Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет __Комп’ютерної інженерії та управління___________________
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти____перший (бакалаврський)_____________________
(рівень вищої освіти)

Інфраструктура автоматизації тестування

програмних продуктів
(тема)

Виконав:
здобувач ____4_____ року навчання,
групи КІУКІ-21-7

__________Омельницький А. А.____________
(прізвище, ініціали)

Спеціальність *123 _Комп’ютерна інженерія*

Тип програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма________________________

_____*Комп’ютерна інженерія* _____________
(повна назва освітньої прграми)

Керівник _проф. Чумаченко С.В._
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ___________ _Чумаченко С.В
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет________Комп'ютерної інженерії та управління ________________

Кафедра_______Автоматизації проектування обчислювальної техніки______

Рівень вищої освіти____перший (бакалаврський)_________________________

Спеціальність_____123_Комп'ютерна інженерія _______________________

(шифр і назва)

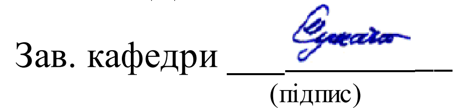
Тип програми ________Освітньо-професійна____________________________

(освітньо-професійна або освітньо-наукова)

Освітня програма ______Комп'ютерна інженерія _________________________

(повна назва

ЗАТВЕРДЖУЮ:

Зав. кафедри __________

(підпис)

«__02__»___05_____ 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту_____Омельницькому Андрію Анатолійовичу___________________

(прізвище, ім'я, по батькові)

1. Тема роботи (проекту) Інфраструктура автоматизації тестування програмних продуктів ____

затверджена наказом по університету від від " 21 "_____05______2025 р. № 403 Ст_____

2. Термін подання студентом роботи до екзаменаційної комісії ______10.06.2025_______

3. Вихідні дані до роботи (проекту)____

Діагностика програмних продуктів

Середовище розробки «InteliJ IDEA»

Selenium, Web-driver

Інструмент безперервної інтеграції «Jenkins»

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):

Програмна реалізація діагностичної інфраструктури

Програмування автоматизованих тестів

Інтеграція створеної інфраструктури із «Jenkins»

Інтеграція інфраструктури із «Zephyr Scale»

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) __15__слайдів___________________________

____
____
____
____

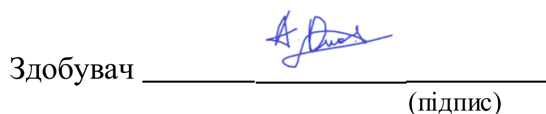
6. Консультанти розділів роботи (проекту)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

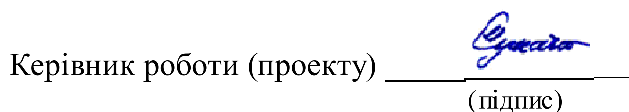
КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Термін виконання етапів проекту (роботи)	Примітка
1	Видача теми проекту, узгодження і затвердження теми	05.05.2025 -06.05.2025	
2	Аналіз проблемної галузі, постановка задачі, вибір інструментальних засобів	06.05.2025 -08.05.2025	
3	Створення та розгортання проекту	08.05.2025 -10.05.2025	
4	Розробка структури класів системи	10.05.2025 -12.05.2025	
5	Інтеграція інфраструктури із зовнішніми інструментами	12.05.2025 -14.05.2025	
6	Розробка автоматизованих тестів	14.05.2025 -18.05.2025	
7	Оформлення пояснювальної записки	18.05.2025 -01.06.2025	
8	Перевірка виконаного проекту керівником,	01.06.2025 -12.06.2025	
9	Захист проекту	12.06.2025 -27.06.2025	

Дата видачі завдання 05.05.2025_______________________________



Здобувач ______________________________
(підпис)



Керівник роботи (проекту) ________________ проф. Чумаченко С.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Записка пояснювальна: 54 сторінки, 22 рисунки, 8 джерел за переліком посилань.

ТЕСТУВАННЯ, ДІАГНОСТИКА, АВТОМАТИЗАЦІЯ, JENKINS, АВТОТЕСТ, XPATH, SELENIUM

Метою даної кваліфікаційної роботи є розробка діагностичної інфраструктури, яка розширить можливості автоматизації тестування програмних продуктів. Для впровадження даної логіки використано популярні та широко-використовувані технології автоматизації, такі як мова програмування Java, фреймворк для тестування Selenium, інструмент безперервної інтеграції Jenkins.

Набір даних методів дозволить створити гнучку систему, що може бути використана для автоматизації тестування різних веб-застосунків. Selenium дозволяє втілити механізацію ручного тестування, мова програмування Java впроваджує постановку різноманітних ідей для розгортання середовища роботи, Jenkins працює над автоматизацією запуска та відлагодження створених програмних рішень.

Результатом роботи в рамках кваліфікаційної роботи є гнучка діагностична структура, здатна виконувати автоматизацію тестування різноманітних програмних продуктів.

ABSTRACT

Explanatory note: 54 pages, 22 figures, 8 sources according to the list of links.

TESTING, DIAGNOSTIC, AUTOMATION, JENKINS, AUTOTEST, XPATH, SELENIUM

The goal of the qualification work is to develop a diagnostic infrastructure that provides possibility for automation testing of software products. For implementation such logic was used popular and wide-used automation technologies like Java programming language, testing framework Selenium, tool of continuous integration Jenkins.

A set of methods allow to create flexible system that can be used for automation testing different web-applications. Selenium allows to implement mechanization of manual testing, Java programming language implements setting of different ideas for deploying work environment, Jenkins works on automating the launch and debugging of created software solutions.

The result of work in this qualification work is flexible diagnostic infrastructure, that can execute automation testing of different software products.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ……………………….……………………………..……………………..…8
ВСТУП……………………………………………………………….……..……..9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТЕСТУВАННЯ ПРОГРАМНИХ ПРОДУКТІВ……..…………..………..11
1.1 Аналіз галузі тестування програмного забезпечення…………...……11
1.2 Розгляд понять SDLC та STLC…………….……….…………….......12
1.3 Аналіз різних технологій тестування ……….………..……………...14
1.4 Роль автоматизованих тестів на реальних проектах……………..17
1.5 Аналіз регресійного тестування та підходів до його виконання….. 19
1.6 Мета розроблюваної системи та постановка задачі………………..22
2 РОЗРОБКА ДІАГНОСТИЧНОЇ ІНФРАСТРУКТУРИ ДЛЯ ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ………………………...……………...…………...…25
2.1 Вибір мови програмування та IDE……………..……………...……25
2.2 Створення Maven проекту та під’єднання до Git……..…………….27
2.3 Впровадження ООП для структури класів ……...…………………30
2.4 Особливості ініціалізації Chrome Web-driver……...…………………32
2.5 Інтеграція проекту із Jenkins та Zephyr Scale……...…………………34
3 ЗАСТОСУВАННЯ РОЗРОБЛЕНОЇ ІНФРАСТРУКТУРИ У РЕАЛЬНОМУ ПРОЕКТІ…………………………………………...……………………39
3.1 Робота із dev-tools браузера….……………...…………………...…39
3.2 Створення веб-елементів та їх ідентифікація………………………41
3.3 Написання локаторів та функцій завдяки xpath…………………...…..43
3.4 Особливості взаємодії із веб-елементами в Selenium ……………...46
3.5 Написання автоматизованих тестів……………..…………………..48
3.6 Запуск створених тестів в Jenkins…………………………………..51
ВИСНОВКИ……………………………………………………………………..53

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ……………………………………..………..54
ДОДАТОК А Графічна частина проекту…………..…………………………..55
ДОДАТОК Б Текст програми автоматизованих тестів…..……………….……..63
ДОДАТОК В Текст програми методів та веб-елементів……………...………66

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення;

IT – інформаційні технології;

QC – Quality Control (контроль якості);

QA – Quality Assurance (забезпечення якості);

UI – User Interface (інтерфейс користувача);

SDLC – Software Development Life Cycle (життєвий цикл розробки програмного забезпечення);

STLC – Software Testing Life Cycle (життєвий цикл тестування програмного забезпечення);

ОС – операційна система;

CI/CD - Continuous Integration and Continuous Delivery (безперервна інтеграція та безперервна розробка);

ПК – персональний комп’ютер;

HTML – HyperText Markup Language (мова розмітки гіпертексту);

ВСТУП

Програмне забезпечення (ПЗ) оточує сьогоденну людину абсолютно всюди: на роботі, в медицині, в науці та навіть в особистому житті. Від коректної роботи програмних продуктів залежить багато чого, в тому числі і життя людини. Саме тому варто приділяти доцільну увагу якісному тестуванню рішень ще до їх комерційного використання на реальних проектах.

Тестування ПЗ – це невід’ємна частини розробки та імплементації програмних продуктів у життя людини. Саме тестування допомагає на ранніх стадіях виявити дефекти у роботі застосунку, локалізувати всі збої та перевірити витримку новостворюваного продукту до великих навантажень.

Імплементація якісного тестування – це запорука успіху роботи ПЗ, оскільки при повноцінному тестовому покритті застосунку він не буде втрачати даних, буде стійким до хакерських атак та буде працювати відповідно до раніше поставлених вимог.

Ручна перевірка логіки всього додатку може бути складною через великий обсяг ресурсів, необхідних для її виконання. Саме тому в більшості комерційних проектів може постати питання щодо автоматизації даного процесу, оскільки впровадження такого підходу зменшує час та вартість тестування ПЗ, розвантажує команду тестувальників та допомагає уникненню інших зовнішніх факторів, таких як звиклість сприйняття інженера, неуважність, втома і подібного.

Розроблювана в рамках кваліфікаційної роботи діагностична інфраструктура – це гнучка система, що дозволяє впровадити автоматизацію тестування різноманітних веб-додатків та рішень. Даний проект передбачає повний цикл тестування ПЗ автоматизованим шляхом: від отримання інфраструктури інженером до кінцевого аналізу звітів та результатів пробігу автотестів за допомоги інструменту Zephyr Scale.

Актуальність даної інфраструктури полягає в її гнучкій імплементації до різних проектів: після отримання доступу до системи інженерам достатньо вставити URL-посилання на їх продукт до спеціального файлу та запустити інфраструктуру для початку роботи із нею. Також для створення такої системи використано класичні та функціональні інструменти, що гарантують надійне та стабільне тестування всіх модулів системи.

Більше того, дана інфраструктура спроектована так, що для команди є можливість одночасної роботи над забезпеченням автоматизації тестування програмного продукту. Проект під'єднаний до системи контролю версій Git, що уможливлює стабільну роботу всієї команди з даною системою та її майбутню інтеграцію до інших сервісів та інструментів.

Тож, результатом даної кваліфікаційної роботи є гнучка діагностична інфраструктура, що уможливлює автоматизацію тестування різноманітних веб-додатків, рішень та продуктів галузі IT.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТЕСТУВАННЯ ПРОГРАМНИХ ПРОДУКТІВ

1.1 Аналіз галузі тестування програмного забезпечення

Тестування – це великий та необхідний процес, що забезпечує якість розроблюваного ПЗ. Саме тестування допомагає виявити більшість проблем та збоїв, що заважають коректній роботі застосунку. Також тестування відповідає за валідацію нового програмного продукту, перевірку відповідності його роботи до поставлених вимог та за проведення постійної регресивної діагностики створюваної системи.

За тестування додатку в команді може відповідати три спеціаліста: тестувальник, Quality Control (QC) інженер та Quality Assurance (QA) інженер. Тестувальник займається лише безпосередньою перевіркою додатку вже після етапу розробки. QC-інженер, крім тестування, займається контролем якості та валідацією всього ПЗ. QA-інженер відповідає за планування тестування, створення тестових випадків, написання документації та долучається до тестування ще на етапі аналізу поставлених вимог. Кожен із цих напрямків належить один одному, адже QC-інженер – це також і тестувальник, в свою чергу QA-інженер – це і QC-інженер, і тестувальник в одному обличчі.

Інколи помилково вважають, що тестуванню підлягає лише UI-частина додатку. Проте сьогоденні реалії вимагають від спеціаліста із тестування ПЗ знання багатьох інструментів для роботи, таких як SQL, Postman, Swagger, REST API, подібних. Кожен із цих інструментів необхідний для імплементації тестування окремих частин розроблюваного продукту.

