та розуміння як новими, так і досвідченими членами команди. Складність призводить до заплутаності коду, підвищення кількості помилок і ускладнює підтримку та модифікацію тестів.

Застосування KISS означає, що тести повинні виконувати лише необхідні перевірки без зайвого ускладнення логіки. Наприклад, якщо потрібно лише підтвердити, що користувач успішно авторизувався, достатньо перевірити наявність кнопки “Вихід”, не перевантажуючи тест іншими перевірками.

YAGNI підкреслює важливість зосередження на поточних потребах проекту. Він попереджає про небезпеку написання коду чи функцій, які, можливо, знадобляться в майбутньому, але не є необхідними зараз. Надмірне передбачення майбутніх вимог призводить до ускладнення проєкту, збільшення обсягу коду й підвищення витрат на його підтримку.

Прикладом YAGNI може бути рішення не додавати логіку для багатофакторної аутентифікації, якщо на поточному етапі потрібна лише авторизація з паролем.

Перевагою дотримання цих принципів є можливість їх взаємодії між собою, тобто вони доповнюють один одного та сприяють створенню збалансованого коду.

DRY і KISS працюють разом для досягнення чистоти й простоти коду. Хоча DRY заохочує уникнення дублювань, надмірне прагнення до цього може зробити код складним для розуміння. Звідси важливість принципу KISS, який вказує на необхідність зберігати рішення якомога простішими. Наприклад, не варто об'єднувати все в одну функцію, якщо це ускладнює читання й підтримку коду.

KISS і YAGNI допомагають уникати зайвої складності та передчасної оптимізації. KISS підкреслює, що рішення мають бути максимально простими, а

YAGNI вчить не додавати функціонал, який може ніколи не знадобитися. Разом ці принципи запобігають ситуаціям, коли код стає перевантаженим зайвими можливостями чи складною логікою.

DRY і YAGNI взаємодіють так, що дозволяють уникнути як дублювання коду, так і передчасного створення зайвих компонентів. Наприклад, прагнення уникнути дублювань не повинно спонукати до створення універсальних рішень для випадків, які ще не виникли. Код має залишатися адаптивним і легко модифікованим тоді, коли це дійсно потрібно. [19]

Спільне дотримання цих принципів допомагає команді створювати підтримуваний та гнучкий код. Тести залишаються зрозумілими, а їхня логіка – чіткою й адаптивною до змін. Завдяки цьому розробники можуть швидше впроваджувати нові можливості та впевненіше працювати зі змінами в системі, що позитивно позначається на загальній якості продукту. Проте, у процесі автоматизації тестування важливо не лише дотримуватися загальних принципів, але й застосовувати перевірені архітектурні патерни, які допомагають структурувати код ефективніше. Одним із таких патернів, який часто використовується в автоматизованому тестуванні інтерфейсів, є Page Object Model (POM). Він безпосередньо реалізує ідеї, закладені в принципах DRY, KISS та YAGNI, і дозволяє розробникам будувати тестовий код так, щоб його було легко масштабувати та підтримувати [10].

Page Object Model (POM) – це архітектурний патерн, який пропонує структурувати код тестів так, щоб кожна вебсторінка або компонент інтерфейсу представлялися у вигляді окремого об’єкта (класу). Всі елементи та методи для взаємодії з цією сторінкою зосереджені всередині відповідного об’єкта. Таким чином, логіка тестів відділяється від логіки роботи з елементами UI, що допомагає досягти чистоти коду та полегшує його підтримку.

У моделі Page Object кожна сторінка чи компонент інтерфейсу представляються окремим класом. У цьому класі зберігаються:

- елементи сторінки (поля вводу, кнопки, посилання тощо);

- методи для взаємодії з елементами (наприклад, логін або заповнення форми).

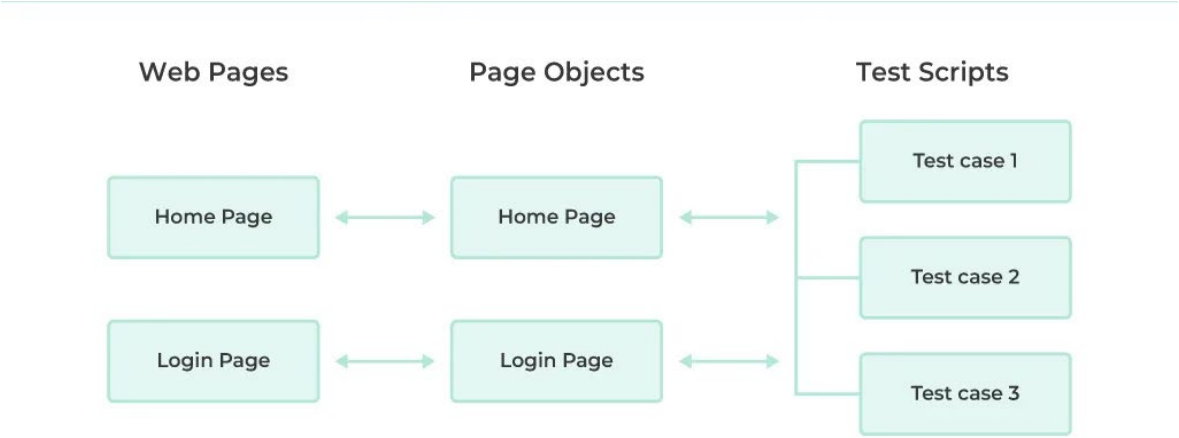

Рисунок 1.1 – Схема побудови фреймворку за паттерном Page Object Model

Тестові сценарії використовують ці класи як інтерфейси для роботи зі сторінками. Завдяки цьому логіка тестів зосереджується лише на перевірках, тоді як технічна взаємодія з елементами сторінки прихована всередині відповідних класів.

Цей підхід робить тести більш стійкими до змін в інтерфейсі: якщо змінюється структура сторінки (наприклад, змінюється ID кнопки), достатньо внести правки лише в одному місці – у класі, що відповідає цій сторінці.

Наведемо основні переваги POM.

Покращена підтримуваність коду. Якщо змінюється логіка сторінки чи локатори, всі зміни вносяться лише в одному місці – у відповідному класі сторінки. Це значно зменшує обсяг роботи та ймовірність помилок.

Зрозумілі тести. Тестові сценарії стають легкими для читання, адже вони містять лише послідовність дій, а не складні технічні деталі. Це допомагає новим членам команди швидко зануритися в роботу.

Стійкість до змін. Якщо в інтерфейсі змінюються назви чи структури елементів, правки потрібно внести лише у відповідному класі сторінки, а не в усіх тестах. Це зменшує ризик ламання тестів через невеликі зміни в UI.

Перевикористання коду. Один і той самий клас сторінки може використовуватися в багатьох тестах, що забезпечує економію часу та уникнення дублювання коду (принцип DRY).

Наведемо можливі недоліки POM і шляхи їх усунення.

Первинні витрати на реалізацію. На початковому етапі побудова структури POM може здатися трудомісткою. Однак ці витрати швидко окупаються на етапі підтримки, коли зміни вносяться лише в одному місці.

Ризик надмірної складності. Якщо кожен дрібний компонент перетворюється на окремий клас, це може ускладнити структуру тестів. Рекомендується дотримуватися принципу KISS та групувати компоненти розумно, створюючи окремі класи лише для значущих частин інтерфейсу.

Залежність від драйвера. У POM класи сторінок зазвичай прив’язані до конкретного інструменту, такого як Selenium WebDriver або Playwright. Щоб мінімізувати залежність, можна використовувати додаткові рівні абстракції, наприклад, створювати базові класи або інтерфейси для сторінок. [12]

1.5 Огляд та аналіз існуючих фреймворків

Selenium. Selenium є одним із найпоширеніших інструментів для автоматизованого тестування веб-додатків, що має модульну архітектуру. Основними компонентами є Selenium WebDriver, який забезпечує взаємодію з браузерами через спеціалізовані драйвери; Selenium IDE, що слугує інтегрованим середовищем для запису та виконання тестів без програмування; та Selenium Grid, який дозволяє виконувати паралельне тестування на кількох машинах і в різних середовищах. Така архітектура забезпечує високу гнучкість і масштабованість у використанні.

Інструмент розробляється спільнотою з відкритим кодом під егідою організації SeleniumHQ. Вперше проєкт був представлений інженерами ThoughtWorks у 2004 році як засіб для автоматизації рутинних задач у браузері. Завдяки активній підтримці

спільноти та внескам розробників із різних компаній Selenium залишається одним із найбільш розвинених інструментів у своїй галузі.

Основними функціями Selenium є підтримка автоматизації веб-додатків у різних браузерах, таких як Chrome, Firefox, Edge, Safari та Opera, а також сумісність із кількома мовами програмування, включаючи Java, Python, C#, JavaScript, Ruby та PHP. Інструмент дозволяє виконувати як функціональні, так і регресійні тести, забезпечуючи паралельне тестування за допомогою Selenium Grid. Selenium інтегрується з іншими інструментами, такими як TestNG, JUnit, Maven, Allure, та CI/CD-платформами, такими як Jenkins і GitLab CI, що робить його універсальним рішенням для автоматизованого тестування.

Серед переваг Selenium варто виділити його безкоштовність і відкритий код, що сприяє доступності для широкого кола користувачів і швидкому впровадженню нових функцій. Інструмент є мультиплатформним і працює на операційних системах Windows, MacOS і Linux. Крім того, його розширюваність дозволяє адаптувати Selenium до потреб конкретного проєкту, а підтримка паралельного тестування значно скорочує час виконання тестів.

Однак Selenium має й недоліки. По-перше, його функціонал обмежується тестуванням лише веб-додатків, що унеможливлює використання для десктопних застосунків. По-друге, початківцям може бути складно налаштувати середовище для роботи через необхідність взаємодії з драйверами та залежностями. Selenium також демонструє нестабільність тестів через чутливість до змін у DOM або затримок у мережі, що може спричиняти флакінність. Окрім цього, інструмент має відносно низьку продуктивність у порівнянні з сучасними альтернативами, такими як Playwright чи Cypress, які пропонують швидше виконання тестів і кращу підтримку сучасних функцій браузера.

Таким чином, Selenium є потужним інструментом для автоматизованого тестування веб-додатків, особливо у великих проєктах, що потребують широкої підтримки браузерів і мов програмування. Однак його недоліки спонукають до

розгляду комбінованого використання з іншими сучасними інструментами для досягнення максимальної ефективності тестування.

Рисунок 1.2 – Логотип Selenium

Puppeteer. Puppeteer є інструментом для автоматизації роботи з веб-браузерами, який побудований на сучасній архітектурі з акцентом на швидкість і простоту використання. Основою його роботи є взаємодія з браузерами через протокол DevTools Protocol, що забезпечує низькорівневий доступ до функцій браузера. Puppeteer розроблений для роботи з Chromium і Google Chrome, надаючи API для програмного управління браузерами, включаючи підтримку таких операцій, як рендеринг сторінок, скриншоти, генерація PDF і автоматизація форм.

Розробником Puppeteer є компанія Google, яка вперше представила цей інструмент у 2017 році. Основною метою розробки було створення зручного інструмента для тестування веб-додатків та інтеграції з екосистемою браузерів Chromium. Завдяки постійній підтримці Google Puppeteer отримує регулярні оновлення, що відповідають сучасним потребам веб-розробників і тестувальників.

Серед основних функцій Puppeteer варто виділити автоматизацію дій у браузері, таких як навігація, кліки, введення тексту та перевірка елементів DOM. Інструмент підтримує генерацію скриншотів і PDF-файлів, емуляцію мобільних пристроїв, моніторинг мережевих запитів і навіть створення тестів для сучасних SPA-додатків. Завдяки прямій інтеграції з DevTools Protocol Puppeteer дозволяє отримувати низькорівневу інформацію про стан браузера та веб-сторінки, що робить його корисним для профілювання та аналізу продуктивності.