Проведення діагностики окремих модулів ПЗ є дуже важливим, оскільки перевірка окремих складових допомагає виявити критичні вразливості додатку ще на етапі розробки. Наприклад, при впровадженні

тестування порталу для страхування, перевірка лише серверної частини допоможе виявити вразливість системи до невалідних даних, таких як завеликі грошові внески, недійсні банківські рахунки, відсутність контактної інформації застрахованої особи, подібного. Інколи трапляється так, що валідація даних відбувається лише на UI, і при потраплянні невалідного запиту до серверу, сервер може перестати працювати. У контексті даного прикладу зупинка роботи продукту може нести за собою великі фінансові втрати сторін: надавача страхових послуг та людини, що бажає застрахуватись. Виявлення таких проблем ще на етапі тестування дозволяє виявити ці збої ще на тестовому середовищі.

Також важливою галуззю є перевірка роботи додатку при великому навантаженні (performance testing). Таке тестування допомагає передбачити можливі проблеми системи при різкому збільшенні кількості користувачів та, відповідно, запитів до продукту.

Окрім звичайного ПЗ тестуванню також підлягають і мобільні додатки. Завжди важливо перевірити коректну роботу застосунку при різних операційних системах, різних моделях телефону, при різному об’єму пам’яті, розміру екрану і подібного.

Тож, галузь тестування не закінчується лише на звичайному ПЗ. Все, що знаходиться поруч з життям людини, має покриватись максимально вичерпними тестами: від звичайних телефонів та годинників до медичного обладнання, машин, літаків та іншого технічного обладнання.

1.2 Розгляд понять SDLC та STLC

Розробка та тестування ПЗ в світі IT не відбувається хаотично: для кожного із цих процесів є чітко встановлені норми та правила. Саме тому існує два поняття Software Development Life Cycle (SDLC) та, як результат, STLC [1]. SDLC – це життєвий цикл розробки програмного забезпечення. Саме тут встановлюються та розмежовуються всі етапи, що необхідні для

створення актуального та багатофункціонального продукту. Кожен із цих етапів має виконуватись в певний і поставлений час: саме це може гарантувати стабільну роботу команди над створюваним додатком. Послідовність кожної фази, яка є під час розробки програмних застосунків, наведена на рисунку 1.1.

Рисунок 1.1 – Етапи SDLC

Під час планування визначається ціль майбутнього проекту, виділення основних вимог, узгодження всіх особливостей функціональності. На етапі аналізу будується архітектура майбутнього додатку, обираються інструменти для роботи та визначається підхід до створення ПЗ.

В рамках розробки та імплементації відбувається сама розробка застосунку, відгалодження коду, поєднання всіх фрагментів в єдиний додаток. Під час тестування відбувається діагностика додатку на коректність роботи, відповідність до раніше поставлених вимог. На останньому етапі підтримки супроводжується даний програмний продукт, виправляються баги, додаються нові оновлення системи, подібного.

Згідно SDLC тестування починається аж на шостому етапі, проте робота над діагностикою розпочинається ще на моменті аналізу вимог до майбутнього додатку, оскільки тестування – це не лише про перевірку фінального застосунку, це також про тестування вимог і підготовку середовища для перевірки коректності роботи продукту. Як тільки стають відомі чіткі вимоги до функціонування ПЗ, команда тестувальників береться за роботу вже відповідно до процесів STLC.

STLC – це життєвий цикл тестування ПЗ, який впорядковує процес тестування до певних стандартів та норм. Як і SDLC, STLC має певний набір покрокових етапів, що допомагають організувати діагностику системи. Детальний набір етапів наведено на рисунку 1.2.

Рисунок 1.2 - Етапи STLC

На першому етапі команда тестувальників розбирає поставлені технічні задачі, аналізує та тестує вимоги. Після цього відбувається планування майбутнього тестування: створюється тест план, в якому описуються технології, що будуть використані в роботі, прописується середовище тестування, вказуються можливі ризики. На третьому етапі розробляється основна тестова документація: списки перевірок, тестові сценараії, подібного. При необхідності, будується спеціальне середовище для тестування (створюються необхідні дані) і лише потім команда QA-інженерів долучається до виконання своїх основних обов’язків: тестує ПЗ.

На останньому етапі тестувальники проводять ретроспективу, проговорюють проблеми, які були знайдені після діагностики продукту, аналізують помилки та як можна запобігти їх появу в майбутньому.

Ці всі етапи відбуваються паралельно із SDLC. Застосування цих двох життєвих циклів допомагає уникнути значних ризиків у розробці програмного продукту, впорядкувати процес роботи команди та покращити діагностику створюваного додатку.

1.3 Аналіз різних технологій тестування

Для впровадження і виконання діагностики використовують безліч

методів, типів та підходів до тестування. Кожен із них перевіряє особливу логіку роботи програмного продукту, розглядає різні аспекти функціональності та впроваджує детальнішу валідацію застосунку.

Серед основних типів та підходів тестування виділяють функціональне, нефункціональне, мануальне та автоматизоване. Функціональне тестування перевіряє основні функції додатку та робить акцент лише на тестуванні критичної логіки ПЗ. Цей тип тестування дозволяє перевірити цілу систему загалом, не витрачаючи час на проведення додаткових перевірок.

Нефункціональне тестування перевіряє роботу застосунку при великому навантаженні, тестується безпека та продуктивність. Приділяється особлива увага для тестування сумісності даного ПЗ із іншими ОС, діагностується зручність користування цим додатком та можливість його роботи у різному середовищі.

Також до окремого типу та різновиду тестування відносять тестування навантаження та роботоздатності системи. Важливо розуміти поведінку ПЗ при різній кількості запитів та користувачів, що одночасно працюють з програмою. Особливої уваги для тестування навантаження потребують додатки, робота яких залежить від конкретної дати. Наприклад, банківські програми будуть піддаватись великим навантаженням лише у п’ятнадцятих та тридцятих числах місяця, коли більшість працюючих людей отримує свою зарплату та перевіряє баланс рахунку. Саме тоді люди різко починають активне користування ПЗ, постійно відкриваючи його. В свою чергу кожна перевірка відправляє запит до системи, що, при різкому та завеликому збільшенні, може негативно вплинути на роботу цілої системи.

Для передбачення роботи застосунку при таких умовах використовується інструмент Jmeter. Він дозволяє в реальному часі зімітувати кількість користувачів, що одночасно працюють із розглянутим застосунком. Для цього достатньо вставити URL-посилання на додаток та ввести кількість запитів, що одночасно будуть надіслані до серверної частини

програмного продукту. Таке тестування допомагає передбачити витік даних, появу збоїв та некоректність роботи ПЗ при великих навантаженнях.

Мануальне тестування передбачає ручний підхід до діагностики продукту. Під час проведення такого типу тестування спеціаліст із забезпечення якості ПЗ вручну відкриває застосунок та перевіряє відповідність його роботи до раніше поставлених вимог. Такий підхід до тестування дозволяє перевірити роботу особливо складної логіки, яку неможливо продіагностувати автоматизованим шляхом. При перевірці, наприклад, роботи бази даних зазвичай використовують саме ручний підхід, адже перевірка даної логіки не вимагає коректної роботи UI частини.

При ручному тестуванні інженер також звертає більше уваги на зручність використання додатком, що дозволяє перевірити поведінку звичайної людини із застосунком. Таке тестування називають Usability-тестуванням [2]. Його проведення розділяється на кілька етапів, таких як дослідження, оцінювання, валідація та порівняння.

Під час дослідження перевіряється, чи дозволяє інтерфейс достатньо зрозуміло та ефективно виконати основну ціль використання даного ПЗ (наприклад наскільки легко можливо оформити замовлення). Після дослідницького етапу відбувається оцінка. Тут відбувається поглиблений аналіз того, що було зроблено на першому етапі роботи.

При валідаційному етапі береться до уваги відповідність інтерфейсу додатку до існуючих норм та стандартів. Може перевірятись, чи знаходяться елементи, наприклад, корзини та завершення роботи у правому верхньому кутку. Більшість такої логіки вже передбачена і стандартизована, тому при використанні нового ПЗ користувач інтуїтивно взаємодіє із додатком відомим йому шляхом. На цьому етапі перевіряється відповідність до згаданих норм. На останньому, порівняльному, етапі беруться до уваги можливі варіанти реалізації новостворюваної логіки і обирається найкращий.

Ще одним підходом до виконання тестування є його автоматизація. Тут QA-інженери пишуть код, що імітує ручне тестування програмно. Такий

підхід значно економить ресурси та забезпечує швидше виконання діагностики всієї системи. Також автоматизоване тестування дозволяє налаштувати паралельність пробігу тестів, створює можливість для кросплатформного тестування та зменшує рівень впливу зовнішніх ризиків.

1.4 Роль автоматизованих тестів на реальних проектах

Запровадження автоматизованого тестування на проекті, де розробляється нове ПЗ, може мати безліч своїх переваг. В першу чергу це використання найкращих практик та методик CI/CD. Наявність автоматизованих тестів в проекті дозволяє налаштувати їх автоматичний запуск після внесення змін до вихідного коду програми. Наприклад, після додавання мінімальних покращень сервіси, такі як Jenkins, можуть автоматично запустити пробіг тестів, пов’язаних із модулем в якому вносились зміни. Налагодження такого процесу дозволяє зекономити час на тестуванні продукту, відкриває можливості для миттєвого виявлення дефектів та допомагає команді тестувальників покращити діагностику розглянутого під час тестування додатку.

Також автоматизовані тести можуть виконувати роль документації. В автоматизованих тестах може зберігатись інформація про ймовірні шляхи використання даним продуктом, які інколи можуть бути занадто специфічними. Таким чином при виконанні тесту перевіряється критична логіка та зберігається покрокова інструкція того, як відтворити цей сценарій. Така документація також може бути корисною для інших учасників команди, які не працювали із даною логікою раніше. Для отримання загальних та базових знань про роботу саме цієї частини додатку достатньо просто запустити тест та ознайомитись із новим модулем самостійно.

Більше того, автоматизовані тести відкривають більше можливостей для розробки продукту. При наявності роботи автотестів, розробники із більшим рівнем впевненості можуть віддавати зміни в коді, які потенційно

мають викликати велику появу дефектів. Працюючі автотести одразу відловлять нові помилки та створять звіти із описаними проблемами.

На довготривалих та Legacy-проектах автоматизовані тести грають особливу роль. Legacy-проект – це застарілий проект, в якому відсутня будь-яка документація, де код реалізований завдяки застарілим інструментам. В таких проектах надзвичайно складно вносити будь-які зміни, адже невідомо, в якому саме місці зроблені покращення можуть спричинити появу критичних багів та дефектів. Наявність автоматизованих тестів на таких проектах допоможе в рази швидше проводити регресійне тестування всієї системи, дозволивши локалізацію проблем одразу.

Автоматизовані тести бувають різних видів. Найпопулярніші та найрозповсюдженіші – це UI тести. Це тести, що діагностують саме інтерфейс програми автоматизованим шляхом. Перевіряється відображення всіх елементів, правопис текстів, натискання кнопок і подібного. Завдяки UI тестам можна перевірити основний функціонал роботи застосунку, критичні шляхи його використання та застосування.

Unit-тести – це автоматизовані тести, що пишуться розробниками. Зазвичай вони перевіряють функціонал окремого модулю, ізольовано від всієї системи. Є корисними, адже вони дозволяють перевірити роботу лише певної частини, в якій вносились зміни та вдосконалення. При наявності проблем розробник одразу розуміє, що доданий ним новий код спричиняє появу дефектів у роботі окремого модулю.

Також автоматизації підлягають REST API тести. Наявність автоматизованих REST API тестів в проекті допомагає провести діагностику лише серверної частини додатку. Великі проекти можуть бути розподілені на окремі веб-сервіси, що згодом викликаються під час створення інших додатків та систем. Такі тести є важливими, оскільки вони дозволяють перевірити логіку роботи окремого сервісу, що не залежна від UI та що може бути викликана для іншого front-end. Також більш детально перевіряють безпеку додатку, включаючи діагностику отриманих відповідей від сервера.

Серед автоматизованих тестів також є Performance-тести, які дозволяють автоматизувати перевірку роботи ПЗ при великому навантаженні. Як згадано раніше, тестування логіки програмного продукту при критичних умовах є важливим, адже дозволяє запобігти втраті та витоку даних, зупинці роботи системи та відобразити наявні проблеми у безпеці.

Ці та багато інших типів та підходів до автоматизації тестування є дуже важливими, адже дозволяють швидше і дешевше проводити регресійне тестування всієї розроблюваної системи. Впровадження регресійного тестування є важливим і критичним процесом, що дозволяє на ранніх етапах виявити вразливість ПЗ до стресових ситуацій.

1.5 Аналіз регресійного тестування та підходів до його виконання

Регресійне тестування – це тип тестування, що проводиться повторно після внесення навіть мінімальних змін до вихідного коду ПЗ. Метою регресійного тестування є перевірка відсутності дефектів та збоїв після впровадження покращень у системі: оновлення фреймворків та бібліотек, виправлення багів, додавання нового функціоналу.

Впровадження та виконання регресійного тестування має безліч своїх переваг. Однією із них є швидке та раннє виявлення багів. Постійне виконання тестування гарантує підвищення якості продукту, зменшуючи ризик пропуску критичних помилок. Також систематична регресія запроваджує зниження вартості тестування в майбутньому та забезпечує більшу впевненість у якості продукту команди тестувальників.

Регресійне тестування програмних продуктів може бути виконане у два способи: ручним та автоматизованим. Ручний підхід може бути позитивним у випадку регресійного тестування складної логіки, такої як тестування баз даних або серверної частини додатку. Автоматизація тестування цих модулів може бути складною та зайняти великий обсяг часу для його виконання, що негативно складається на процесі в цілому.

В свою чергу впровадження ручного регресійного тестування може призвести до появи ряду негативних проблем. Перш за все це вплив людського фактору, адже при великому об’єму даних для тестування QC-інженери можуть піддатись втомі, що негативно позначиться на якість програмного продукту в цілому. При великій втомі тестувальник може просто не помітити критичний баг, що може зупинити роботу всієї системи. Також ручне тестування вимагає значних ресурсів, таких як час. Для впровадження його ручним способом необхідне залучення великої кількості спеціалістів, які будуть змушені проводити лише регресивне тестування, без залучення їх до виконання своїх інших прямих та робочих обов’язків.

В свою чергу автоматизований підхід до регресійного тестування має значно більшу кількість переваг в своєму використанні. Основна із них це унеможливлення впливу людських факторів на даний процес. При автоматизації тестування немає впливу втоми, звиклості сприйняття, та неуважності. Автоматизовані тести унеможливлюють зниження якості тестування програмного продукту через указані фактори.

Автоматизація регресійного тестування дозволяє знизити час для його виконання. Перш за все можливо налаштувати щоденний запланований запуск тестів вночі, тим самим на ранок вже мати готові результати проведеного регресійного тестування. Також можливо розділити пробіг тестів паралельно. Аби автоматизовані тести не проходили один за одним крок за кроком, є можливість налаштувати пробіг тестових сценаріїв паралельно в декількох браузерах. Це знизить загальний час виконання збірки та дозволить в рази пришвидшити час регресійного тестування ПЗ.

Більше того, автоматизація регресії відкриває більші можливості для генерації та формування отриманих результатів у звітах. Інструменти, такі які Jenkins, дозволяють генерувати змістовні звіти після завершення виконання автоматизованого регресійного тестування [3]. Можливо налаштувати скріншот стану програмного продукту при зупинці виконання тесту, можливо додати відеозапис пробігу тесту, запис місця номеру рядку,

де саме зупинився тест, подібного. Ці та інші інструменти дозволяють широкий аналіз отриманих результатів після виконання збірки.

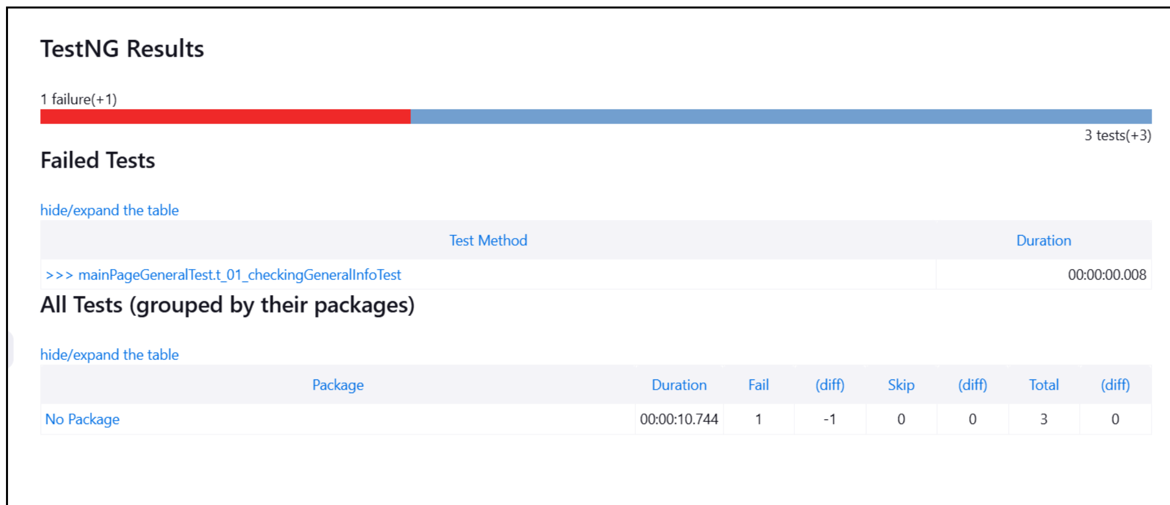


Рисунок 1.3 – Аналіз пробігу тестів в Jenkins

Окрім цього Jenkins дозволяє інтеграцію із іншими системами та ресурсами. Тож цілком можливо налаштувати автоматичну генерацію звітів після виконання повного регресійного тестування ПЗ в необхідній системі, по типу Jira та інших. Така можливість полегшить роботу всіх членів команди, адже вся необхідна інформація буде зібрана в одному місці. Тим самим вся проектна документація та звітності про дефекти будуть зібрані та збережені у єдиній системі відстежування.

Тож, впровадження якісного автоматизованого регресійного тестування має безліч своїх плюсів та переваг, які неодмінно показують себе на комерційних та реальних IT-проектах. Впровадження такої системи допомагає уникнути великої кількості ризиків та надзвичайних ситуацій, що можуть спричинити значні неприємності для всіх сторін, що беруть участь в розробці, тестуванні, підтримці, та як в результаті, використанні розроблюваного багатофункціонального програмного продукту.

1.6 Мета розроблюваної системи та постановка задачі

Використання надійного ПЗ є запорукою успіху багатьох галузей та індустрій сьогодення: від звичайних соціальних мереж до банківських систем, страхових компаній та державних установ. Невід’ємною частиною розробки таких програмних продуктів є тестування, що дозволяє виявити більшість дефектів ще на етапі проектування різноманітних рішень.

Як було сказано раніше, для діагностики подібних продуктів використовують два підходи: ручне та автоматизоване тестування. Ручний підхід вимагає залучення значного обсягу ресурсів для його виконання. Більше того, такий підхід має ряд недоліків у своїй роботі, один із них – це об’єм часу, що потрібний для його повного впровадження, інший – це людський фактор, що може завадити вчасному виявленню критичних дефектів системи.

Тож, метою даної кваліфікаційної роботи є створення гнучкої інфраструктури, що буде відповідати за виконання тестування програмних продуктів автоматизованим шляхом. Автоматизований підхід до тестування дозволить скоротити обсяг часу для впровадження повного регресійного тестування системи, зменшить ризик пропуску значних відхилень у роботі застосунку та запровадить гнучку роботу команди над тестуванням розроблюваного продукту.

Така система дозволятиме інженерам із забезпечення якості ПЗ вчасно та швидко проводити регресійне тестування всієї системи, використовуючи продуктивні та багатофункціональні інструменти для роботи. Концепція створюваної інфраструктури зможе розширити плацдарм роботи тестувальників, зменшивши об’єм необхідних ресурсів для виконання поставлених задач та забезпечити стабільне тестування всіх функцій додатку.

Більше того, використання такої системи в команді дозволить IT-компаніям та проектам скоротити вартість виправлення дефектів, знаходячи їх ще на етапі розробки створюваного ПЗ. Застосування даної

інфраструктури у проведенні регресійного тестування також дозволить зменшити вплив людського фактору на перевірку логіки роботи застосунку, оскільки автоматизований підхід до тестування може виконувати великий обсяг роботи безперервно та без впливу зовнішніх людських факторів, таких як втома, неуважність, звиклість сприйняття, подібного.

Тож, кінцева ідея інфраструктури полягає в розробці гнучкого середовища, що матиме повний набір інструментів для впровадження повного тестування програмного додатку. Використання такого середовища можливе в багатьох галузях, основні із них це IT-компанії та проекти, що спеціалізуються на розробці гнучких веб-рішень та сайтів.

Особливим попитом на таку інфраструктуру можуть користуватись онлайн-магазини, портали для надавання адміністративних послуг, програмні системи управління, банківські сайти та навіть застосунки для керування страховими процесами. Додатки, функціонал яких напряму взаємодіє із кінцевими користувачами та які мають в своїй роботі фінансовий контекст, потребують більш детального тестування та перевірки, оскільки від правильності їх роботи залежать значні ресурси, такі як гроші, товари та інколи навіть людське життя.

Використання цієї інфраструктури в таких типах проектів дозволить запобігти виникненню значних ризиків, пов’язаних через некоректність роботи ПЗ. Тому розроблювальна система стане корисною та продуктивною в таких галузях, оскільки її використання допоможе значно раніше виявляти появу критичних дефектів та багів у роботі додатку.

Для розробки та налагодження такої інфраструктури необхідно виконати ряд поставлених задач:

– провести аналіз схожих існуючих систем, їх інструментів та зони використання;

– обрати мову програмування та інтегроване середовище розробки, інтегрувати необхідні фреймворки та бібліотеки до проекту;

– проаналізувати існуючі інструменти для управління тестовими сценаріями;

– розгорнути Jenkins на локальній машині чи сервері, провести інтеграцію проекту із системою контролю версій до Jenkins;

– створити та налагодити автоматизовані тести в спроектованій діагностичній інфраструктурі для реального проекту;

Для технічного використання створеної системи необхідно виконати наступні кроки:

– розглянути принципи роботи із dev-tools браузера;

– створити унікальні локатори та функції для обраного додатку;

– проаналізувати можливості роботи із веб-елементами в Selenium;

– створити та налагодити автоматизовані тести в спроектованій діагностичній інфраструктурі для реального проекту;

– проаналізувати результати запуску доданих тестів завдяки інструменту Jenkins;

Виконання описаних кроків допоможе створити гнучку діагностичну інфраструктуру, що дозволятиме виконувати тестування програмних продуктів автоматизованим шляхом. Використання створеної системи допоможе інженерам із забезпечення якості ПЗ полегшити процес тестування, зменшити об’єм ресурсів для його виконання та зменшить ймовірність опущення критичних дефектів у роботі застосунку.

2 РОЗРОБКА ДІАГНОСТИЧНОЇ ІНФРАСТРУКТУРИ ДЛЯ ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ

2.1 Вибір мови програмування та IDE

Для розробки та створення гнучкої діагностичної інфраструктури можуть бути використані безліч мов програмування. Серед них Python, JavaScript, TypeScript, Ruby та багато інших. Кожна із них має власні переваги та недоліки у використанні, проте лише декілька з них можуть бути застосованими для реальних проектів індустрії IT.

Python є простою мовою програмування з простим синтаксисом. Має підтримку великої кількості бібліотек та фреймворків, може бути використана на різних платформах, операційних системах, подібного. Проте на масштабних проектах використання Python супроводжується повільнішим часом компіляції (на від міну від інших мов), поганою читабельністю та витратою великої кількості пам’яті для роботи.

Мова програмування JavaScript відзначається хорошою взаємодією із браузером, оскільки має ту саму мову, що і front-end сторінки, вирізнається активною підтримкою нових фреймворків, таких як Cypress та Plawright, має активну спільноту, яка розвиває використання даної технології як основної в автоматизації. Однак JavaScript погано підтримує автоматизацію REST API тестів, має складніший синтаксис, не є універсальною для автоматизації більшості тестових сценаріїв.

Ruby гармонійно співпрацює із такими фреймворками як Cucumber та RSpec, що розширює плацдарм для створення автоматизованих тестів. Має зрозумілий синтаксис, що дозволяє почати роботу з нею навіть із низьким рівнем знань. Проте не є сильно поширеною у тестуванні, має менші можливості для автоматизації тестування REST API, має великий час компіляції та запуску створеного коду.

Однією із найпопулярніших мов програмування, що використовується в автоматизації тестування, є Java. Це об’єктно орієнтована мова програмування, що дозволяє побудувати гнучку та чітку структуру класів, має широку підтримку різноманітних фреймворків, може бути використаною у автоматизації тестування різних частин додатку: UI, REST API, мобільних додатків, подібного. Також підтримує простішу інтеграцію із різними системами по типу Jenkins, оскільки такі рішення в більшості випадків також реалізовані на мові програмування Java.