Серед переваг Puppeteer слід відзначити високу швидкість виконання завдань завдяки прямій взаємодії з браузером без необхідності використання проміжних драйверів. Інструмент забезпечує стабільність і передбачуваність виконання тестів, що зменшує ризик виникнення флакіністі. Puppeteer має інтуїтивний API, який легко освоїти навіть новачкам, а також можливість роботи з сучасними веб-технологіями, такими як Shadow DOM і веб-компоненти. Крім того, Puppeteer підтримує емуляцію пристроїв і середовищ, що дозволяє проводити тестування адаптивних інтерфейсів.

Проте Puppeteer має і певні обмеження. Головним недоліком є його орієнтація виключно на браузери Chromium, що обмежує його використання для тестування у Firefox, Safari чи інших браузерах. Крім того, Puppeteer підтримує лише JavaScript і TypeScript, що може бути незручним для проєктів, де використовуються інші мови програмування. У порівнянні з Selenium Puppeteer має меншу гнучкість у масштабних проєктах, де потрібна підтримка різних мов програмування та браузерів.

Таким чином, Puppeteer є ефективним інструментом для автоматизації тестування веб-додатків, особливо в контексті сучасних SPA-додатків і застосунків, що використовують новітні веб-технології. Його швидкість, стабільність і зручний API роблять його ідеальним вибором для вузькоспеціалізованих завдань, однак його обмежена сумісність із браузерами поза екосистемою Chromium є значним недоліком у проєктах із високими вимогами до кросбраузерності.

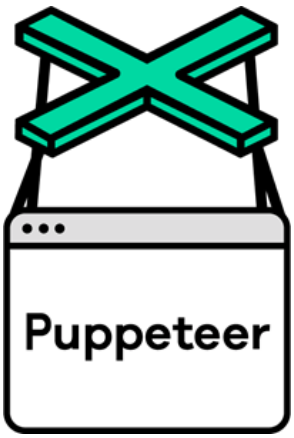

Рисунок 1.3 – Логотип Puppeteer

1.6 Визначення вимог до фреймворків для автоматизованого тестування

На основі проведеного аналізу основних інструментів для створення фреймворку автоматизованого тестування вебінтерфейсів сформульовано низку вимог до програмної реалізації та методів, які забезпечать досягнення мети дипломної роботи.

Для забезпечення продуктивності та можливості розширення фреймворку мова програмування має бути багатопоточною, підтримувати інтеграцію з основними інструментами для автоматизації тестування та з бібліотеками для роботи з динамічними вебелементами. [11] Згідно з цих потреб було вирішено використати наступні технології:

- використання Java як основної мови програмування для забезпечення стабільності, продуктивності та можливості багатопоточної обробки тестів;
- вкористання Maven як системи управління збірками для легкого управління залежностями та налаштування структури проєкту.

Структура фреймворку повинна бути чіткою, підтримувати можливість розширення, забезпечувати легкість управління та підтримки, а також мати модульну архітектуру для можливості використання в різних проєктах.

- впровадження чіткої архітектури з модулями, що відповідають за різні компоненти;
- використання патерну Page Object для зберігання локаторів і методів роботи з елементами сторінок, що сприяє підвищенню читабельності та полегшенню підтримки коду;
- інкапсуляція елементів управління тестами та звітністю у окремі модулі для забезпечення можливості заміни компонентів у майбутньому.

Для забезпечення ефективного тестування вебінтерфейсів фреймворк повинен підтримувати паралельне виконання тестів, можливість групування тестів, інтеграцію з інструментами для UI-автоматизації та звітності.

Використання TestNG для підтримки паралельного виконання тестів та групування сценаріїв, що дозволяє масштабувати тести для великих проєктів.

Інтеграція Playwright для автоматизації браузерних тестів завдяки підтримці роботи з динамічними елементами та багатобраузерній сумісності.

Розробка налаштовуваних фільтрів для вибору тестів, що підвищує ефективність та гнучкість запуску тестів.

Інтеграція з Allure для автоматичної генерації звітів, що включають детальну інформацію про виконання тестів, прикріплені скріншоти та індикатори успішності. [20]

РОЗДІЛ 2.

РОЗРОБКА ФРЕЙМВОРКУ ДЛЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

2.1 Постановка задачі, призначення та вимоги до розробки

Щоб підвищити ефективність та якість тестування веборієнтованих інтерфейсів потрібно провести дослідження задля удосконалення методів та організації фреймворків.

В процесі постановки задачі до кваліфікаційної роботи було сформовано план прикладного дослідження.

1. Аналіз існуючих рішень та популярних проблем. Перегляд сучасних методик та підходів для написання автоматизованих тестів. Визначити основні недоліки, що зустрічаються у сучасних рішеннях.
2. Формування вимог до фреймворку. Виходячи з проблемних зон, сформувати вимоги для фреймворку.
3. Розробка архітектури фреймворку. Провести чітке планування архітектури проєкту з урахуванням вибраного стеку технологій.
4. Реалізація прототипу та тестування. Розробити прототип фреймворку для тестування веборієнтованих користувацьких інтерфейсів. Провести перевірку прототипу на відповідність до вимог з застосуванням оптимізаційних заходів.
5. Документування та оцінка результатів. Провести детальний аналіз отриманих результатів, на основі яких сформувати об’єктивну оцінку щодо проведеної роботи, опираючись на сформовані раніше вимоги.

Задача кваліфікаційної роботи полягає у створенні та вдосконаленні методів розробки тестового фроймворку, який зможе забезпечити ефективний процес автоматизованого тестування веборієнтованих користувацьких інтерфейсів. Серед основних вимог до фреймворку відноситься:

- модульність і масштабованість;

- гнучкість конфігурації;
- стандартизація тестування;
- висока швидкість тестування та надійність;
- докладне логування та звітність.

Фреймворк повинен підтримувати ключові принципи автоматизованого тестування. В результаті реалізації задумки, вона значно зменшить час для регресійного тестування, підвищить якість продукту та швидкість випуску оновлень на ринок.

2.2 Методологія дослідження

Дане дослідження використовує емпіричний підхід, який передбачає експериментальну перевірку ефективності фреймворку для автоматизованого тестування веборієнтованих користувацьких інтерфейсів. Проведене дослідження включало аналіз існуючих методів тестування, розробку та тестування нових компонентів, а також удосконалення вже наявних.

Основна гіпотеза дослідження полягає в тому, що використання новітніх технологій і методів удосконалення структури фреймворку дозволить значно підвищити ефективність автоматизованого тестування, скоротити час налаштування тесту та підвищити надійність виконання. тестів.

Для розробки фреймворку була обрана ітераційно-інкрементальна модель, яка відповідає вимогам гнучкості та адаптивності, які необхідні для цього типу проекту. Дана модель передбачає поетапне розширення функціональних можливостей фреймворку разом з постійним тестуванням і налаштуванням фреймворку, що дає можливість своєчасно усувати виявлені недоліки і тим самим підвищує ефективність системи в цілому.

Процес розробки фреймворку охоплював декілька ключових етапів. Спочатку було проведено детальний аналіз вимог, який включав збір та вивчення потреб цільової аудиторії, а також вимог до інтеграції з іншими інструментами. Далі

виконано етап проектування, в межах якого розроблено архітектуру фреймворку з визначенням основних компонентів та взаємодії між ними. На етапі реалізації створено окремі модулі, включаючи ті, що відповідають за налаштування тестових середовищ, запуск тестів, обробку результатів і генерацію звітів. Після цього було проведено всебічне тестування кожного модуля та всієї системи в цілому, що дозволило впевнитися у відповідності фреймворку заявленим вимогам. Завершальним етапом стало впровадження фреймворку та розробка рекомендацій щодо його інтеграції та підтримки в контексті CI/CD.

Таким чином, обрані методологія, підходи та інструменти дозволили забезпечити високу якість, надійність та адаптивність розробленого фреймворку, а також ефективну інтеграцію в сучасні процеси розробки програмного забезпечення.

2.3 Теоретичні аспекти дослідження

Архітектура фреймворку є об’єктно-орієнтованою, що сприяє повторному використанню коду і розширюваності функціоналу. Основу структури становлять класи, які описують різні типи тестів та їхні компоненти.

Класи для обробки інтерфейсів відповідають за взаємодію з вебелементами. За допомогою бібліотеки Playwright у фреймворку реалізовано об’єкти, які керують діями, що виконує тест, такими як клік, заповнення форм або навігація.

Класи для управління тестовими даними забезпечують зручне зберігання та доступ до наборів даних, які використовуються під час тестування. Ці класи відповідають за структурування даних, що дозволяє легко змінювати тестові сценарії.

Класи для налаштування середовищ містять конфігурації для тестування різних браузерів, версій ОС та інших умов. Це дозволяє тестувальникам легко змінювати параметри середовища, зменшуючи час на переналаштування.

Взаємодія між об’єктами відбувається через Page Object класи, де основним принципом є організація коду таким чином, що тести не містять конкретних деталей про елементи інтерфейсу. Це дозволяє тестам залишатися незалежними від змін у

структурі сторінок, що робить фреймворк більш стабільним і довготривалим у використанні. Усе це також значно спрощує підтримку, оскільки кожна сторінка має централізоване місце для модифікації взаємодій із її елементами.

Ефективність використання фреймворку оцінюється за такими критеріями, як надійність тестів, можливість масштабування та зручність оновлення при змінах інтерфейсу. Завдяки патерну Page Object Model фреймворк демонструє високі показники за вказаними критеріями, що забезпечує конкурентоспроможність та легкість інтеграції з іншими інструментами автоматизації.

2.4 Обґрунтування вибору інструментальних засобів

IntelliJ IDEA є інтегрованим середовищем розробки, що використовується для різних мов програмування, таких як Java, Python, Scala, PHP та інших. Цей інструмент, розроблений компанією JetBrains, є потужним засобом для професійних розробників.

Середовище IntelliJ IDEA доступне у двох основних версіях: безкоштовній “Community Edition” та комерційній “Ultimate Edition”. Перша версія надає базові функції, необхідні для розробки, тоді як “Ultimate Edition” пропонує повний набір функціональних можливостей і надає безкоштовні ліцензії для активних розробників відкритих проєктів.

Програмний код версії “Community Edition” розповсюджується під ліцензією Apache 2.0, що забезпечує високий рівень відкритості та доступності. Бінарні збірки даного середовища розроблені для трьох основних операційних систем: Linux, Mac OS X та Windows, що дозволяє користувачам обирати найбільш зручну для них платформу.

IntelliJ IDEA володіє низкою переваг, які роблять його видатним інструментом для розробників. Ця платформа забезпечує розширену інтелектуальну підтримку кодування, що включає автозаповнення, аналіз коду та рефакторинг, що значно покращує продуктивність і знижує ймовірність помилок. IntelliJ IDEA підтримує

велику кількість мов програмування, таких як Java, Kotlin, Python, Scala, PHP та інші, що дозволяє розробникам працювати з різними проектами в одному середовищі. Платформа пропонує широкий спектр плагінів, які дозволяють налаштувати середовище відповідно до потреб користувача, роблячи його гнучким інструментом, який адаптується до різних вимог проектів. IntelliJ IDEA має вбудовану підтримку для систем контролю версій, таких як Git, а також інтеграцію з системами безперервної інтеграції та розгортання, що забезпечує зручну і безперебійну роботу з проєктами.

Оптимізована для швидкої роботи, IntelliJ IDEA зменшує час виконання завдань і підвищує ефективність розробників, завдяки продуманому і зручному інтерфейсу користувача. Крім того, IDE має велику спільноту користувачів і розробників, що забезпечує постійну підтримку та обмін досвідом, а офіційна документація та навчальні матеріали допомагають швидко освоїтися з інструментом. Ці переваги роблять IntelliJ IDEA потужним і надійним вибором для професійних розробників, що прагнуть до високої якості і ефективності в своїй роботі. [17]

Рисунок 2.1 – Логотип IntelliJ IDEA

Java – це мова програмування загального призначення, якою володіє корпорація Oracle. Java побудована на принципах об'єктно-орієнтованого програмування. Мова Java дотримується принципу WORA (“Write Once, Run Anywhere”), який дає багато переваг для крос-платформного тестування.