Тож, для реалізації даної діагностичної інфраструктури було обрано саме мову Java, оскільки вона є класичним рішенням для автоматизації тестування, легко взаємодіє із різними фреймворками та дозволяє паралельний запуск тестів, що пришвидшує загальний час їх виконання.

Як середовище розробки із мовою програмування Java може буде обрано багато різних IDE, таких як Visual Studio Code, Eclipse та IntelliJ IDEA. Visual Studio Code є легшим у використанні, підтримує безліч мов програмування, проста інтеграція різноманітних плагінів. Проте даний додаток не є повноцінним середовищем розробки, має незручний інтерфейс використання та має складне налаштування проекту.

Eclipse є класикою для розробки різних додатків, використовуючи Java. Є безкоштовною для використання в навчальних та комерційних цілях, підтримує використання різних систем контролю версій. Однак є доволі застарілим, адже на ринку працює більше двадцяти чотирьох років. Тому інтерфейс є доволі одноманітним та менш інтуїтивним у використанні.

І найкращим рішенням для впровадження в рамках кваліфікаційної роботи діагностичної інфраструктури є додаток IntelliJ IDEA. Дане середовище є найпродуктивнішим для розроблення систем та цифрових рішень на Java, адже має широку підтримку JDK та Kotlin, безперешкодно інтегрується із різними системи та має швидшу компіляцію створених автоматизованих тестів. Також доволі легко впроваджується із Git та чудово підходить для великих і нестандартних проектів.

Тому для реалізації діагностичної інфраструктури буде використано саме IntelliJ IDEA, адже вона відповідає поставленим вимогам для коректної роботи, є стандартизованою у галузі автоматизації тестування програмних продуктів та чудово підходить для вирішення щоденних задач із підтримки та додавання нових автотестів у системі.

2.2 Створення Maven проекту та під’єднання до Git

Для створення діагностичної інфраструктури було обрано Maven проект. Даний тип проектів дозволяє автоматично завантажити всі необхідні файли та фреймворки до створюваного проекту. Також в такого типу проектах самостійно вибудовується структура з класами та файлами відповідно до загальноприйнятих стандартів та правил. Це полегшує написання коду та майбутню інтеграцію системи до сторонніх сервісів.

Аби створити новий Maven проект, завантажуємо Java JDK та переходимо в IntelliJ IDEA. Дане середовище розробки має ряд своїх шаблонів та типів структур проекту, що можуть бути використані для автоматизації тестування. Для створення діагностичної інфраструктури ідеально підходить проект типу quick start (рисунок 2.1).

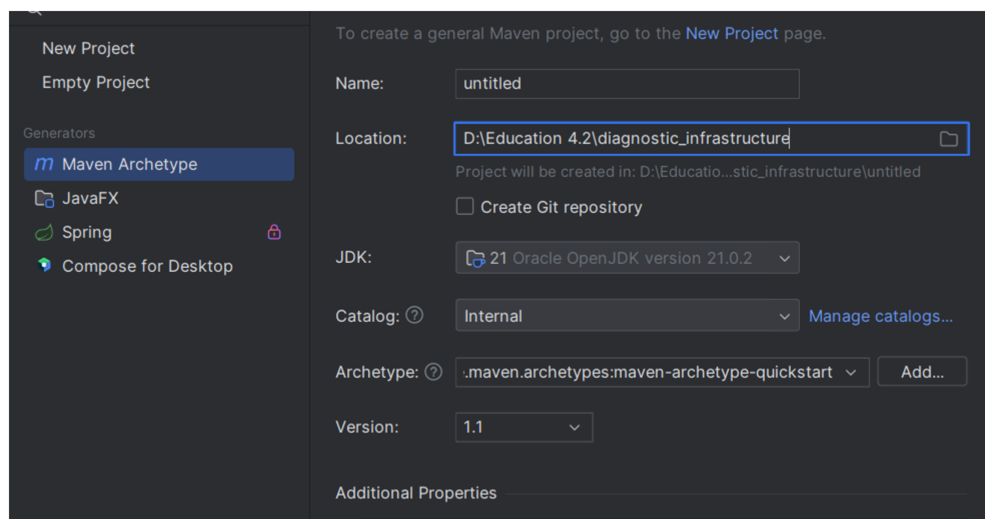


Рисунок 2.1 – Створення нового проекту

Після додавання нового проекту IntelliJ IDEA почне автоматичний запуск збірки проекту: створюються необхідні папки, файли та класи. Основним та найважливішим із них на даний момент є файл pom.xml. В даному файлі прописуються всі залежності, що необхідні для додавання потрібних фреймворків та бібліотек. Після додавання залежностей Maven проект автоматично завантажує необхідні дані для коректної роботи діагностичної інфраструктури. Приклад додавання залежностей, що необхідні для створення даного проекту, наведені в лістингу 2.1.

Лістинг 2.1 – Додавання залежностей в pom.xml

```
<dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>7.10.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>4.18.1</version>
    </dependency>
```

Наступним кроком розробки діагностичної інфраструктури є її під’єднання до системи контролю версій. Система контролю версій – це інструмент, що дозволяє одночасну роботу кількох спеціалістів над вихідним кодом програми. Також дана система зберігає історію змін в додатку, тим самим розширює можливість управлінням вихідною версією застосунку.

У разі внесення змін, що призводять до зупинки програми, завдяки системи контролю версій легко можливо повернути зміни та відновити роботоздатність продукту, яка була до внесених покращень.

Також системи контролю версій уможливлюють інтеграцію розроблюваної інфраструктури до різних систем, наприклад Jenkins. Це дозволить налаштувати кращі процеси CI/CD та автоматичний запуск створюваних програмних тестів.

Загалом є дві найпоширеніші системи контролю версій: GIT та SVN. SCN - це система, що була розроблена у двохтисячному році. На даний момент SVN вже є застарілим рішенням, оскільки протягом двох десятиліть галузь IT реалізувала більш гнучкі та сучасніші рішення. SVN має гіршу підтримку роботи у команді, є менш стабільним та уможливлює виникнення конфліктів між версіями файлів та одночасним внесенням змін до того самого класу кількома спеціалістами в той же самий проміжок часу.

Найпродуктивнішим рішенням є використання GIT. GIT підтримує легшу паралельну роботу завдяки гілкам репозиторію, є набагато швидшим у роботі та більш часто використовується в різних інтеграціях, на відміну від SVN. Проте освоєння GIT є складнішим і вимагає більшого часу для вивчення основних команд для роботи із репозиторієм.

Для додавання системи контролю версій було використано GitHub – онлайн-сервіс для розміщення та управління репозиторіями GIT [4]. Для створення нового репозиторію авторизуємось в GitHub, переходимо на вкладку репозиторії – новий – створити. Після створення система перенаправить нас на нову сторінку (рисунок 2.2).

...or create a new repository on the command line

```
echo "# diagnostic_infrastructure" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/AndriiOmelnytskyi/diagnostic_infrastructure.git
git push -u origin main
```

Рисунок 2.2 – Команди для інтеграції проекту до GIT

На цій сторінці знаходяться команди, що варто виконати в терміналі створеного проекту для його додавання до цього репозиторію. Після успішного виконання всіх команд, проект отримає підтримку GIT і можливість роботи із створеним репозиторієм прямо із IntelliJ IDEA

2.3 Впровадження ООП для структури класів

Важливим аспектом розробки діагностичної інфраструктури для автоматизації тестування програмних продуктів є створення певної структури класів, що забезпечить кращий та зручніший підхід до написання тестів та впровадження діагностики всієї системи в цілому.

Після створення Maven проекту середовище розробки IntelliJ IDEA автоматично створює розподілення класів за двома папками: main та test. Даний підхід є стандартизованим у галузі тестування, адже він дозволяє створювати чистіший та читабельний код, має кращі здібності до інтеграції з сторонніми сервісами та має чітке розмежування між тестами та методами, що потрібні для впровадження автоматизації в системі.

До теки tests додаються всі класи, в яких зберігаються автоматизовані тести системи. Назва кожного класу створюється відповідно до логіки та модулю, яка перевіряється автотестами в ньому. У разі не проходження тесту це дає одразу розуміння того, з яким саме модулем чи логікою порталу виникла проблема та, можливо, з’явились відповідні баги.

Також створюється клас baseTest, від якого відбувається наслідування інших класів з тестами в системі. В цьому базовому класі відбувається ініціалізація класів із сторінками, що будуть використовуватись безпосередньо в тестах. Впровадження базового класу із тестами зменшує навантаження на звичайні класи із тестами, зменшує дублювання коду та дозволяє винести спільні методи в окреме місце.

В папці main зазвичай створюються всі класи, що необхідні для розгортання середовища для автоматизації тестування. Тут зберігаються

класи, що дозволяють інтегрувати результати пробігу всіх тестів до інших сервісів, класи, що необхідні для коректності роботи всієї системи (наприклад класи для роботи та зчитування файлів pdf, zip, ics і подібного) та класи, що зберігають в собі всі програмно описані HTML-елементи сторінки.

Для зручності роботи такі класи розмежовуються в окремих папках main. В нашому випадку створюємо папку pages та зберігаємо в ній всі класи, що необхідні для програмного опису окремих сторінок нашого веб-додатку. Підхід, в якому реалізовується створення окремого класу для окремої сторінки веб-додатку, дозволяє розподілити розміщення всіх веб-елементів у проекті в певному порядку та дозволяє краще візуалізувати створюваний код, що покращує майбутню роботу із ним.

Основним класом сторінок є basePage. BasePage – це головний клас із сторінкою, в якому відбувається оголошення всіх необхідних даних, що необхідні для коректної роботи всіх класів із сторінками. Це, наприклад, може бути ініціалізація драйверу та запуск нового вікна браузера. Даний підхід дозволяє використати вже створену логіку із ініціалізації в базовому класі безпосередньо у всіх класах із програмним описом веб-сторінок додатку. Це значно зменшить об’єм створюваного коду, адже зникає необхідність у дублюванні коду програми в кожному класі. Створена структура класів зображена на рисунку 2.3.

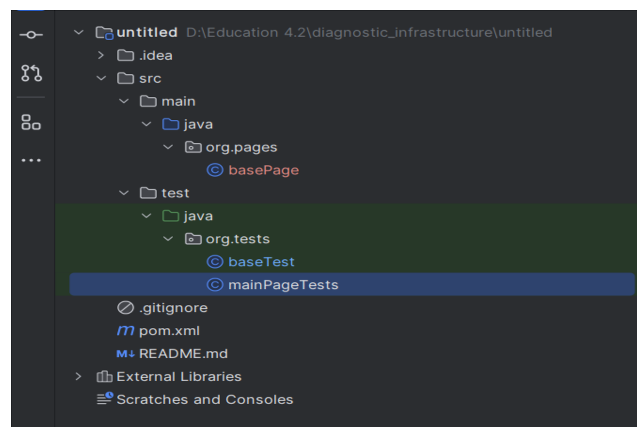


Рисунок 2.3 – Створення базової структури класів інфраструктури

Розмежування класів таким чином є одним із принципів ООП в проекті. Впровадження ООП в розробці діагностичної інфраструктури дозволяє використовувати абстракцію, яка в свою чергу дозволяє виділити об’єкт, в нашому випадку це HTML-елемент, та уможливити майбутню роботу з ним.

Також ООП має метод інкапсуляції, що дозволяє створити геттери та сеттери для управлінням створеними веб-елементами. Використання окремих геттерів спрощує написання коду та уможливлює взаємодію із веб-елементом різними методами, що наявні в бібліотеці Selenium.

Тож, впровадження ООП в проекті є позитивним ключем, що розширює можливості повторного використання коду, спрощує читабельність коду та запроваджує краще автоматизоване тестування всього розроблюваного програмного продукту.

2.4 Особливості ініціалізації Chrome web-driver

Для початку роботи із автоматизованими тестами в діагностичній інфраструктурі слід оголосити ініціалізацію Chrome web-driver. Web-driver – це інтерфейс, що дозволяє запускати та управляти будь-яким браузером програмно. Умовно кажучи, це певний міток між кодом та браузером, що дозволяє зв’язати інфраструктуру із певним браузером.

Даний інструмент є корисним, оскільки дозволяє виконувати тестування програмного продукту в певному конкретному браузері. Web-driver може знаходити будь-які елементи на сторінці, дозволяє зімітувати дії реального користувача та дозволяє виконати певні задані перевірки.