Java використовується для підтримки внутрішніх корпоративних систем в багатьох великих корпораціях. Програми, написані на мові Java, працюють більш ніж на 3 мільярдах пристроїв. Незважаючи на те, що для юніт тестування

найпопулярнішою платформою є JUnit, багато платформ для автоматизації тестування з відкритим вихідним кодом були розроблені на мові Java.

У цій мові є як плюси, так і мінуси. З позитивного – це PageObject, спрощує код для автоматизації та є можливість писати дуже прості для розуміння тести. У Java існує велике ком'юніті, нею володіють багато розробників і на ній вже написана величезна кількість інструментів. Внаслідок цього тестувальники часто мають можливість запитати поради у розробників і легше знайти готове рішення під певне завдання. З мінусів – код читається складніше, ніж навіть в Python. Ще одним мінусом є повідомлення про помилки, які часто складно зрозуміти [7].

Рисунок 2.2 – Логотип Java

“Apache Maven” – це засіб автоматизації роботи з програмними проєктами, який спочатку використовувався для Java проєктів. Використовується для керування (management) та складання (build) програм. Створений 2002 року Джейсоном ван Зилом. За принципами роботи кардинально відрізняється від Apache Ant, та має простіший вигляд щодо build-налаштувань, яке надається в форматі XML. XML-файл описує проєкт, його зв'язки з зовнішніми модулями й компонентами, порядок будування (build), теки та необхідні плагіни. Сервер із додатковими модулями та додатковими бібліотеками розміщується на серверах.

Для опису програмного проєкту, який потрібно побудувати (build), Maven використовує конструкцію відому як Project Object Model (POM), залежності від зовнішніх модулів, компонентів та порядку побудови. Виконання певних, чітко

визначених завдань – таких, як компіляція коду та пакетування відбувається шляхом досягнення заздалегідь визначених цілей (targets).

Ключовою особливістю Maven є його мережева готовність (network-ready).

Рушій ядра може динамічно завантажувати плагіни з репозиторію, того самого репозиторію, що забезпечує доступ до багатьох версій різних Java-проєктів з відкритим кодом, від Apache та інших організацій та окремих розробників. Цей репозиторій та його реорганізований наступник – Maven 2 намагається бути де-факто механізмом для дистрибуції Java програм, але прийняття його в такій ролі йде повільно.

Maven забезпечує підтримку побудови не просто перебираючи файли з цього репозиторію, але й завантажуючи назад артефакти у кінці побудови. Локальний кеш звантажених артефактів діє як первісний засіб синхронізації виходу проєктів на локальній системі.

Maven базується на плагін-архітектурі, що дозволяє зробити використання будь-якої програми контрольованим через стандартний вхід. Теоретично, це могло б дозволити будь-кому писати плагіни для інтерфейсу з інструментами для побудови (компілятори, тестери тощо) для будь-якої мови. На ділі, підтримка й використання для мов відмінних від Java були мінімальною. Тепер існують плагіни для .NET та C/C++. [14]

Рисунок 2.3 – Логотип Maven

TestNG – це фреймворк для автоматизованого тестування програмного забезпечення, написаний мовою Java. Він призначений для створення та запуску тестів для перевірки функціональності та якості програмних продуктів.

TestNG дозволяє розробникам та тестувальникам створювати тести, які можуть бути запущені автоматично, що заощаджує час та підвищує ефективність тестування.

Фреймворк підтримує різні типи тестів, включаючи функціональні, інтеграційні та тести навантаження.

Однією з ключових переваг TestNG є його гнучкість та настроюваність. Користувачі можуть створювати власні тестові сценарії, використовуючи анотації та конфігураційні файли, що дозволяє їм адаптувати фреймворк до конкретних потреб їх проекту.

TestNG також підтримує паралельне виконання тестів, що дозволяє прискорити процес тестування та підвищити продуктивність. Крім того, фреймворк інтегрується з іншими інструментами та системами, такими як Jenkins, Maven та Eclipse, що спрощує процес тестування та розгортання програмного забезпечення. [13]

Рисунок 2.4 – Логотип TestNG

Playwright – це потужна бібліотека для автоматизації, створена Microsoft, яка дозволяє тестувати вебдодатки на різних браузерах. Основною перевагою Playwright є його здатність підтримувати кілька браузерів, включаючи Chromium (наприклад, Google Chrome і Microsoft Edge), WebKit (який використовується в Safari), та Firefox. Це забезпечує повне охоплення тестування та гарантує, що додатки будуть працювати коректно на будь-якому з цих браузерів.

Також у Playwright є підтримка кількох мов програмування. Розробники можуть писати свої тестові скрипти на JavaScript, TypeScript, Python, C# та Java, що робить цей інструмент дуже гнучким та зручним для інтеграції в існуючі проекти, незалежно від мови, яку використовує команда розробників.

Playwright підтримує різні платформи, включаючи Windows, macOS, Linux, Android та iOS. Це означає, що розробники можуть тестувати свої додатки на будь-

якій з цих платформ, що знижує ймовірність виникнення проблем під час запуску на різних пристроях користувачів.

Ще одним важливим аспектом Playwright є можливість виконання тестів як в режимі без графічного інтерфейсу (Headless), так і з графічним інтерфейсом (Headful). Режим без графічного інтерфейсу корисний для швидкого виконання тестів в CI-середовищах, що дозволяє прискорити процес тестування та інтеграції. Режим з графічним інтерфейсом, зі свого боку, є ідеальним для розробки та відладки, дозволяючи розробникам бачити, що відбувається на екрані під час тестування.

До переваг також можна віднести те, що Playwright має вбудовані автоматичні чеки, які допомагають зменшити ризик провалу тестів через непередбачувані обставини. Це значно підвищує надійність та стабільність тестів, полегшуючи процес автоматизації [15].

Рисунок 2.5 – Логотип Playwright

Allure Report – це популярний інструмент з відкритим вихідним кодом для візуалізації результатів тестування. Його можна додати до вашого тестового процесу без потреби в значній конфігурації. Він створює звіти, які можна відкрити будь-де і прочитати будь-кому, без потреби в глибоких технічних знаннях.

На відміну від спеціалізованих інструментів звітування для певних фреймворків, Allure Report підтримує кілька мов і фреймворків, дозволяючи використовувати будь-яку їх комбінацію. Наприклад, якщо вам потрібно окремо запускати тести бекенду та фронтенду, ви все одно можете перетворити всі результати в один тестовий звіт – і отримати чіткіше уявлення про те, що відбувається у вашому проєкті.

Водночас, Allure Report має цілу екосистему інтеграцій з різними тестовими фреймворками та бібліотеками. Вони додають певні дані автоматично і надають API для додавання ще більшої кількості даних.

Загалом Allure має масу корисних властивостей та гнучко налаштовується, тому це чудовий вибір для створення звітності власних автотестів. [16]

Рисунок 2.6 – Логотип Allure Reporter

2.5 Етапи програмної реалізації

Для того щоб безпосередньо перейти до написання фреймворку для автоматизованого тестування потрібно спочатку налаштувати основні модулі середовища розробки.

Для початку на ПК потрібно завантажити сам IDE, це можна зробити з офіційного сайту продукту: https://www.jetbrains.com/idea/. JetBrains пропонує різні варіанти програми проте для особистих цілей достатньо скористатися версією Community Edition. Вона безкоштовна тому чудово підходить для власних розробок чи навчання.

На етапі налаштування можна вибрати місце для встановлення програми, а також додаткові опції, як-от створення ярликів на робочому столі чи асоціацію типів файлів з IntelliJ IDEA. Після завершення інсталяції можна відкрити програму і побачити вітальний екран, де вам запропонують імпортувати налаштування з іншої інсталяції, якщо це необхідно. Далі потрібно пройти початкове налаштування, яке включає вибір теми інтерфейсу та необхідних плагінів, які можуть бути корисними для роботи з конкретними мовами програмування чи фреймворками.

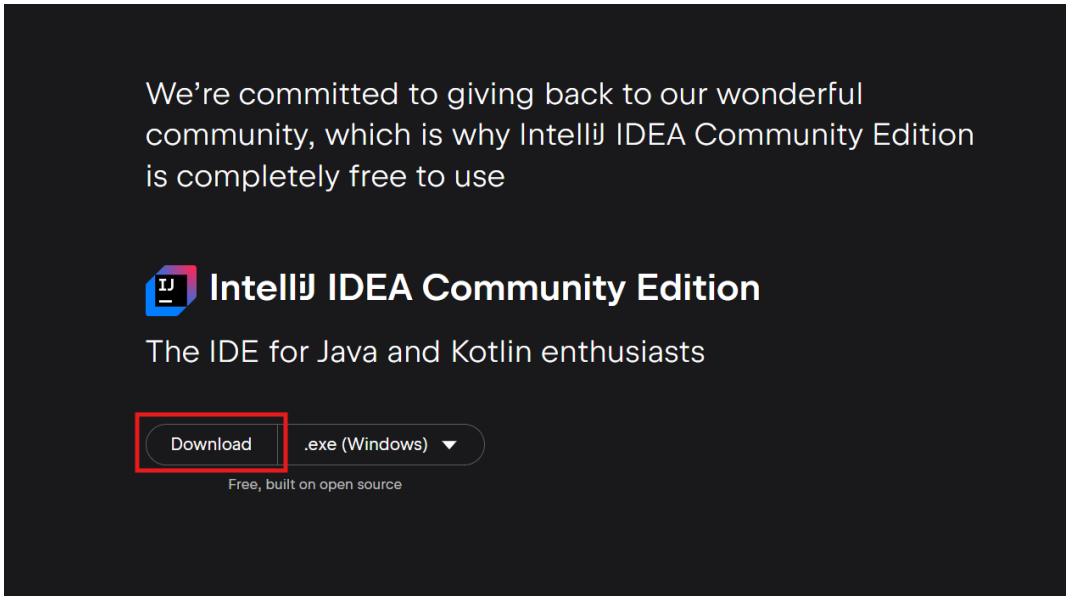

Рисунок 2.7 – Кнопка завантаження IntelliJ IDEA Community Edition

Середовище також потребуватиме інструментів для розробки на Java, або JDK (Java Development Kit), який можна скачати на однойменному офіційному сайті. Сам JDK складається з двох компонентів: JVM (Java Virtual Machine) і JRE (Java Runtime Environment), перший відіграє роль платформи на якій запускаються всі програми написані на Java, а другий відповідає за запуск таких програм.

Аби JDK запрацював у IDE достатньо вказати шлях до папки у налаштуваннях середовища: File - Project Structure

Після цього потрібно скачати та встановити інструмент який буде відповідати за налагодження проєкту, в даному випадку це Maven. Для того щоб завантажити його достатньо перейти на офіційний сайт продукту та обрати зручний спосіб для завантаження.

Після того як архів було завантажено, його потрібно розархівувати у довільну папку та підключити до середовища IntelliJ IDEA. Це можна зробити наступним кроком:

File - Settings… - Build, Execution, Deployment - Build Tools - Maven

У цьому меню потрібно вказати шлях до папки Maven, яку було розпаковано раніше. Після налаштування натисніть кнопку “Apply” та “Ok”.