Також web-driver має можливість запускати різні браузери, такі як Chrome, Fire Fox, Edge і подібних. Дана можливість є корисною, оскільки розширює майбутній потенціал для кросбраузерного тестування. Деякі проблеми можуть з’являтись лише в певних браузерах і можливість запуску автоматизованих тестів в різних браузерах дозволяє виконати перевірку логіки роботи додатку в різних середовищах.

В рамках розроблюваної діагностичної інфраструктури було обрано Chrome web-driver, який обирає браузер Chrome як своє основне середовище роботи. Chrome web-driver належить до Selenium web-driver, що є частиною бібліотеки Selenium. Тож, для початку роботи із web-driver в pom.xml додати залежність, яка підключить Selenium до розроблюваного проекту.

Після цього перезбираємо повністю проект. Це потрібно для увімкнення всіх функцій Selenium та завантаження необхідних бібліотек для роботи із web-driver. Виконання ініціалізації драйверу в проекті виконуємо в базовій класі-сторінці. Такий підхід буде практичним, адже таким чином ми зможемо лише одного разу оголосити використання web-driver і не дублювати цей код в кожному класі із описаною програмно сторінкою.

Приклад ініціалізації web-driver, який було використано в розробці даної діагностичної інфраструктури, наведено в лістингу 2.2.

Лістинг 2.2 – Приклад ініціалізації web-driver

```
public basePage() {
    if (driver == null) {
        WebDriverManager.chromedriver().setup();
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--start-maximized");
        driver = new ChromeDriver(options);
        driver.get("https://ad.nure.ua");
    }
}
```

Рядок WebDriverManager.chromedriver().setup() дозволяє ініціалізувати драйвер, WebDriverManager.chromedriver().setup() створює нову опцію роботи з браузером, options.addArguments("--start-maximized") розкриває вікно браузера на повну, driver = new ChromeDriver(options) запускає бразуер та driver.get("https://ad.nure.ua") переходить за вказаним в параметрі посиланням на веб-додаток.

Саме тут людина, яка буде користуватись даною діагностичною інфраструктурою, зможе вказувати різні посилання на різні програмні

рішення. Використання такого підходу створює гнучкі можливості для застосування даної структури в різних проектах та рішеннях. Для тестування іншого додатку інженеру із забезпечення якості ПЗ достатньо вказати лише посилання на цей застосунок – після перезбірки проекту стане можливість виконувати автоматизоване тестування вже іншого програмного продукту.

Ініціалізація драйверу в базовій сторінці також допоможе полегшити роботу із тестами. Для того, аби в класах з автотестами можна було використовувати програмно описані HTML-елементи в класах з сторінками, слід в baseTest оголосити функцію, в якій ми будемо ініціалізовувати всі необхідні класи. Після цього оголошуємо функцію тегом @BeforeClass, що дозволить виконати цю функцію до запуску всіх тестів із класу. Потім в класі з тестами успадковуємось від baseTest. Це дозволить використати створену функцію та ініціалізувати використання сторінок в тестах.

Даний підхід спростить виконання та запуск тестів, їх написання та підтримку, оскільки всі необхідні елементи та функції вже винесені в окремі класи та немає жодної необхідності їх дублювати знову.

2.5 Інтеграція проекту із Jenkins та Zephyr Scale

Детальний аналіз отриманих результатів після пробігу автоматизованих тестів є дуже важливим етапом тестування, адже саме тут відбувається локалізація можливих багів, що були знайдені автотестами.

Аби не запускати автоматизовані тести кожного разу вручну, є ряд інструментів, що дозволяють пришвидшити і механізувати даний процес. Одним із них є Jenkins. Дана технологія дозволяє налаштувати різні механізми, що будуть автоматично виконувати процеси тестування, такі як запуск тестів, перезбірка рішення, оновлення бази даних, подібного.

Для використання Jenkins може бути використаний окремий сервер, на якому буде розгорнуто даний інструмент. В такому випадку Jenkins стає

доступним для всієї команди, що дозволяє одночасну роботу над аналізом результатів виконання автоматизованих тестів.

Однак є випадки, коли ця технологія повинна бути розгорнута локально на машині інженера. Jenkins, що працює на ПК тестувальника, може бути використаний для запуску окремих частин тестів через Jenkins. На відміну від запуску всіх тестів на окремому сервері, такий підхід дозволяє зекономити час тестування, не вимагає пробігу всіх тестів та має результати окремо вже в необхідний момент роботи.

Для інтеграції розроблюваної діагностичної інфраструктури завантажуємо та встановлюємо Jenkins на комп’ютер. Після встановлення відкриваємо localhost та переходимо до головної сторінки даного інструменту. Для початку роботи слід створити окремий Item, який буде відповідати за відлагодження та запуск тестів (рисунок 2.4).

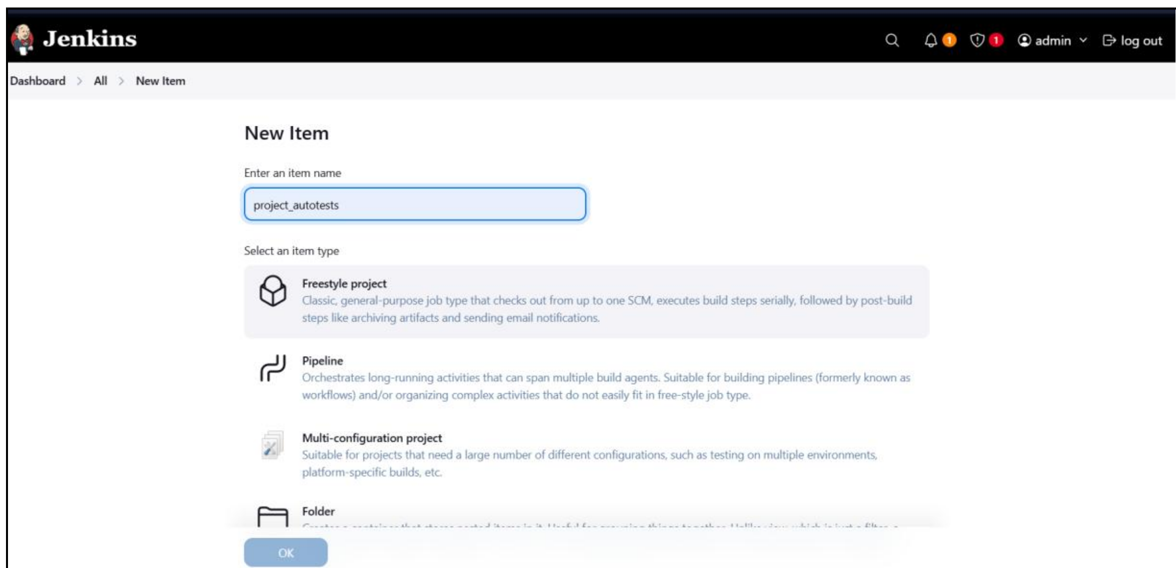


Рисунок 2.4 - Створення нового Item

Після додавання нового проекту в Jenkins налаштовуємо конфігурацію. Тут можливо додати кроки, які будуть виконуватись під час запуску збірки, вказати посилання на GitHub, додати інтеграцію із сторонніми сервісами, по типу Zephyr Scale. Приклади таких кроків наведено на рисунку 2.5.

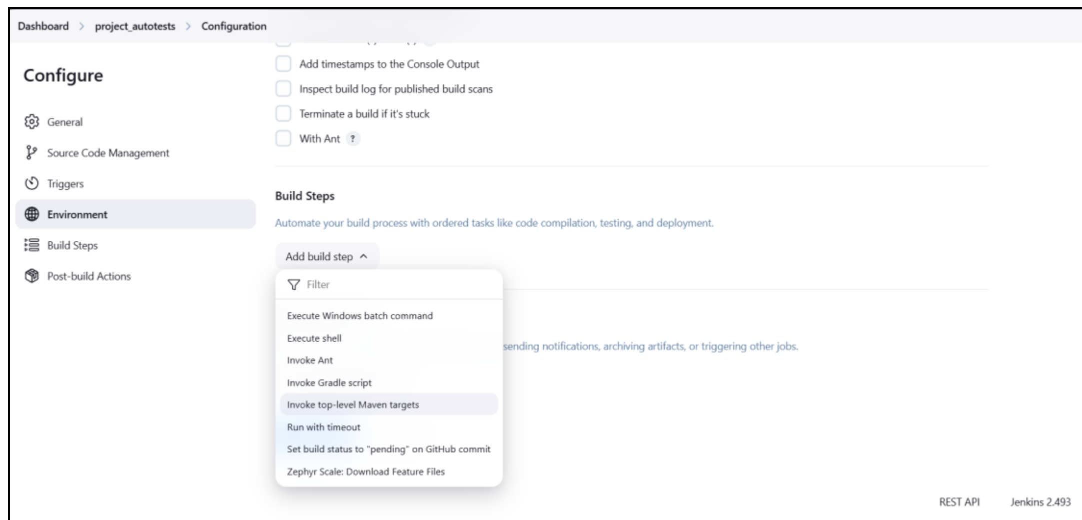


Рисунок 2.5 – Приклад додавання кроків збірки

Ці кроки необхідні для коректної збірки всього проекту. Тут вказуються, які дії мають бути виконані перед або після запуску всіх тестів в проекті. Наприклад, можливо додати крок «Invoke top-level Maven targets», що буде викликати перезбір всього Maven проекту та запускати абсолютно всі функції, що помічені анотацією test.

Для покращення результатів аналізу звітів можна використати сторонні сервіси та системи. Після пробігу всіх тестів jenkins зможе автоматично надсилати до іншого сервісу дані результатів виконня всіх автотестів системи. Такий підхід дозволить зберігати всю інформацію в єдиній системі, в якій працює команда, та дозволить автоматичну генерацію звітів у більш гнучкій та читабельній формі. Однією із таких систем є Zephyr Scale [5].

Zephyr Scale є частиною додатку Jira, що дуже часто використовується на більшості комерційних IT-проектів. Ця програма є майже стандартом у використанні для організації та контролю роботи спеціалістів. Zephyr Scale підтримує управління мануальним та автоматизованим тестуванням. Є можливість створення різноманітної тестової документації, що необхідна для виконання діагностини системи. Такоє є можливість генерації звітів з тестування, які містять в собі різноманітні графіки, інформацію про результати пробігу та багато іншого.

Більше того, Zephyr Scale підтримує інтеграцію із сторонніми сервісами, надаючи доступ до генерації власного API ключа. Після отримання даного унікального API можливо додати його до Jenkins чи проекту, тим самим встановиши зв’язок між діагностичною інфраструктурою та платформою для аналізу звітів із тестування. Приклад генерації такого ключа доступу наведено на рисунках 2.6 та 2.7.