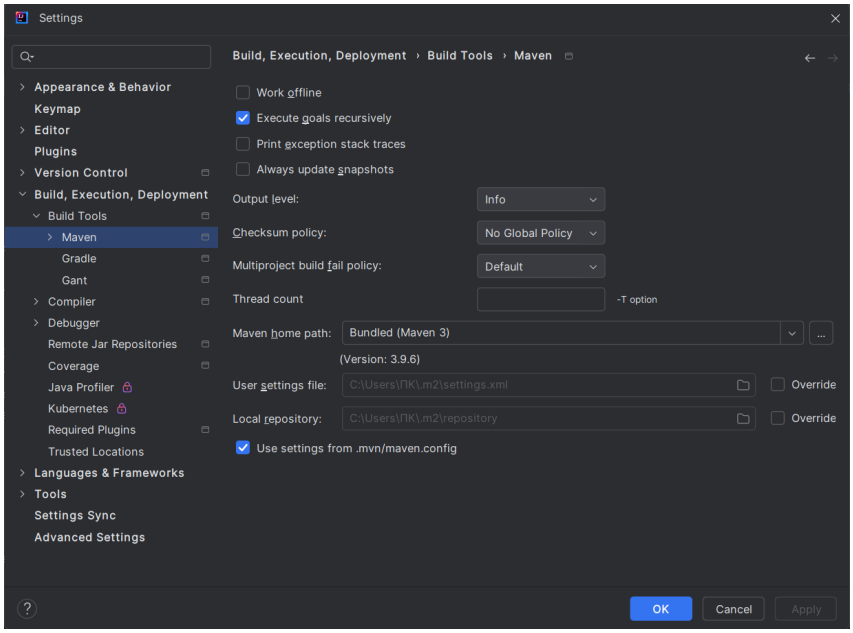

Рисунок 2.8 – Меню налаштування Maven

Щоб перевірити, що Maven встановлено правильно, достатньо відкрити термінал в IntelliJ IDEA та ввести команду *mvn -version*. Якщо все встановлено правильно, повинна відобразитись версія Maven, а також шлях до JDK.

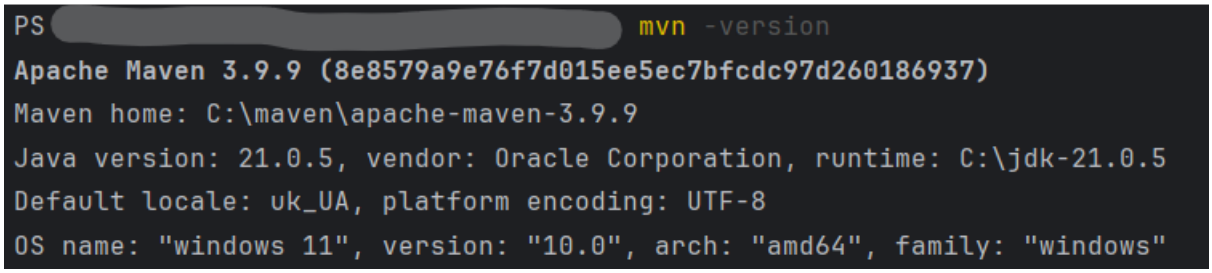

Рисунок 2.9 – Відображення версії Maven та шлях до JDK

Коли Maven успішно підключено, створений проєкт буде містити спеціальний конфігураційний файл під назвою “pom” у розширенні XML. Він відповідає за

організацію та підключення інструментів у проєкті. Саме він буде використовуватися для підключення таких інструментів як TestNG, Playwright та Allure Reports.

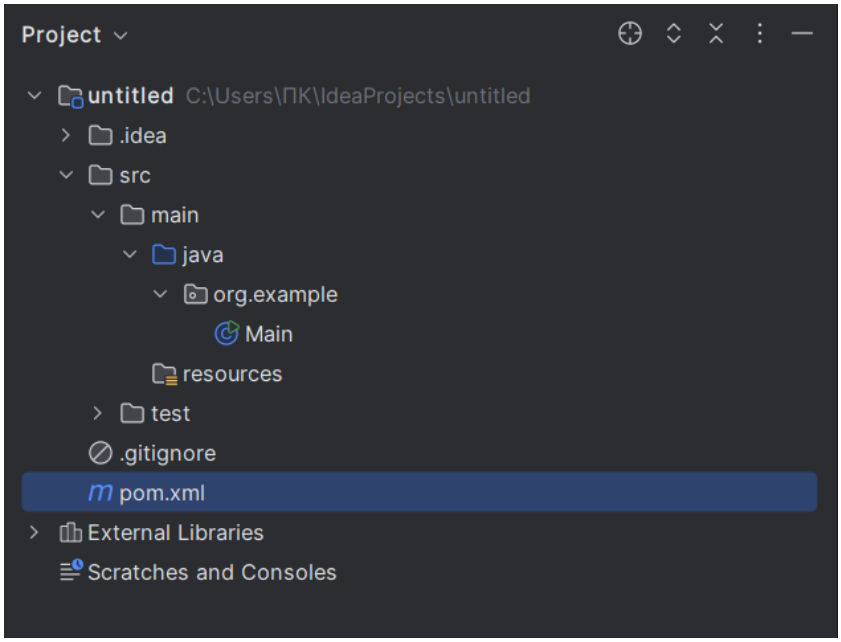

Рисунок 2.10 – Конфігураційний файл “pom”

Після того як вдалося підключити білдер до проєкту, наступним кроком потрібно інтегрувати фреймворк для тестування у це середовище.

Враховуючи використання Maven це можна зробити завдяки сайту https://mvnrepository.com/. Цей вебрепозиторій містить величезну кількість різного роду плагінів та інструментів які допомагають у написанні та організації коду у проєктах на базі Maven. Достатньо зробити запит у пошуку, ввівши назву потрібного інструменту, та обрати потрібну версію. Після цього потрібно скопіювати зміст зі скриптом і вставити його у розділ “dependencies” як окремий “dependency”.

Після створення проєкту на базі Maven і налаштування його структури, наступним етапом є інтеграція Playwright як основного інструмента для автоматизації тестування вебінтерфейсів. Для цього необхідно додати відповідну бібліотеку Playwright до файлу pom.xml, вказавши її у секції залежностей. Це забезпечує завантаження необхідних артефактів і їх подальше використання в проєкті. Після цього потрібно виконати ініціалізацію Playwright у проєкті, створивши базовий клас,

який буде відповідати за запуск браузера, відкриття сесії та її завершення. У цьому класі слід реалізувати методи для ініціалізації браузера, вибору необхідного контексту та сторінки. Такий підхід дозволяє забезпечити повторне використання налаштувань і зменшує дублікацію коду.

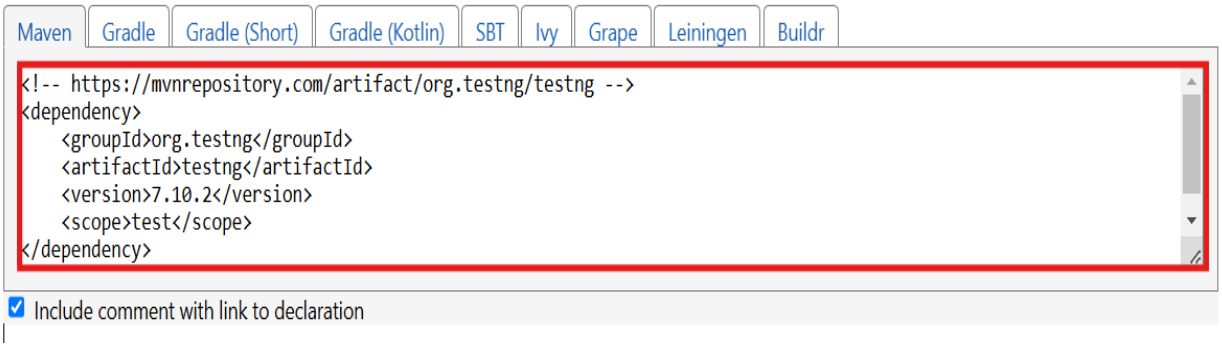

Рисунок 2.11 – Скрипт для підключення TestNG

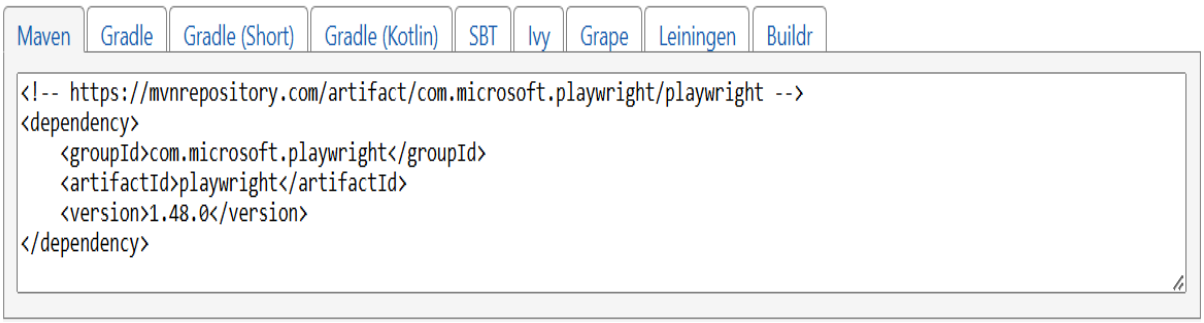

Рисунок 2.12 – Скрипт для підключення Playwright

Для реалізації ініціалізації тестового середовища в рамках тестового фреймворку створюється клас TestInit, який буде відповідати за налаштування тестового оточення перед кожним тестовим методом. У цьому класі реалізується метод із позначкою @BeforeMethod, що дозволяє забезпечити виконання необхідних налаштувань перед запуском кожного тесту. У методі setUp відбувається створення екземпляра Playwright, ініціалізація браузера Chromium у звичайному режимі (без headless) із використанням додаткових аргументів, які відповідають за максимізацію вікна браузера під час запуску. Після цього створюється новий браузерний контекст,

у якому вимикається фіксований розмір viewport, щоб забезпечити адаптацію інтерфейсу до розмірів вікна. Нарешті, ініціалізується нова сторінка, яка стане основою для подальшої взаємодії під час виконання тестів.

Такий підхід дозволяє ізолювати середовище для кожного тесту, забезпечуючи його чистоту та унеможливлюючи вплив результатів одного тесту на інший. Клас TestInit також може містити додаткові методи для завершення роботи браузера після тестів, забезпечуючи звільнення ресурсів і запобігаючи можливим витокам пам’яті.

```
@BeforeMethod
public void setUp() {
    page = Playwright
            .create()
            .chromium()
            .launch(headlessFalse.setArgs(List.of( e1: "--start-maximized")))
            .newContext(new Browser.NewContextOptions().setViewportSize(null))
            .newPage();
}
```

Рисунок 2.13 – Before-метод setUp

Наступним кроком потрібно описати класи що відповідатимуть принципам патерну Page Object Model. Клас AbstractBasePage абстрагує основні дії, що стосуються взаємодії зі сторінками, і надає загальні методи для роботи з елементами DOM. Основна мета цього класу – спрощення реалізації конкретних сторінок проєкту, забезпечуючи їхню стабільність і зменшуючи дублікацію коду.

Клас оголошений як абстрактний, оскільки він не має прямого застосування і слугує основою для створення похідних класів сторінок. У ньому визначено абстрактний метод open, який кожен дочірній клас повинен реалізувати самостійно, залежно від специфіки своєї сторінки. Для роботи з вебсторінками у класі оголошено поле page, яке інкапсулює функціональність Playwright для взаємодії з браузером.

У класі реалізовано метод openUrl, що дозволяє здійснювати навігацію до заданого URL-адресу. Він використовується для відкриття сторінок, забезпечуючи уніфікованість цього процесу в рамках тестів. Конструктор класу приймає об'єкт Page

і призначає його у відповідне поле, забезпечуючи доступ до функціональності Playwright.

Для роботи з елементами DOM реалізовано набір методів, які дозволяють перевіряти їхній стан, наприклад, перевірку наявності, видимості, відсутності чи прихованості. Ці методи базуються на використанні Playwright API і забезпечують стабільність тестів, оскільки дозволяють дочекатися потрібного стану елемента перед подальшою взаємодією з ним. Аналогічно, у класі передбачені методи для отримання списків елементів, що відповідають певному локатору та стану. Це зручно для перевірок і роботи з множинними елементами на сторінці.