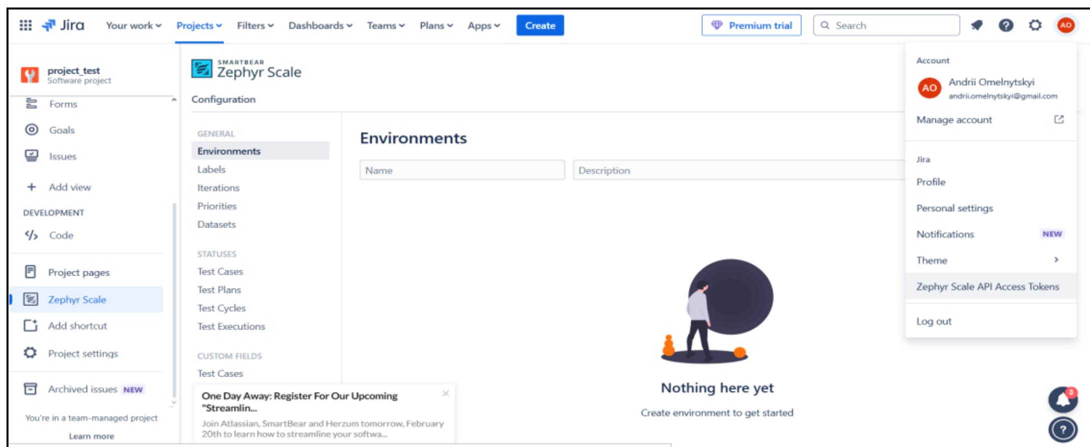
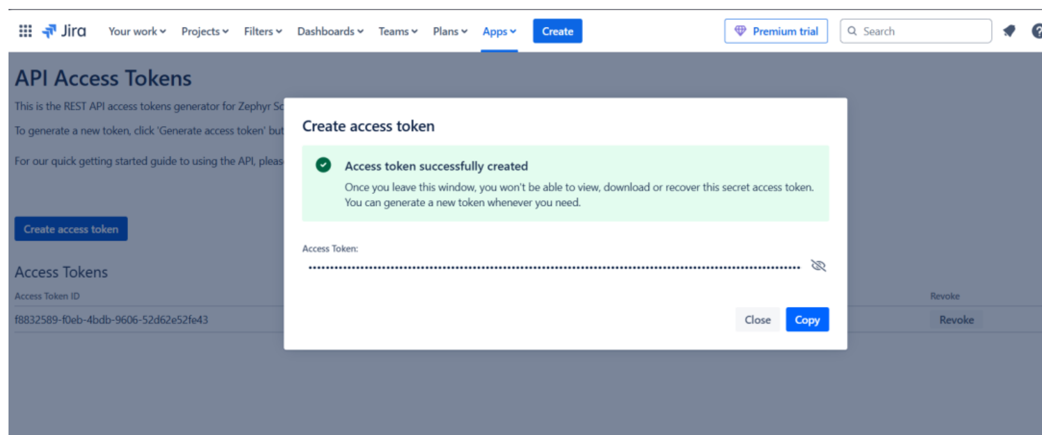


Рисунок 2.6 – Генерація ключа доступу Zephyr Scale



Рисуннок 2.7 – Згенерований ключ доступу

Для інтеграції проекту в Jenkins встановлюємо необхідний плагін Zephyr Scale та додаємо згенерований API ключ до системи. Потім в налаштуваннях Item додаємо новий крок збірки: Zephyr Scale: Publish Test Results. Цей крок опублікує звіт з пробігу тестів за вказаним ключем доступу.

Також можливо налаштувати інтеграцію із Zephyr Scale через REST сервіси. Для цього в теці main додаємо новий клас ZephyrScaleUploader. В середині класу додаємо посилання на Zephyr Scale, вказуємо ключ проекту, наш API та xml-файл, в якому зберігається результат пробігу тестів.

Такий файл можна згенерувати прямо в проекті, додавши suite. Suite дозволяє винести певні класи із тестами до окремих папок та розширює можливість запуску тестів лише за певними модулями. Бувають випадки, коли слід перевірити функціонал лише певної логіки, тому саме тут це буде корисним, адже не треба витрачати час на тестування всієї системи. Також запуск автоматизованих тестів через suite генерує звіт у форматі xml-файлу, який можливо відправити до сторонньої системи для аналізу.

Після виконання цих дій формуємо HTTP запит до системи Zephyr Scale. В запиті передаємо назву нашого xml-файлу із результатом пробігу тестів та інші необхідні дані для створення змістовного звіту в Zephyr Scale.

Поетапне виконання вищеописаних кроків дозволяє створити діагностичну інфраструктуру для автоматизації тестування програмних продуктів, яка буде гнучкою в інтеграції та використанні в інших системах та яка матиме широкі можливості для виконання автоматизованого тестування різних веб-додатків та програмних рішень.

3 ЗАСТОСУВАННЯ РОЗРОБЛЕНОЇ ІНФРАСТРУКТУРИ У РЕАЛЬНОМУ ПРОЕКТІ

3.1 Робота із dev-tools браузера

Для початку роботи із даною діагностичною інфраструктурою слід розуміти основні принципи роботи із dev-tools браузера. Dev-tools – це вбудований інструмент роботи з браузером, що допомагає працювати над розробкою та тестуванням створюваного веб-продукту. В dev-tools можна отримати інформацію про HTML-розмітку сторінки, про наявність помилок, роботу HTTP запитів. Також в dev-tools можливо відобразити сторінку в різних форматах, що значно спрощує крос-платформне тестування.

На рисунку 3.1 зображено візуалізацію даного програмного продукту у його використанні в мобільний додатках. Така дозволяє зобразити будь-яке ПЗ у різних розмірах, що в свою чергу відкриває можливість перевірки роботи додатку в різних умовах та при різній масштабованості.

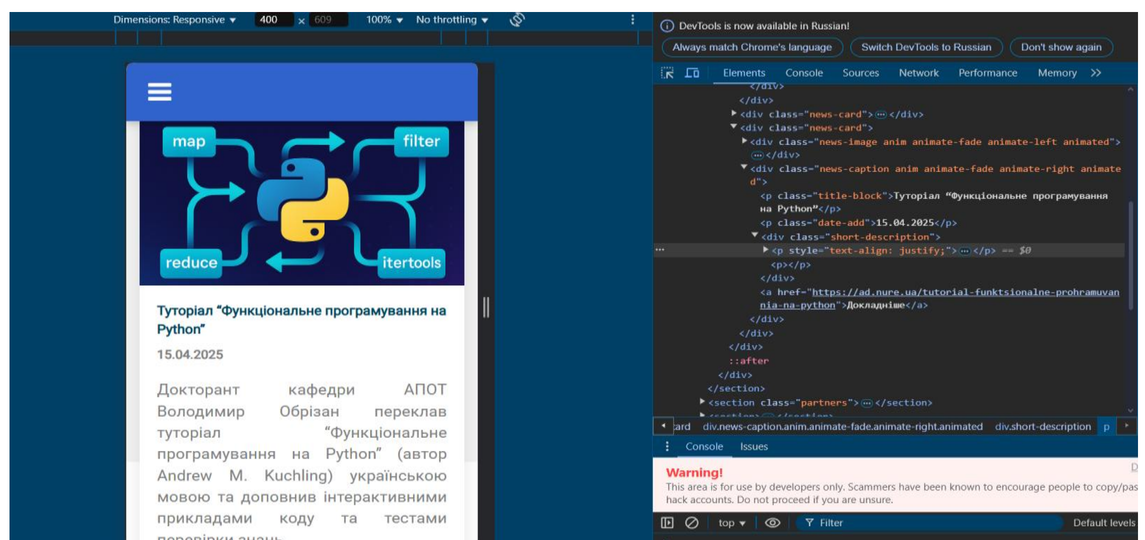


Рисунок 3.1 – Використання dev-tools для крос-платформного тестування

В рамках проведення та впровадження автоматизованого тестування завдяки розробленої діагностичної інфраструктури dev-tools браузера допоможе знайти ідентифікатори різноманітних веб-елементів на сторінці додатку. Для цього слід натиснути правою кнопкою миші, обрати «дослідити» та перейти до сторінки, яку відкриє сам браузер (рисунок 3.1)

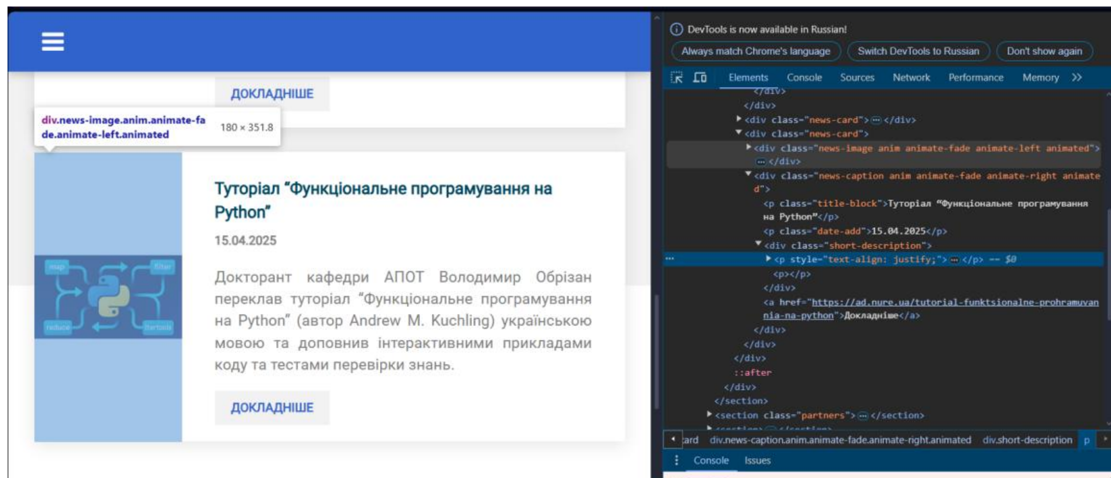


Рисунок 3.2 – Перегляд dev-tools

В боковому меню, що з’явилось після переходу до dev-tools, бачимо повну HTML-розмітку даного застосунку: всі описані front-end розробником елементи, які відображаються на цій сторінці, їх ідентифікатори та імена. Для того, аби провести автоматизацію тестування, web-driver має зрозуміти з яким саме елементом йому потрібно побудувати взаємодію. Тож, для локалізації певного HTML-елементу шукаємо ідентифікатори, що були задані розробником під час створення даної сторінки.

Розуміння основних назв та імен даних елементів на сторінці допоможе впровадити кращу та якіснішу автоматизацію тестування, розкриє можливість чіткої взаємодії із певним елементом та представить більше інформації про коректність роботи всієї системи.

Dev-tools є корисним інструментом, адже дозволяє знайти всі елементи, що мають один і той самий ідентифікатор, та створити унікальні та гнучкі функції, що будуть використані в розробці автотестів.

3.2 Створення веб-елементів та їх ідентифікація

Аби діагностична інфраструктура працювала правильно, в проекті слід програмно описати всі веб-елементи, з якими відбуватиметься взаємодія в рамках тестування. Веб-елемент – це інтерфейс Selenium, що являє собою програмний опис HTML-елементу на сторінці розглянутого ПЗ. Опис таких елементів відбувається завдяки локаторам та ідентифікаторам, що дозволяють точно прописати шлях до цього елементу на сторінці.

Опис таких веб-елементів в коді відбувається завдяки різним ідентифікаторам. Кожен із них забезпечує полегшення пошуку різних веб-елементів драйвером та дозволяє побудувати взаємодію із конкретною частиною програмного продукту. Список ідентифікаторів, що є доступними в даному проекті, наведений на рисунку 3.3

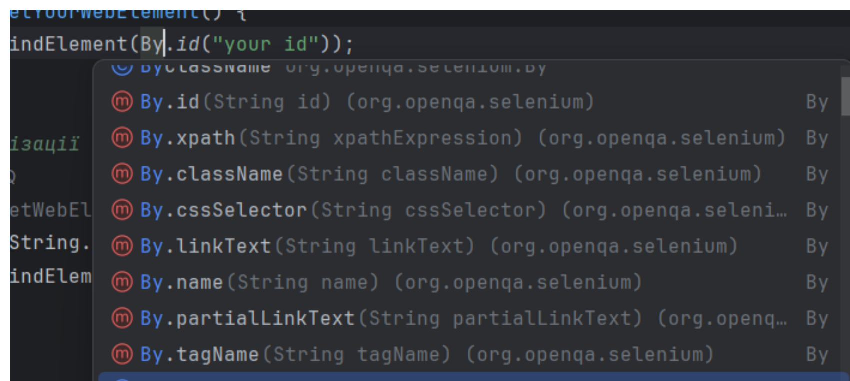


Рисунок 3.3 – Список можливих тегів

Id дозволяє ідентифікувати елемент за id елемента в HTML-розмітці, className знаходить елемент за іменем класу, яким оголошений цей веб-елемент, відповідно name знаходить елемент за його іменем в коді.

Також є метод linkText, який знаходить елемент за текстом, який знаходиться в цьому елементі. Пошук елементу саме таким чином є ризиковим та недоцільним, адже цей підхід шукає всі збіжності із цим текстом на сторінці. Умовно, слово, за яким ідентифікували елемент, зустрічається на сторінці тричі. Драйвер знайде перший елемент, який

матиме цей текст, і буде взаємодіяти лише з ним. Тож, якщо в необхідному елементі на сторінці цей текст зустрічається не вперше, то драйвер попросту не зможе з ним виконати жодних дій. Саме тому важливо чітко та правильно оголосити пошук елемент, аби драйвер зміг правильно виконати перевірку функціоналу, пов’язаною із цим HTML-елементом.

Для того, аби створити новий веб-елемент в системі, спочатку знаходимо його локатор, за яким він буде знаходитись на веб-сторінці. Для прикладу візьмемо кнопку із посиланнями на зовнішні ресурси (рисунок 3.4).