Реалізовані методи використовують параметри тайм-ауту, що дозволяє контролювати тривалість очікування, і є гнучкими завдяки можливості приймати різні локатори.

```
9  abstract public class AbstractBasePage<PAGE> {  4 inheritors
10     public Page page;
11
12     public abstract PAGE open();  1 usage  2 implementations
13
14     protected Object openUrl(String url) {  1 usage
15         page.navigate(url);
16         return null;
17     }
18
19     public AbstractBasePage(Page page) { this.page = page; }
22
23     protected ElementHandle isAttachedElement(String locator) {  no usages
24         new Page.WaitForSelectorOptions()
25                 .setState(WaitForSelectorState.ATTACHED)
26                 .setTimeout(15);
27         return page.waitForSelector(locator);
28     }
29
```

Рисунок 2.14 – Реалізація класу AbstractBasePage

Для забезпечення ефективного управління життєвим циклом тестів у рамках проєкту важливим компонентом є реалізація спеціальних слухачів (listeners). Вони

дозволяють виконувати певні дії на різних етапах виконання тестів, таких як їхній початок, завершення, успішне виконання чи провал. У цьому контексті наступним важливим класом є MyListeners, який реалізує інтерфейс ITestListeners з бібліотеки TestNG.

Клас MyListeners відповідає за обробку подій, пов'язаних із виконанням тестів, і надає можливість логувати ці події для полегшення аналізу результатів. Цей клас реалізує інтерфейс ITestListener, який є частиною бібліотеки TestNG і використовується для перехоплення подій під час виконання тестів. Реалізуючи методи інтерфейсу ITestListeners, цей клас забезпечує логування старту й завершення тестів, відображення інформації про успішно виконані, пропущені або невдалі тести. Наприклад, метод onTestStart викликається на початку кожного тесту, тоді як onTestSuccess відображає успішне завершення тесту, а onTestFailure – інформацію про провал.

Логіка виведення інформації реалізована через метод System.out.println, що дозволяє виводити повідомлення в консоль. Це може бути корисним на початкових етапах проєкту для швидкого моніторингу тестів. Кожен із методів отримує контекст або результати тесту, що дозволяє зчитувати їхні назви чи додаткову інформацію, яка стане в пригоді для деталізації звітів.

```
7   public class MyListeners implements ITestListener {  2 usages
8
9   @     public void onStart(ITestContext context) {  no usages
10            System.out.println("Starts Test execution ..... [" + context.getName() + "]");
11        }
12
13  @     public void onFinish(ITestContext context) {  no usages
14            System.out.println("Finish Test execution ..... [" + context.getName() + "]");
15        }
16
```

Рисунок 2.15 – Створення слухачів на прикладі початку та завершення тесту

Інтеграція цього класу в тестовий фреймворк забезпечує кращу прозорість виконання тестів і створює основу для подальшого розширення, наприклад, інтеграції зі сторонніми інструментами для автоматизованого логування чи збереження результатів.

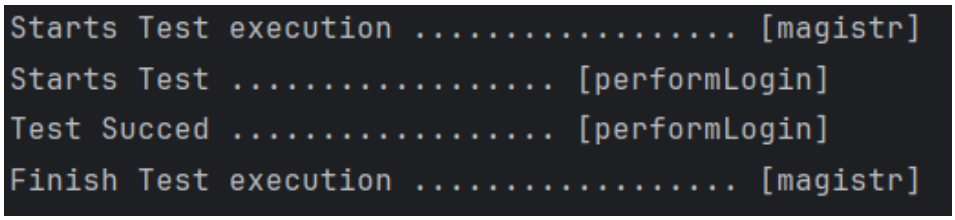

Рисунок 2.16 – Приклад застосування слухачів

На скріншоті вище продемонстровано приклад використання класу MyListeners у тестовому фреймворку. Слухачі, реалізовані через ITestListener, підключені до тестового набору, і це дозволяє відстежувати ключові етапи виконання тестів у режимі реального часу.

У консолі видно, що тестовий запуск розпочався з повідомлення:

“Starts Test execution ..... [magistr]”.

Це вказує на старт виконання тестового набору під назвою “magistr”. Далі, коли виконується конкретний тестовий метод, слухачі генерують повідомлення:

Starts Test ..... [performLogin].

Це свідчить про початок тесту performLogin.

Після успішного завершення цього тесту слухачі фіксують подію успіху, про що говорить повідомлення:

Test Succed ..... [performLogin].

Після виконання всіх тестів у наборі, слухачі логують завершення тестового запуску через повідомлення:

Finish Test execution ..... [magistr].

Загалом клас MyListeners забезпечує прозорий процес тестування. Інформація, яка виводиться, дозволяє відслідковувати статус виконання тестів: які тести

запустилися, завершилися успішно, або зазнали невдачі. У конкретному випадку всі тести виконалися коректно, що підтверджується відсутністю повідомлень про провал (onTestFailure) чи пропущені тести (onTestSkipped).

Для реалізації патерну Page Object Model (POM) потрібно спершу зайнятися створенням класів, які описують вебсторінки та їхні елементи. Початок реалізації цього підходу можна спостерігати на прикладі класу LoginPageElements, що відповідає за елементи сторінки авторизації.

```
8  public class LoginPageElements extends AbstractBasePage {  2 usages  1 inheritor
9      @Override  1 usage
10     public LoginPage open(){
11         openUrl(SAUCEDEMO_URL);
12         return new LoginPage(page);
13 }
14
15     public LoginPageElements(Page page){  1 usage
16         super(page);
17     }
18
19     private static final String SAUCEDEMO_URL = "https://www.saucedemo.com/";  1 usage
20
21     private static final String USERNAME_INPUT = "//input[@id='user-name']";  1 usage
22     private static final String PASSWORD_INPUT = "//input[@id='password']";  1 usage
23     private static final String LOGIN_BTN = "//input[@type='submit']";  1 usage
24
25     protected ElementHandle getUsernameInput(){ return isVisibleElement(USERNAME_INPUT);}  1 usage
26     protected ElementHandle getPasswordInput(){ return isVisibleElement(PASSWORD_INPUT);}  1 usage
27     protected ElementHandle getLoginButton(){ return isVisibleElement(LOGIN_BTN);}  1 usage
28 }
```

Рисунок 2.17 – Реалізація класу LoginPageElements

Клас LoginPageElements є дочірнім класом, який успадковується від базового класу AbstractBasePage. Це забезпечує доступ до загальних методів, таких як openUrl для навігації до сторінки чи методів перевірки стану елементів (наприклад, isVisibleElement), що спрощує роботу з ними.

LoginPageElements містить URL-адресу сторінки авторизації (SAUCEDEMO_URL) та XPath-локатори для основних елементів сторінки – поля введення імені користувача, поля введення пароля та кнопки логіна. Завдяки цьому всі елементи сторінки зберігаються централізовано, що забезпечує легкість у зміні локаторів у разі оновлення інтерфейсу.

У класі реалізовано методи для отримання доступу до цих елементів (getUsernameInput, getPasswordInput, getLoginButton), кожен із яких використовує метод isVisibleElement, щоб дочекатися видимості відповідного елемента перед взаємодією з ним. Це підвищує стабільність тестів, особливо на динамічних сторінках, де елементи можуть не з’являтися одразу.

Метод open реалізує навігацію до сторінки авторизації за допомогою методу openUrl і повертає об’єкт класу LoginPage, що дозволяє переходити до наступного рівня взаємодії із цією сторінкою. Конструктор класу приймає об'єкт Page із Playwright, що забезпечує взаємодію з браузером, і передає його до базового класу через super.

Цей клас є основою для створення сценаріїв тестування сторінки авторизації, дозволяючи тестам виконувати необхідні дії з елементами сторінки, зберігаючи їхню ізольованість і незалежність від змін у внутрішній реалізації.

Наступним кроком для реалізації патерну Page Object Model буде створення класу LoginPage, який розширює функціональність класу LoginPageElements і безпосередньо описує дії, що виконуються на сторінці авторизації. Цей клас є логічним продовженням і реалізує бізнес-логіку, пов’язану зі взаємодією з елементами сторінки.

Клас LoginPage успадковує всі локатори та методи роботи з елементами з батьківського класу LoginPageElements. Це дозволяє уникнути дублювання коду та концентруватися лише на створенні специфічних дій, які необхідно виконати на цій сторінці.

```
7   public class LoginPage extends LoginPageElements {  7 usages
8
9       public LoginPage(Page page) {  2 usages
10          super(page);
11      }
12
13      @Step("Виконати логін")  1 usage
14      public LoginPage doLogin(){
15          getUsernameInput().fill( value: "standard_user");
16          getPasswordInput().fill( value: "secret_sauce");
17          getLoginButton().click();
18          return this;
19      }
20  }
21
```

Рисунок 2.18 – Реалізація класу LoginPage

Основною функцією класу є метод doLogin, який реалізує сценарій авторизації. Метод заповнює поля введення імені користувача та пароля за допомогою методів getUsernameInput і getPasswordInput, після чого натискає кнопку логіна через getLoginButton. Вхідні дані, як-от standard_user і secret_sauce, у цьому випадку задані як статичні, але в реальних сценаріях вони можуть бути параметризовані для тестування різних облікових записів.

Для підвищення читабельності звітів інтегровано анотацію @Step із бібліотеки Allure. Вона дозволяє додати до звітів інформацію про виконання цього кроку (“Виконати логін”), що робить звіти більш зрозумілими та деталізованими.

Метод doLogin повертає поточний об'єкт LoginPage, дозволяючи створювати ланцюжки викликів (chain of calls) у тестових сценаріях. Це робить код більш лаконічним і простим у використанні.

Клас LoginPage забезпечує ізоляцію логіки дій на сторінці від самих тестів, що дозволяє змінювати або розширювати функціонал сторінки без впливу на тестові сценарії, що використовують цей клас. Таким чином, цей підхід дотримується принципів модульності та повторного використання коду, закладених у патерн Page Object Model.

Останнім елементом у реалізації патерну Page Object Model є тестовий клас MyTests, який безпосередньо використовує сторінкові об’єкти та реалізує тестові сценарії.

```
@Listeners(MyListeners.class)
public class MyTests extends TestInit {

    @Test
    public void performLogin(){
        LoginPage loginPage = new LoginPage(page);
        InventoryPage inventoryPage = new InventoryPage(page);

        loginPage
                .open()
                .doLogin();

        Assert.assertTrue(inventoryPage.getAppLogo().isVisible());
    }
}
```

Рисунок 2.19 – Тестовий клас MyTests

Клас MyTests успадковується від TestInit, забезпечуючи доступ до базової конфігурації та ініціалізації браузера, яка виконується в @BeforeMethod. Також до класу підключено слухачі MyListeners через анотацію @Listeners, що дозволяє вести логування процесу виконання тестів у реальному часі.

У тестовому методі performLogin реалізовано сценарій перевірки авторизації. На першому етапі створюються об'єкти LoginPage і InventoryPage, які відповідають за сторінку авторизації та наступну сторінку після успішного входу. Вони отримують у конструкторі об'єкт Page, що забезпечує взаємодію з браузером через Playwright.

Тест починається із виклику методу open, який відкриває сторінку авторизації, а потім викликається метод doLogin, що виконує логін із фіксованими обліковими даними. Після цього перевіряється успішність входу за допомогою асертації:

Assert.assertTrue(inventoryPage.getAppLogo().isVisible());

Ця перевірка визначає, чи став логотип програми видимим на сторінці InventoryPage, що є підтвердженням успішного входу.

Такий підхід дозволяє зробити тест максимально зрозумілим і структурованим. Всі дії на сторінках ізольовані у відповідних класах LoginPage та InventoryPage, тому тестовий метод фокусується лише на сценарії, а не на деталях взаємодії з вебелементами.

Клас MyTests завершує реалізацію патерну Page Object Model, демонструючи повну інтеграцію його елементів: структуризацію сторінок, абстрагування логіки, використання слухачів для моніторингу та стабільну ініціалізацію тестового середовища. Це дозволяє легко підтримувати тести, масштабувати їх, а також підвищує їхню стабільність і зрозумілість.

Таким чином, на прикладі просто тестового сценарію авторизації було розглянути приклад реалізації патерну Page Object Model. Було показано, як створення структурованих класів для роботи з елементами сторінки (LoginPageElements), бізнес-логікою сторінки (LoginPage) та безпосередньо тестових сценаріїв (MyTests) дозволяє досягти високого рівня ізоляції коду, його повторного використання та легкості у підтримці. Кожен клас виконує свою чітко визначену роль: базові класи забезпечують спільну функціональність, сторінкові об’єкти описують специфіку взаємодії зі сторінками, а тести концентруються на перевірці функціональності без занурення в деталі роботи з елементами. Така структура не лише підвищує зрозумілість і зручність у розробці тестів, але й дозволяє легко адаптувати їх до змін у додатку, що є основною перевагою використання патерну Page Object Model у автоматизованому тестуванні.

Для покращення організації тестів і зручності їх підтримки, можна розширити представлену структуру, розділивши тести на класи відповідно до логічних груп або функціональних модулів. Наприклад, окремі класи тестів можуть бути створені для перевірки авторизації, управління товарами чи перевірки профілю користувача. Такий підхід дозволить не лише чітко відокремити тести за призначенням, але й зробить звіти більш структурованими, оскільки кожна група тестів буде відображати свої специфічні сценарії.

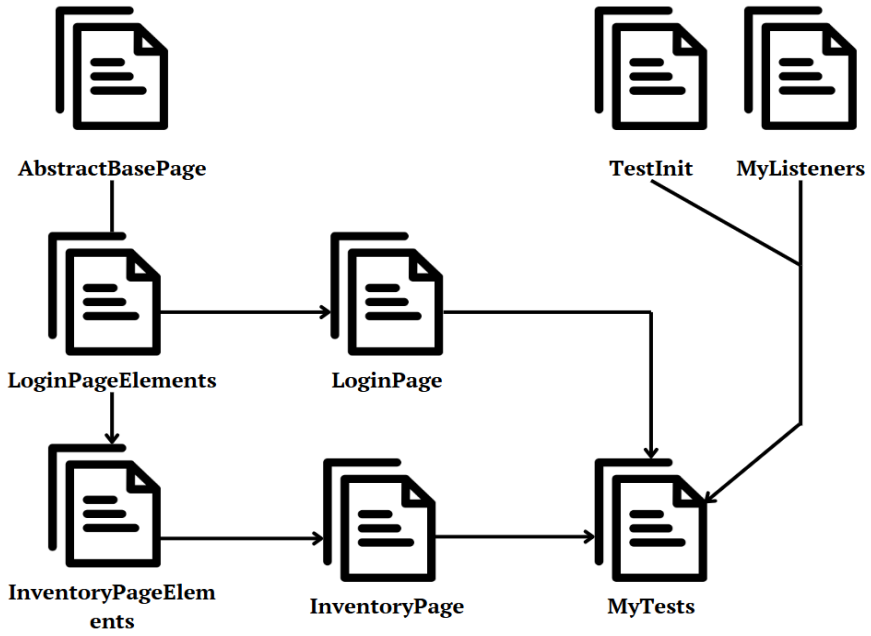

Рисунок 2.20 – Схема роботи патерну на прикладі розглянутого тестового сценарію

Крім того, для кожної групи тестів можна додати окремі базові класи, які забезпечать специфічну ініціалізацію або передумови для їх виконання, що зменшить дублювання коду. Наприклад, для тестів, що перевіряють функціонал товарів, можна створити спеціалізований базовий клас із методами для додавання товарів у кошик або роботи з категоріями. Така деталізація дозволить підвищити масштабованість і спростить адаптацію нових тестів.

Візуально це може виглядати як розширення ієрархії класів, подібно до представленої на діаграмі, де кожна група тестів отримує свій набір класів, зберігаючи основні принципи повторного використання і абстрагування, притаманні патерну Page Object Model.

Далі потрібно налаштувати звітність створеного фреймворку за допомогою Allure Reporter, який згадувався раніше. Щоб це зробити потрібно додати новий код у класі TestInit, що відповідає за інтеграцію механізму створення звітів із системою тестування.

Для підключення Allure до проєкту необхідно виконати декілька кроків. Спершу потрібно додати залежність Allure TestNG у файл pom.xml, щоб забезпечити інтеграцію із TestNG:

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/io.qameta.allure/allure-maven -->
<dependency>
    <groupId>io.qameta.allure</groupId>
    <artifactId>allure-maven</artifactId>
    <version>2.8</version>
</dependency>
```

Рисунок 2.21 – Скрипт для підключення Allure

Після цього необхідно налаштувати механізм збору даних для звітів Allure, додавши відповідні анотації, як-от @Step, у ключові методи класів, щоб забезпечити відображення основних дій тестового сценарію у звіті.

```
@AfterMethod
public void voidtearDown(ITestResult result) {
    int status = result.getStatus();
    if (status == ITestResult.SKIP || status == ITestResult.FAILURE) {
        byte[] screenshot = page.screenshot(new Page.ScreenshotOptions());
        Allure.addAttachment( name: "Test" + result.getMethod().getMethodName(), type: "image/jpeg",
                new ByteArrayInputStream(screenshot), fileExtension: "jpeg");
    }