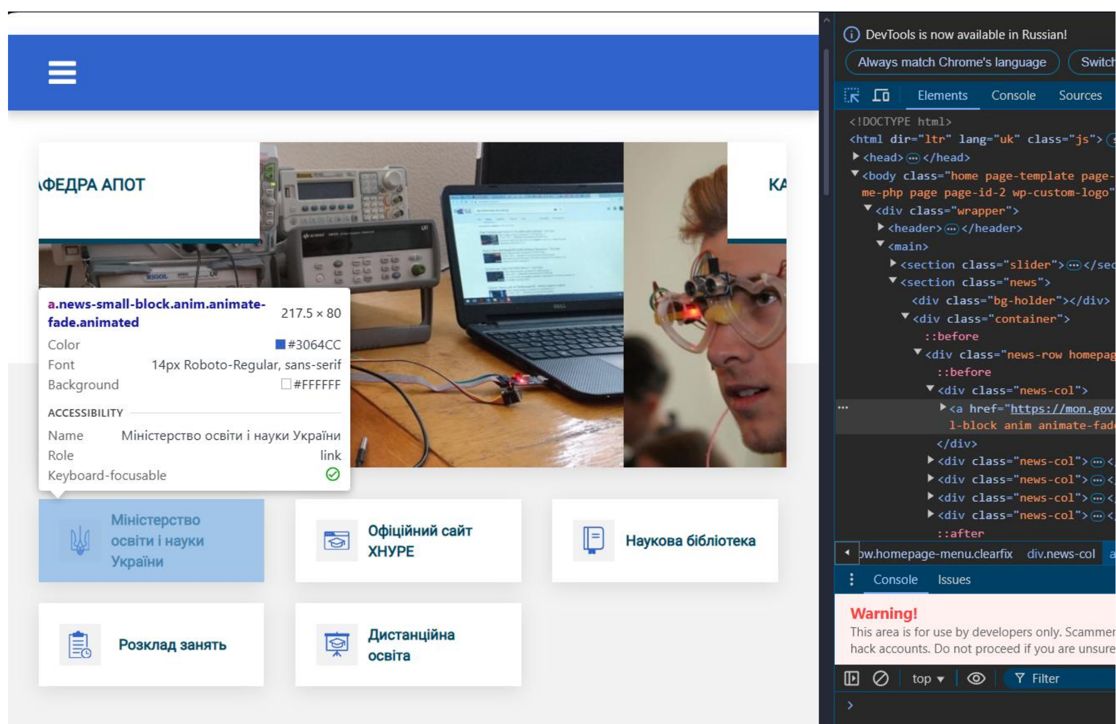


Рисунок 3.4 – Програмний опис кнопки

Бачимо, що дана кнопка має клас із назвою «news-col». Тож, для оголошення локатору використовуємо саме цю назву. Для програмного опису нового HTML-елементу в нашій діагностичній інфраструктурі створюємо нову функцію типу «WebElement». Даний тип даних є частиною фреймворку Selenium і саме він розширює можливість роботи драйверу із компонентами веб-сторінки, що тестується.

Важливо оголошувати веб-елементи через «public» аби цей HTML-елемент був доступним у всіх класах з тестами. У разі використання private даний елемент буде видим лише в рамках даного класу сторінки.

В return даної функції використовуємо наш драйвер та оголошуємо використання методу «findElement». Саме цей метод дозволяє знаходити всі веб-елементи, за заданими в ньому параметрами. Для того, аби задати локатор, за яким слід знайти елемент, оголошуємо «By» та обираємо той ідентифікатор, що необхідний для пошуку елемента.

Після написання тегу із типом локатора в лапках передаємо текстове значення імені цього ж локатора. Приклад оголошення такої функції для отримання доступу до іконки з посиланням на зовнішній сайт за назвою класу було наведено на рисунку 3.5.

```
no usages   new *
public WebElement getCardDescriptionByClassName(){
    return  driver.findElement(By.className("card-body"));
}
```

Рисунок 3.5 – Оголошення нового веб-елементу

3.3 Написання локаторів та функцій завдяки xpath

Оголошення локаторів та пошуку різноманітних веб-елементів також може відбуватись за допомоги xpath. Xpath – це мова, що може бути використаною для задання точного шляху до веб-елемента в документах типу HTML. Ця мова дозволяє знаходити та задавати маршрут до конкретного елементу чи вузла, що описаний в файлі типу HTML.

Перевагами використання xpath в проекті є його можливість вибору елементів за певними та конкретними атрибутами, назвою чи позицією в розмітці. Також xpath розширює можливість пошуку веб-елементу як по всьому файлу типу HTML, як і по його певним блокам.

Більше того, xpath дозволяє та підтримує фільтрацію за різними атрибутами, наприклад ідентифікатором чи назвою класу. Також ця мова дозволяє використання різноманітний функцій, що спрощують та полегшують програмний опис веб-елементів.

Xpath має велику кількість вбудованих методів для роботи із файлами типу HTML, проте найпопулярніші із них є функції для роботи з текстовими форматами, функції для логічних операцій та числові функції.

Серед методів, що присутні в функції для роботи з текстом, наявний метод text(), що надає текст нашого HTML-елемента, є starts-with(), що проводить перевірку з наявності даного тексту в конкретно заданому рядку, є метод contains(), що дозволяє перевірити, чи міститься даний текст в певному елементі. Також присутній substring(), що повертає певний вміст рядка.

До методів, що є у логічних функціях, відносять not() – логічне заперечення та Boolean(), що перетворює значення в форматі true або false. Серед методів наявних у функціях для числових операцій є last() – повертає останній вузл, присутній position() – повертає положення певного вузла, та count(), що рахує кількість знайдених вузлів.

Приклад написання xpath можна розглянути на сторінці, що буде покрита автоматизованими тестами. Відкриваємо dev-tools та отримуємо доступ до HTML-документу цього веб-продукту. Потім переходимо до пошуку елементів (натискаємо Ctrl + F) та пишемо наш xpath.

Спочатку вказуємо «//» для того, аби задати пошук всій сторінці типу HTML. Потім оголошуємо назву блоку, в якому знаходиться наш елемент. Це може бути тег div, a, td, p та подібних. Після оголошення типу елементу, прописуємо квадратні дужки. Це зможе виділити де починається і де закінчується оголошення такого блоку. В саму блоці прописуємо символ собачки та вказуємо тег, за яким варто знайти цей веб-елемент: це може бути id, name, class, href та подібних. Потім ставимо знак дорівнює, одинарні лапки та в лапках текстовим значенням вказуємо назву даного тегу. Приклад написання xpath наведений на рисунку 3.6.

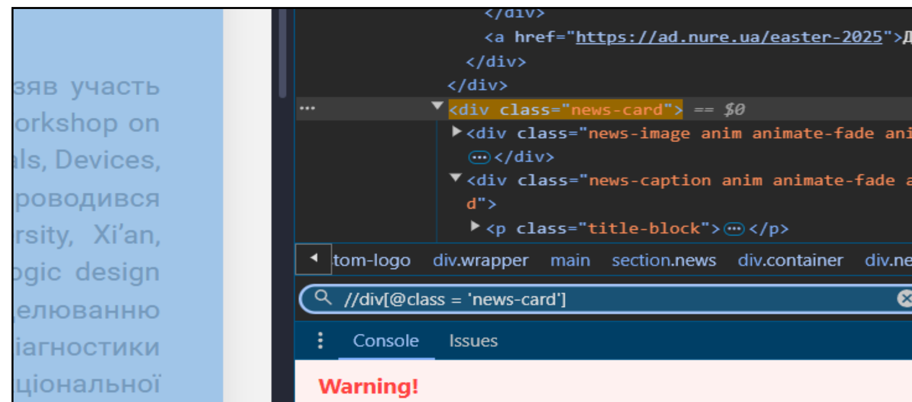


Рисунок 3.6 – Приклад написаного xpath

В автоматизації тестування використання xpath розширює можливості створюваних тестів, дозволяє чітко задати шлях до певного HTML-елементу та зменшує ризик появи проблем, пов’язаних із доступом до елементів.

В контексті веб-елементів використання xpath майже нічим не відрізняється від оголошення веб-елементу із іншим локатором. Проте разом із xpath можливо створити унікальну та гнучку функцію, що буде зменшувати дублювання коду та збільшить його читабельність.

Для цього в текстове значення xpath можливо передати параметр, який буде заповнюватись безпосередньо в автоматизованому тесті. Наприклад, в параметри функції можна передати текстову змінну, яку потім можна використати в xpath та передати дані в неї вже в самому автотесті. Використання змінних із функціями xpath дозволить створити загальні методи для отримання доступу до веб-елементу. Приклад одного із таких веб-елементів наведений на рисунку 3.7.

```
public WebElement getCardByName(String nameOfCard) {
    return driver.findElement(By.xpath( xpathExpression: "//div[@class = 'news-col'][contains(., '" + nameOfCard + "')]"));
}
```

Рисунок 3.7 – Оголошення функції із xpath

В прикладі бачимо, що даний веб-елемент метод contains(), що перевіряє, чи містить даний елемент заданий в параметрі текст. Цей текст

можна буде згодом передавати в автоматизованому тесті. Цей веб-елемент може бути використаним одразу для пошуку всіх п’яти кнопок із посиланнями на зовнішні ресурси: для цього достатньо в самому веб-елементі передати текстове значення заголовку кнопки. Приклад використання цього елементу наведено на рисунку 3.8.

```
    mainPage.getCardByName( nameOfCard: "Міністерство освіти і науки України");
    mainPage.getCardByName( nameOfCard: "Офіційний сайт ХНУРЕ");
    mainPage.getCardByName( nameOfCard: "Наукова бібліотека");
    mainPage.getCardByName( nameOfCard: "Розклад занять");
    mainPage.getCardByName( nameOfCard: "Дистанційна освіта");
}
```

Рисунок 3.8 – Використання веб-елементу із параметром

3.4 Особливості взаємодії із веб-елементами в Selenium

Створені веб-елементи дозволяють отримати доступ до HTML-елементу сторінки, що тестується. Для виконання більших дій варто використовувати основні методи, що наявні в бібліотеці Selenium. Вони дозволяють зімітувати дії реального користувача із програмним продуктом та автоматизувати тестування будь-якого ПЗ. Основні – на рисунку 3.9.

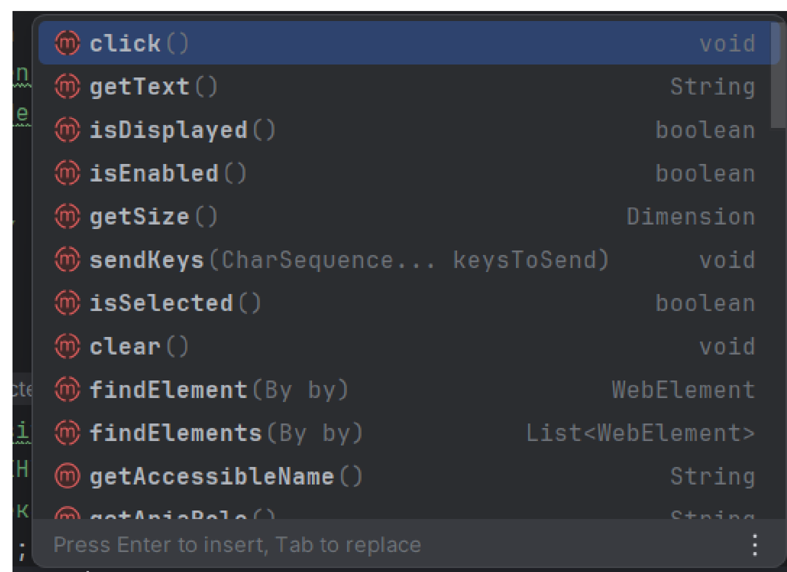


Рисунок 3.9 – Основні методи взаємодії Selenium

Click() дозволяє натиснути на елемент, може бути використаним для перевірок клікабельності тих чи інших елементів. Використовується також для міграції між сторінками, діагностики специфічної логіки та подібного.

GetText() дозволяє отримати точний текст описаного програмно веб-елементу. Є корисним у використанні для автоматизації тестування, наприклад, новинних сайтів та продуктів, які мають в собі взаємодію із великим обсягом тексту. Отриманий із getText() прямо під час пробігу та запуску автоматизованого тесту можливо використати для порівняння очікуваних та реальних слів на сторінці.

IsDisplayed() перевіряє, чи відображається даний елемент на сторінці. Повертає значення true або false. У використанні з функціями Java стане корисним для перевірки видимості елементів у програмних продуктах.

InEnabled() перевіряє, чи доступна взаємодія із певною логікою роботи. Інколи функціонал програмного забезпечення вимагає виведення, наприклад прапорців, статично обраними. Цей метод у вигляді true або false повертає результат перевірки: якщо true, то з елементом можлива взаємодія, якщо false – веб елемент має статично обране значення.

GetSize() дозволяє отримати розмір веб-елементу, його висоту чи ширину. Selenium також дозволяє відкривати сторінки браузера у різних розмірах, тож цей метод може стати корисним у автоматизації тестування гнучкості програмного забезпечення під різні розміру гаджету та екрану.

SendKeys() дозволяє заповнити дані в поля вводу. Може бути використаним для тестування введення даних у формах, спроби авторизації, фільтрації та навіть стійкості ПЗ до хакерських атак. Завдяки цьому методу можливо створити декілька автоматизованих тестів, що будуть заповнювати форму авторизації та реєстрації додатку. Створені тести можливо запустити паралельно, тим самим зімітувавши хакерський напад на продукт.

Clear() дозволяє очистити форму введення. Наприклад, при переході на логіку додатку деякі поля можуть містити раніше введений текст. Цей метод дозволяє його очистити, тим самим звільнивши місце для нового тексту.

Також є ряд методів, які дозволяють взаємодіяти із списками та прапорцями. За допомоги getValue() можливо отримати масив із всіма значеннями, які наявні в цьому веб-елементі. В подальшому цей масив даних завдяки функціям Java можливо порівняти із очікуваним. Якщо масиви будуть рівні, тест пройде, якщо ні – повідомить про помилку.

Більше того, Selenium дозволяє взаємодіяти із клавіатурою робочої машини. Є імітація дій, що можливо виконати із кнопок клавіатури, наприклад натискання клавіші пробілу, enter і подібних. Даний функціонал фреймворку є корисним для перевірки роботи програмного продукту, який може використовуватись людьми із обмеженими можливостями.

Особливістю використання Selenium для взаємодії із веб-елементами є можливість імплементації очікувань в автоматизованих тестах. Існують спеціальні методи, які дозволяють створити функції для очікування завантаження сторінки, видимості елементу, анімації завантаження, тощо. Додати таку логіку можливо різними шляхами. Перший шлях це додавання прямої паузи, під час якої зупиняється проходження всього тесту. Другий шлях це очікування відображення конкретного елемента. Перший підхід є більш стабільним, другий – більш грамотним, оскільки немає необхідності зупиняти весь тест, йде пряме очікування конкретного елементу.

Дані методи, що доступні в бібліотеці Selenium, в поєднанні із функціями Java гарантують створення гнучких та багатофункціональних тестів, які зможуть проводити діагностику більшої частини логіки роботи ПЗ та зможуть знаходити баги ще на ранніх етапах розробки.

3.5 Написання автоматизованих тестів

Написання та створення автоматизованих тестів відбувається у класах, що знаходяться в теці tests. Для зручності роботи із діагностичною інфраструктурою, створюємо окремий клас із тестами на окрему сторінку програмного продукту. Створюємо базовий клас із тестами baseTests. В

ньому додаємо ініціалізацію всіх класів із сторінками, де описані веб-елементи програмного рішення. Для цього створюємо функцію setup та в ній оголошуємо ініціалізацію сторінок з класами через mainPage = new mainPage(). Варто додати оголошення кожного класу, що необхідний для створення автоматизованого тесту. До функції додаємо @BeforeClass.

@BeforeClass – це анотація бібліотеки TestNG, яка дозволяє розділяти певні частини коду на окремі блоки, виконання яких залежить від зазначеного тегу. Наприклад, @AfterClass дозволяє виконати деяку частину коду після завершення пробігу всіх тестів із класу, @BeforeClass відповідає за виконання функцій до запуску всіх тестів, @AfterMethod дозволяє виконувати деякі функції після кожного тесту.

TestNG – це багатофункціональний фреймворк, що дозволяє групувати тести за групами, виконувати їх паралельний запуск та підтримує роботу із різними типами файлів, наприклад XML. Наявністю анотацій, TestNG дозволяє зменшити об’єм коду, запобігши дублюванню функцій в кожному автоматизованому тесті. Він збільшує читабельність та продуктивність коду.

В прикладі із розглянутою діагностичною інфраструктурою використання анотації дозволить оголосити використання класів із сторінками одного разу перед запуском всіх тестів із класом, тим самим прибравши необхідність написання того самого коду в кожному тесті.

Наступним кроком є оголошення тесту в класі із тестом. Для цього використовуємо анотацію @Test, яка перевизначить створену функцію як тест. Дана анотація теж є частиною фреймворку TestNG. Після оголошення тесту створюємо функцію типу void та додаємо їй назву. Важливо кожен тест оголошувати разом із JavaDoc, в якому чітко описується суть та ідея створеного тесту. Це необхідно для дотримання чіткої документації проекту та для кращого розуміння іншими членами команди створюваного тесту.

Для того, аби створити новий коментар для тесту, перед його анотацією додаємо символи «/***/». В середині виразу описуємо необхідну документацію, позначаючи важливі моменти спеціальними анотаціями

JavaDoc, такими як @param, @return, @throws та подібних. Використання цих тегів дозволить створити гнучку документацію, яка буде корисною для спеціалістів із забезпечення автоматизації тестування ПЗ, що ніколи не працювали із даною логікою. Для генерації всієї документації в терміналі виконуємо команду javadoc -d docs mainPageTests.java. Якщо все зроблено правильно, то середовище розробки підтягне доданий коментар до тесту. Приклад використання JavaDoc наведено на рисунку 3.10.

Рисунок 3.10 – Приклад використання JavaDoc

Для безпосереднього написання автоматизованого тесту викликаємо назву необхідного класу із сторінкою веб-додатку, що слід протестувати, та проводимо автоматизацію дій, що треба для діагностики програмного продукту. Важливо зазначити, що дії відбуваються із описаними раніше HTML-елементами, а перевірка коректності роботи застосунку також може відбуватись завдяки функцій, доступних в Java [6].

В мові програмування Java доступна функція Assert. Вона дозволяє перевірити різноманітний стан двох сутностей. Наприклад, завдяки функції Assert.assertTrue можливо перевірити, чи повертає вказана у функції умова відповідь у вигляді true. Основні методи Selenium передають результат своїх перевірок у форматі true або false, тому даний метод Java дозволить перевірити, чи правдива умова, що вказана в перевірці роботи ПЗ. Також є функція assertFalse, яка працює аналогічним чином.

Більше того, Java має функцію Assert. assertEquals, яка перевіряє, чи два значення в середині методу рівні. Цю функцію можна використати у порівнянні текстів у програмному продукту: в тесті отримувати безпосередньо текст веб-елемента та через функцію assertEquals перевіряти, чи дорівнює щойно отримане значення тексту із тим, що вказано як очікуване. Якщо не рівні, тест поверне помилку.

Так в автоматизованих тестах можливо працювати із драйвером напряму. Можливо отримати актуальне посилання сторінки програмного рішення, яка зараз відкрита в основному вікні. У поєднанні із функціями Java можна перевіряти зміну посилання після переходу на нову сторінку, актуальність відкритої сторінки, управляти відкритим вікном, тощо.

Використання описаних методів та функцій допоможе інженерам із забезпечення якості ПЗ створити гнучкі тести для автоматизації тестування різноманітних програмних продуктів.

3.6 Запуск створених тестів в Jenkins

Після написання та створення тестів, розроблена діагностична інфраструктура дозволяє запускати збірку і пробіг всіх автотестів через інструмент Jenkins [7]. Для цього переходимо в створений раніше item, натискаємо кнопку «зібрати зараз». Після цього Jenkins почне забирати внесені зміни в проекті, відповідно оновлювати проект та в результаті запускати всі функції, які помічені анотацією @Test. По завершенню роботи цей інструмент виведе список внесених змін до проекту та назви тестів, що не пройшли під час збірки

Також до Jenkins було додано ряд додатків, таких як TestNG Results та Zephyr Scale. Вони дозволяють покращити аналіз звітів, що генеруються автоматично після проходження всіх тестів в проекті. TestNG дозволяє згенерувати практичні графіки із візуалізацією успішно та негативно пройдених тестів. Приклад наведено на рисунку 3.11.

Рисунок 3.11 – Приклад візуалізації пройдених тестів

Для кращого аналізу можна використати системи по типу Zephyr Scale. Створена діагностична інфраструктура дозволяє надіслати результат по пройденим тестам до Jira, аби в майбутньому згенерувати гнучкі звіти в додатку Zephyr Scale. Цей підхід допоможе краще управляти збірками, розподіляти тести між членами команди, подібного.

Для підтримки такої інтеграції було додано клас ZephyrUploader, в якому можливо виконати REST запит напряму до API Zephyr Scale. При вказанні вірних даних, таких як API access token та ключа проекту, в Zephyr створиться новий звіт із актуальними результатами проходження збірки. Приклад таких звітів наведено на рисунках 3.12.

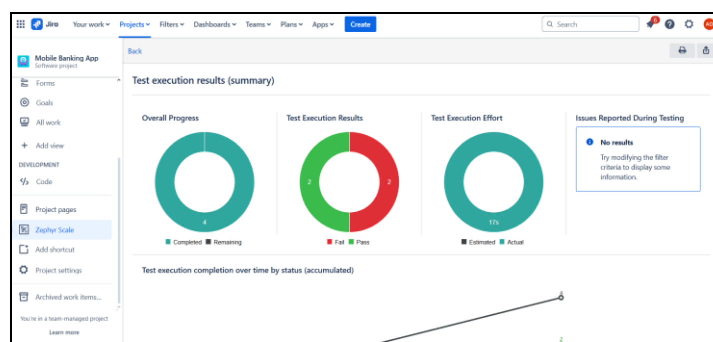


Рисунок 3.12 – Графічне зображення результату пробігу тестів

Тож, розроблена діагностична інфраструктура дозволяє проводити аналіз результатів із автоматизації тестування двома способами: через Jenkins та через Zephyr Scale. Ці два способи є ефективними та продуктивними у своєму використанні, адже є лідерами на ринку IT.

ВИСНОВКИ

Якісне ПЗ – це запорука успіху роботи у будь-якій галузі, адже програмні рішення оточують людину абсолютно у всіх сферах її життя. Саме від правильної роботи додатків залежить результат вирішення більшості проблем та задач, поставлених сьогоденням.

В ході виконання кваліфікаційної роботи було розроблено діагностичну інфраструктуру, здатну виконувати автоматизацію тестування різноманітних програмних продуктів та додатків. Дана система є гнучкою, адже з легкістю підтримує тестування різних веб-рішень: достатньо інтегрувати посилання на додаток в систему, після чого інфраструктура буде готова для впровадження автоматизації тестування розглянутого ПЗ.

Для розробки такої системи було використано сучасні рішення та інструменти, використання яких дозволяє створити практичну інфраструктуру, готову до інтеграції із різними системами та проектами. Серед технологій, що були використані для впровадження в рамках кваліфікаційної роботи проекту, є мова програмування Java, середовище розробки IntelliJ IDEA, фреймворки Selenium, TestNG, Junit, система безперервної інтеграції Jenkins та додаток для аналізу звітів Zephyr Scale.

Розроблена діагностична інфраструктура може бути використаною на реальних та комерційних проектах ринку IT для діагностики різноманітних програмних продуктів, адже для її впровадження було застосовано лише новітні та потужні інструменти роботи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Життєвий цикл розробки програмного забезпечення (Software Development Life Cycle - SDLC). URL: https://www.maxzosim.com/software-development-life-cycle-sdlc/ (дата звернення 10.05.2025).

2. Глюза М., Вовк О. Usability-тестування як ефективний показник успішності веб-продуктів // Науковий простір: актуальні питання, досягнення та інновації. 2023. С. 348-350.

3. А.А Омельницький. Дослідження інструменту Jenkins для регресійного тестування / XXIX Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. – Харків: ХНУРЕ, 2025. – С.52-54.

4. Що таке GitHub і як з ним працювати. URL: https://training.qatestlab.com/blog/technical-articles/what-is-github-and-how-to-work/ (дата звернення 15.05.2025).

5. Welcome to the Zephyr Documentation. URL: https://support.smartbear.com/zephyr/docs/en/welcome.html (дата звернення 17.05.2025).

6. Rahul S. Hands-On Automation Testing with Java for Beginners. 2018. C.95

7. Laster B. Jenkins 2: Up and Running. 2018. С. 102-110.

8. А.А Омельницький, В.О. Кошель, О.О. Супрун. 2D-відстеження об’єктів у відеопослідовності / XXVIII Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. – Харків: ХНУРЕ, 2024. – С.123-125.