}
```

Рисунок 2.22 – Метод voidtearDown який відповідає за генерацію звіту зі скріншотами у Allure Report

У методі @AfterMethod, який знаходиться у класі InitTest, реалізовано збереження скріншотів у звітах Allure у випадках, коли тест завершується статусом “SKIP” або “FAILURE”.

Після завершення кожного тесту виконується перевірка статусу результату через об'єкт ITestResult. Якщо тест пропущено або завершився помилкою, фреймворк знімає скріншот за допомогою методу page.screenshot, що надається Playwright. Отриманий скріншот зберігається у вигляді масиву байтів, який потім додається до звіту Allure за допомогою методу Allure.addAttachment. У додатку вказується ім'я тесту, формат зображення та сам вміст скріншота. Це дозволяє деталізувати звіти та отримувати наочну інформацію про стан вебсторінки в момент збою або пропуску тесту.

Для того щоб переглянути сформований звіт по пройдених тестах, потрібно скористатися командою allure serve. Ця команда генерує інтерактивний HTML-звіт на основі зібраних результатів тестів і автоматично відкриває його у веббраузері.

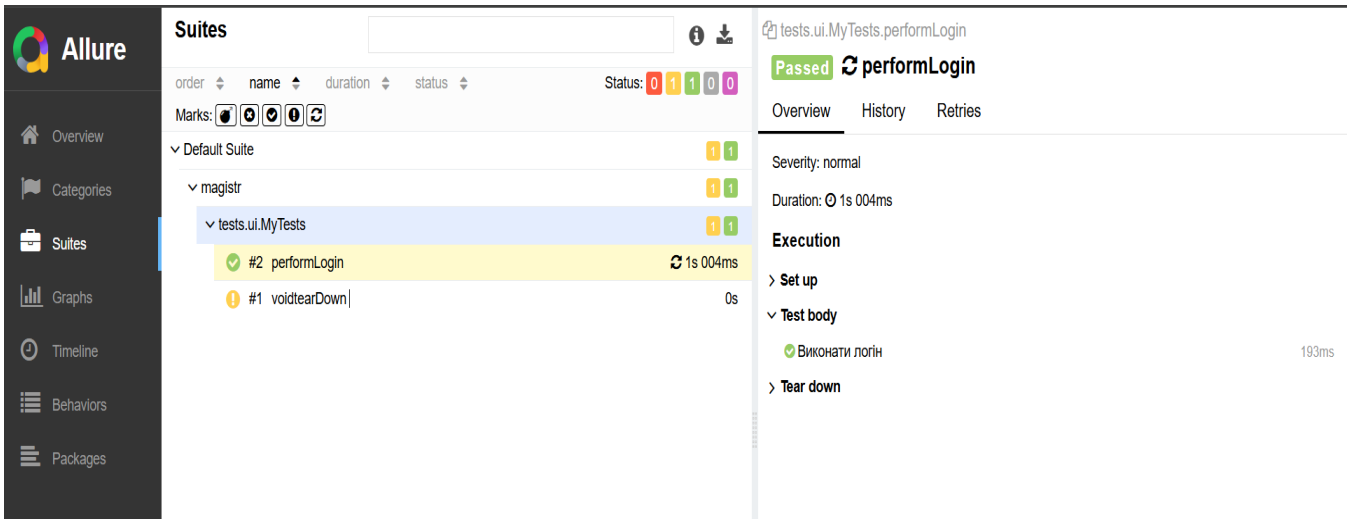

Рисунок 2.23 – Приклад звіту Allure Report

2.6. Організація тестування та налагодження програмного засобу

У процесі створення фреймворку для автоматизованого тестування веборієнтованих користувацьких інтерфейсів було реалізовано комплексний підхід до тестування його компонентів. Основними цілями тестування стали перевірка коректності роботи створених класів і методів, забезпечення їхньої взаємодії згідно з

концепцією патерну Page Object Model, а також виявлення можливих недоліків у реалізації фреймворку.

Для перевірки працездатності фреймворку було використано функціональне тестування, що охоплювало тестові сценарії авторизації на тестовому ресурсі (наприклад, https://www.saucedemo.com/). [21] Основними інструментами для тестування стали бібліотека Playwright для взаємодії з вебінтерфейсом, TestNG для організації тестових наборів, а також Allure Reporter для формування звітів про виконання тестів.

Методи тестування були реалізовані через класи та методи, що відображають бізнес-логіку вебдодатка, такі як LoginPage, InventoryPage та інші. Особливу увагу приділено налаштуванню методу setUp у класі TestInit, який забезпечує ініціалізацію тестового середовища, а також методу tearDown, який дозволяє збирати скріншоти помилок і інтегрувати їх у звіти Allure.

Оцінка успішності тестування фреймворку проводилася за наступними критеріями:

- стабільність виконання тестів;
- коректність виконання тестів;
- зручність використання;
- масштабованість.

Результатом тестування стало успішне виконання ключових тестових сценаріїв, таких як авторизація користувача, перевірка доступності елементів інтерфейсу та перевірка переходів між сторінками.

Генеровані звіти Allure надавали детальну інформацію про кожен тест, зокрема, статус його виконання, часові характеристики, логи та прикріплені скріншоти помилок. Такі звіти дозволили швидко виявляти й усувати недоліки у процесі розробки фреймворку.

2.7 Аналіз отриманих результатів дослідження, рекомендації щодо використання та впровадження

Щоб оцінити продуктивність розробленого фреймворку було виконано серію тестових запусків з метою вимірювання ключових метрик:

1. Час виконання тестів.
2. Споживання пам’яті.
3. Стабільність роботи.

Таблиця 2.1

Таблиця метрик оцінювання

Метрика	Результат
Час виконання одного тесту	Від 1 до 3 сек
К-ть затраченої пам’яті	До 200 Мб
К-ть помилок за 30 виконань	0

Аналіз результатів:

Час: результати тестування продемонстрували стабільний час виконання, який відповідає очікуванням для вебтестів із використанням Playwright.

Пам’ять: обсяг оперативної пам’яті знаходиться в межах прийнятних значень для автоматизованого тестування з обробкою скріншотів і звітів.

Стабільність: відсутність збоїв протягом 30 тестових запусків свідчить про повну відсутність так званих flaky тестів.

Загалом аналіз результатів надав очікувану картину стосовно метрик для оцінки продуктивності тестів. Дана ефективність забезпечується завдяки використанню

патерну Page Object Model та інструменту Playwright який значно швидший в порівнянні зі старішими аналогами.

Ризики. При збільшенні кількості та навантаження тестів буде відповідно збільшуватись к-ть часу потрібна для проведення тестів та обсяг оперативної пам’яті. Також виникає ризик появи flaky тестів при масштабованості.

Розроблений фреймворк може бути впроваджений у процес розробки вебдодатків, особливо для проєктів із регулярним оновленням функціональності. Його використання дозволить автоматизувати рутинні перевірки інтерфейсу (регресію) та підвищити якість продукту загалом.

Умови експлуатації. Для забезпечення коректної роботи фреймворку необхідні такі умови:

- апаратні засоби:

процесор із частотою від 2.5 ГГц (4 ядра);

оперативна пам’ять не менше 4 ГБ;

накопичувач SSD із вільним обсягом не менше 10 ГБ.

- програмні засоби:

операційна система: Windows 10, MacOS 11 або Ubuntu 20.04 і новіші;

Java Development Kit (JDK) 11 або вище;

середовище розробка (IDE), рекомендовано IntelliJ IDEA;

менеджер залежностей Maven;

інструменти для інтеграції Allure Reporter.

Рекомендації щодо впровадження. Фреймворк може бути інтегрований у CI/CD системи (наприклад, Jenkins, GitLab CI) для регулярного виконання тестів після кожного оновлення коду. Для підвищення масштабованості рекомендується використовувати контейнеризацію (Docker) та хмарні середовища для розподіленого виконання тестів. Покращити якість звітності можна шляхом додавання користувацьких метрик у звіти Allure.

ВИСНОВКИ

У результаті виконаного дослідження було удосконалено методи розробки та організації фреймворку для автоматизованого тестування веборієнтованих користувацьких інтерфейсів. В процесі роботи було проведено аналіз існуючих підходів до автоматизованого тестування, визначено основні проблеми, такі як низька гнучкість традиційних фреймворків, недостатня інтеграція сучасних інструментів і відсутність підтримки кросплатформенності. На основі цього розроблено власний фреймворк із використанням технологій Java, Maven, TestNG, Playwright та Allure. Запропонований підхід дозволив суттєво підвищити ефективність процесу тестування завдяки можливості створення масштабованих, легко підтримуваних та адаптивних тестових сценаріїв.

Науковим результатом дослідження є розробка вдосконаленої методики організації автоматизованого тестування веб інтерфейсів, що здатний забезпечити високу точність, стабільність і зручність у використанні та підтримці. Практичне значення роботи полягає в тому, що створений фреймворк може бути впроваджений у процеси розробки програмного забезпечення та сприятиме підвищенню якості продукту та прискоренню релізів. Експериментальне тестування довело, що використання фреймворку зменшує час на тестування функціональності на ~30%, мінімізує кількість людських помилок і забезпечує стабільну роботу інтерфейсів на різних платформах.

Водночас, під час виконання роботи було виявлено низку недоліків. Зокрема деякі оновлені версії використаних інструментів можуть конфліктувати з собою. Це не завжди призводить до повної відмови роботи фреймворку, проте може мати вплив на його ефективність та зрозумілість. Ці недоліки можуть бути усунені з часом самими розробниками цих технологій, а також шляхом подальшого вдосконалення механізмів оптимізації взаємодії та розробки універсальних конфігурацій для інтеграції у складні проєкти.

Для подальшого розвитку цієї розробки рекомендовано розглянути залучення CI/CD процесів для розгортання тестів на віддалених серверах. Це дасть змогу автоматично виконувати перевірку вже реалізованих компонентів ПЗ (регресія) після кожного релізу. Також доцільно вивчити можливості розширення фреймворку для роботи з іншими мовами програмування та інструментами, що дозволить зробити його універсальним для різних доменів. Отримані результати створюють базу для розвитку нових методів автоматизації тестування, що відповідають сучасним вимогам ринку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Автоматизоване тестування - QALight. URL: https://qalight.ua/baza-znaniy/avtomatizovane-testuvannya/ (дата звернення: 04.11.2024).
2. Види тестування та відмінності між ними. Шпаргалка з тестування. QualityAssuranceGroup. URL: https://qagroup.com.ua/publications/vydy-testuvannya-ta-vidminnosti-mizh-nymy/ (дата звернення: 06.11.2024).
3. Ким бути: мануальним тестувальником чи автоматизатором? | Онлайн-курси від компанії QATestLab. Онлайн-курси від компанії QATestLab | Головна сторінка. URL: https://training.qatestlab.com/blog/helpful-materials/manual-tester-or-automatization-tester/ (дата звернення:06.11.2024).
4. Леонов О. Автоматичне тестування ПЗ (визначення, процес створення). Друкарня. URL: https://drukarnia.com.ua/articles/avtomatichne-testuvannya-pz-viznachennya-proces-stvorennya-WWNqn (дата звернення: 06.11.2024).
5. Обираємо мову програмування для автоматизації тестування: головні принципи і розбір кейсу. ITC.ua. URL: https://itc.ua/ua/blogs/obyrayemo-movu-programuvannya-dlya-avtomatyzatsiyi-testuvannya-golovni-pryntsypy-i-rozbir-kejsu/ (дата звернення: 12.11.2024).
6. Огляд видів тестування. Онлайн-курси від компанії QATestLab | Головна сторінка. URL: https://training.qatestlab.com/blog/technical-articles/review-the-types-of-testing/ (дата звернення: 12.11.2024).
7. Огляд мов програмування для автоматизованого тестування. Онлайн-курси від компанії QATestLab | Головна сторінка. URL: https://training.qatestlab.com/blog/technical-articles/overview-of-programming-languages-for-automated-testing/ (дата звернення: 14.11.2024).
8. Роль тестування в процесі розробки ПЗ - QALight. QALight. URL: https://qalight.ua/baza-znaniy/rol-testuvannya-v-protsesi-rozrobki-pz/ (дата звернення:14.11.2024).

9. Page Object - простий приклад з використанням Selenide. *IT-notes*. URL: https://www.it-notes.wiki/automation-testing/page-object-simple-example-using-selenide/ (дата звернення: 16.11.2024).
10. Page Object Model pattern for effective automation testing. *Spyrosoft*. URL: https://spyro-soft.com/developers/page-object-model-pattern-for-effective-automation-testing (дата звернення: 17.11.2024).
11. Фреймворк в програмуванні - QALight. QALight. URL: https://qalight.ua/baza-znaniy/frejmvork-v-programuvanni/ (дата звернення: 17.11.2024).
12. Що таке POM (Page Object Model)?. QualityAssuranceGroup. URL: https://qagroup.com.ua/publications/shcho-take-pom-rage-object-model/ (дата звернення: 17.11.2024).
13. Що таке TestNG?. QualityAssuranceGroup. URL: https://qagroup.com.ua/publications/what-is-testng/ (дата звернення: 17.11.2024).
14. Beekeeper R. JavaRush. URL: https://javarush.com/ua/groups/posts/uk.3119.vse-jsho-vi-khotli-znati-pro-maven---java-proekt-vd-a-do-ja (дата зверення: 18.11.2024).
15. Fast and reliable end-to-end testing for modern web apps | Playwright. URL: https://playwright.dev/ (дата зверення: 18.11.2024).
16. Introduction. Allure Report – Open-source HTML test automation report tool. URL: https://allurereport.org/docs/ (дата звернення: 18.11.2024).
17. Java IntelliJ IDEA. Уроки для початківців. W3Schools українською. W3Schools українською. Безплатні уроки онлайн для початківців, школярів та студентів. URL: https://w3schoolsua.github.io/hyperskill/intellij-idea.html#gsc.tab=0 (дата звернення: 18.11.2024).
18. KISS, DRY Ñ YAGNI. Teletype. URL: https://blog.ohyr.dev/kiss-yagni-dry (дата зверення: 18.11.2024).
19. KISS, DRY, S.O.L.I.D, YAGNI – навіщо дотримуватись принципів програмування?. Только самое интересное. Форум программистов. Ит Новости. Ит юмор. Ит статьи. Новости It компаний, отзывы. senior.ua. URL:

https://senior.ua/articles/kiss-dry-solid-yagni--navscho-dotrimuvatis-principv-programuvannya (дата звернення: 18.11.2024).

20. Selenide: лаконичные и стабильные UI тесты на Java. URL: https://ru.selenide.org/documentation/page-objects.html (дата звернення: 20.11.2024).
21. Swag labs. *Swag Labs*. URL: https://www.saucedemo.com/ (дата зверення: 20.11.2024).

ДОДАТКИ

Додаток А

Технічне завдання

Мета розробки

Створення фреймворку для автоматизованого тестування веб-орієнтованих користувацьких інтерфейсів, який забезпечить підвищення ефективності та якості тестування, підтримку багатобраузерності, кросплатформенність і інтеграцію з CI/CD.

Вимоги до фреймворку

Фреймворк повинен:

- підтримувати автоматизацію тестування веб-додатків;
- забезпечувати стабільне виконання тестів у середовищах Windows;
- автоматично формувати звіти про результати тестування з використанням Allure Reporter;
- мати модульну архітектуру для розширення функціоналу.

Функціональні вимоги

- виконання функціональних, регресійних та UI-тестів;
- генерація інтерактивних звітів із кроками виконання тестів, скріншотами помилок і тривалістю виконання;
- підтримка параметризації для динамічного тестування різних наборів даних;
- взаємодія з браузерами через Playwright API.

Апаратні вимоги

Для забезпечення коректної роботи фреймворку необхідні:

- процесор із частотою від 2.5 ГГц (4 ядра);
- оперативна пам’ять не менше 4 ГБ;
- накопичувач SSD із вільним обсягом не менше 10 ГБ.

Програмні вимоги

- операційна система: Windows 10, MacOS 11 або Ubuntu 20.04 і новіші;
- Java Development Kit (JDK) версії 11 або вище;
- менеджер залежностей Maven для управління бібліотеками та збірки проєкту;
- інструменти для інтеграції звітності: Allure Reporter;
- середовище розробки (IDE): рекомендовано IntelliJ IDEA.

Етапи розробки

1. Підготовка середовища:

- Налаштування робочого середовища з використанням рекомендованого програмного забезпечення.
- Встановлення та конфігурація залежностей через Maven (pom.xml).
- Інсталяція Playwright та необхідних браузерів.

2. Розробка фреймворку:

- Створення структури проєкту, включаючи модулі для тестів, конфігурації та звітності.
- Розробка базових класів для взаємодії з Playwright і TestNG.
- Інтеграція з Allure Reporter для автоматичної генерації звітів.

3. Написання тестових сценаріїв:

- Створення прикладів тестів для перевірки функціоналу, валідації UI та поведінки веб-додатків.
- Реалізація параметризації для багаторазового виконання тестів із різними наборами даних.

4. Тестування та оптимізація:

- Проведення перевірки працездатності фреймворку у визначених середовищах.
- Оптимізація швидкодії тестів та усунення можливих помилок.

Очікувані результати

Результатом виконання проєкту стане фреймворк, який:

- дозволяє автоматизувати тестування веб-інтерфейсів у сучасних браузерах;
- генерує наочні звіти про виконання тестів;
- підтримує масштабованість і розширення функціоналу;
- легко інтегрується в CI/CD конвеєри для безперервного тестування.

Обмеження

- фреймворк підтримуватиме лише браузери, сумісні з Playwright API;
- виконання тестів можливе лише на платформах із зазначеними апаратними і програмними характеристиками.

Додаток Б

Інструкція користувачу

Загальні відомості

Програма являє собою автоматизований фреймворк для тестування веборієнтованих користувацьких інтерфейсів. Фреймворк створений для підвищення ефективності тестування шляхом автоматизації виконання тестових сценаріїв, генерування звітів і забезпечення підтримки кросбраузерного тестування.

Функціональне призначення

Фреймворк призначений для розв’язання задач автоматизованого тестування веб-додатків. Основні завдання, які можна виконувати:

- автоматичне тестування функціоналу користувацьких інтерфейсів;
- генерування звітів про результати тестування;
- інтеграція з процесами CI/CD для забезпечення безперервної перевірки якості.

Умови застосування програми

Апаратні вимоги:

- процесор із частотою від 2.5 ГГц (4 ядра);
- оперативна пам’ять не менше 4 ГБ;
- накопичувач SSD із вільним місцем не менше 10 ГБ.

Програмні вимоги:

- операційна система: Windows 10, MacOS 11 або Ubuntu 20.04 і новіші;
- Java Development Kit (JDK) 11 або вище;
- середовище розробки (рекомендовано IntelliJ IDEA);
- менеджер залежностей Maven;
- інструмент для генерування звітів Allure Reporter.

Повідомлення оператору

У разі виникнення помилок під час роботи програми видаються такі повідомлення:

- “Configuration file not found” – файл конфігурації не знайдено. Перевірте наявність файлу config.properties у кореневій директорії.
- “Dependency download error” – помилка завантаження залежностей Maven. Перевірте інтернет-з’єднання та конфігурацію pom.xml.
- “Test execution failed” – виконання тестів завершилося невдачею. Ознайомтеся з логами тестів для діагностики помилок.
- “Allure report generation failed” – не вдалося згенерувати звіт Allure. Перевірте правильність інтеграції Allure із Maven.

У кожному з випадків необхідно виконати дії, зазначені в повідомленні, або звернутися до системного адміністратора чи розробника.

Опис роботи програми

Фреймворк дозволяє виконувати автоматизоване тестування за такою послідовністю:

1. Підготовка середовища:

- Встановіть JDK, IntelliJ IDEA, Maven і Allure.
- Клонуйте репозиторій із фреймворком.
- Відкрийте проєкт у IntelliJ IDEA і завантажте залежності через Maven.

2. Конфігурація фреймворку:

- У файлі config.properties задайте URL тестового середовища, браузер і параметри тестування.
- Перевірте коректність версій залежностей у pom.xml.

3. Запуск тестів:

У IntelliJ IDEA запустіть окремий тестовий клас або метод.

Для запуску з командного рядка використовуйте:

mvn clean test

Після запуску тестів повинна появитися панель яка відображатиме перебіг цього процесу, приклад панелі наведено на рисунку нижче.

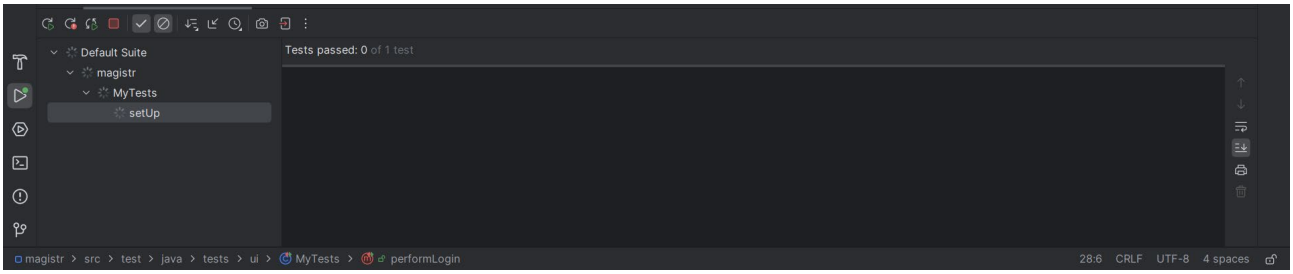

Рисунок Б.1 – Запуск тестів

Під час проходження автотестів програма самотужки відкриє браузер та почне відтворювати кроки, що прописані у скриптах. Цей процес автоматизований тому втручатися в нього не варто.

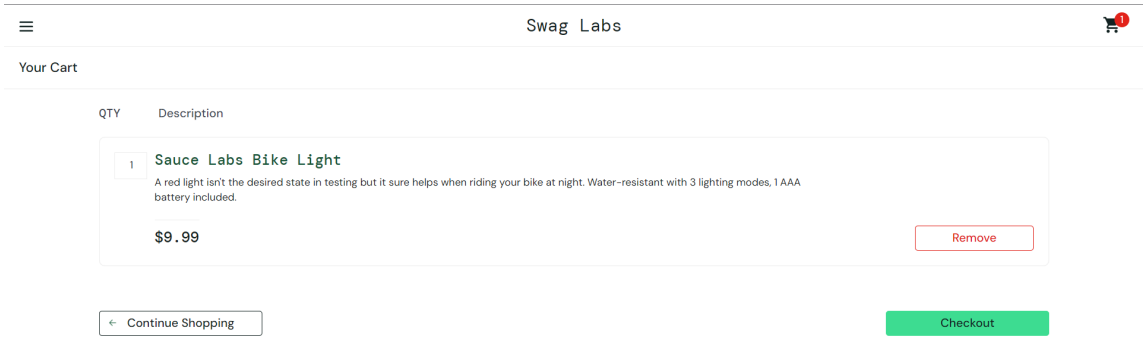

Рисунок Б.2 – Програма відкрила браузер та відтворює кроки тестового сценарію

Коли процес буде завершено, у нижній панелі в середовищі Intellij IDEA буде відображено результати запуску тестів. Там буде вказано к-ть тестів розділені на наступні категоріїї: Пройдено, провалено або пропущено.

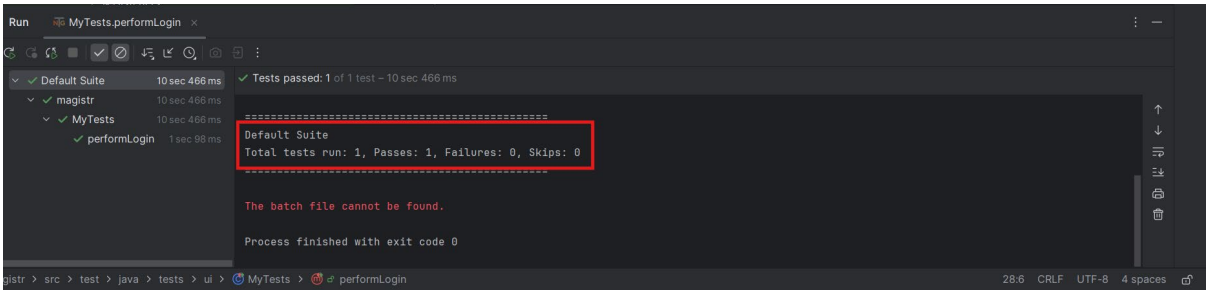

Рисунок Б.3 – Результати проходження тестів

Щоб скористатись додатком Allure Report потрібно прописати в терміналі наступну команду:

allure serve target/allure-results

Вона відкриє сторінку в браузері з програмою Allure яка відобразить результати запуску останніх тестів.

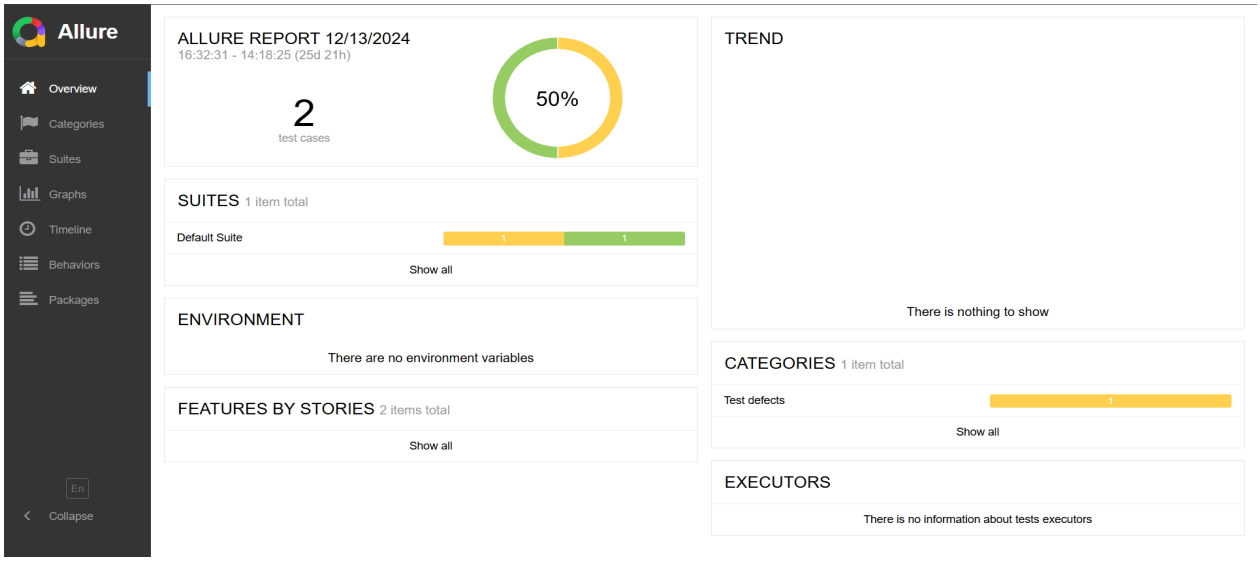

Рисунок Б.4 – Відображення інструменту звітності Allure

4. Створення нових тестів:

Для написання нових тестових сценаріїв потрібно внести зміни у відповідні файли середовища. Локатори потрібно добавити у класи де містяться загальні елементи відповідної сторінки, у класі що наслідує попередній потрібно прописати відповідні дії, що виконуються на основі тої інформації що міститься в батьківському класі. Наприкінці потрібно дописати новий тест за допомогою спеціальної анотації TestNG – @Test. В ньому вказати об’єкти потрібних класів та описати виконання потрібних кроків.

АНОТАЦІЯ

Скорук Д. А. Удосконалення методів розробки та організації фреймворку для автоматизованого тестування веборієнтованих користувацьких інтерфейсів. Рукопис.

Кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 Комп'ютерні науки. Волинський національний університет імені Лесі Українки, Луцьк, 2024 р.

У кваліфікаційній роботі представлено результати дослідження методів та підходів до розробки і організації фреймворку для автоматизованого тестування веборієнтованих користувацьких інтерфейсів. З урахуванням сучасних тенденцій у розробці програмного забезпечення визначено основні проблеми, що виникають під час створення тестових фреймворків, та запропоновано шляхи їх вирішення.

Мета роботи – підвищення ефективності та якості автоматизованого тестування шляхом удосконалення методів розробки фреймворків, забезпечення їхньої модульності, масштабованості та інтеграції у сучасні конвеєри CI/CD.

У рамках дослідження реалізовано фреймворк на основі стека технологій Java, Maven, TestNG, Playwright і Allure. Розроблений фреймворк забезпечує інтерактивне генерування звітів, кросбраузерне тестування та легке налаштування конфігурацій. Практичне впровадження продемонстровано на прикладах тестування функціональних сценаріїв у вебдодатках.

Основними науковими та практичними результатами роботи є:

- розробка гнучкої архітектури фреймворку для автоматизованого тестування;
- підвищення точності тестування через використання Playwright API;
- спрощення аналізу результатів тестування завдяки інтеграції з Allure Reporter;

Результати дослідження можуть бути використані у сфері розробки вебдодатків для забезпечення якості продуктів, оптимізації роботи команд QA та прискорення циклів релізів. Подальші дослідження можуть бути спрямовані на адаптацію фреймворку для мобільних платформ і вдосконалення процесів інтеграції з хмарними сервісами.

Ключові слова: автоматизоване тестування, фреймворк, веборієнтовані інтерфейси, Playwright, TestNG, Allure, CI/CD.

ANNOTATION

Skoruk D. A. Improvement of Methods for Developing and Organizing a Framework for Automated Testing of Web-Based User Interfaces. Manuscript.

Qualification work for obtaining the educational degree of "Master" in the specialty 122 Computer Science. Lesya Ukrainka Volyn National University, Lutsk, 2024.

The qualification work presents the results of research on methods and approaches for developing and organizing a framework for automated testing of web-based user interfaces. Considering modern trends in software development, the main challenges in creating testing frameworks have been identified, and solutions to address these issues have been proposed.

The aim of the study is to enhance the efficiency and quality of automated testing by improving framework development methods, ensuring modularity, scalability, and integration into modern CI/CD pipelines.

Within the research, a framework was implemented using the technology stack of Java, Maven, TestNG, Playwright, and Allure. The developed framework provides interactive report generation, cross-browser testing, and easy configuration. Practical implementation is demonstrated with examples of testing functional scenarios in web applications.

The main scientific and practical results of the work include:

- development of a flexible architecture for automated testing frameworks;
- improved test accuracy using the Playwright API;
- simplified result analysis through integration with Allure Reporter;
- reduced testing time due to parallel test execution.

The research outcomes can be applied in the field of web application development to ensure product quality, optimize QA team workflows, and accelerate release cycles. Future research may focus on adapting the framework for mobile platforms and improving integration processes with cloud services.

Keywords: automated testing, framework, web-based interfaces, Playwright, TestNG, Allure, CI/CD.