МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ЦЕНТР ЗАОЧНОЇ, ДИСТАНЦІЙНОЇ ТА ВЕЧІРНЬОЇ ФОРМ НАВЧАННЯ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК
СЕКЦІЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ПРОЕКТУВАННЯ

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему

«Інформаційна технологія автоматизації тестування API та web-сервісів»

за спеціальністю 122 «Комп'ютерні науки»,
освітньо-професійна програми «Інформаційні технології проектування»

Виконавець роботи: студент групи ІТмз91с

Нестерук Богдан Володимирович

Кваліфікаційну роботу захищено на засіданні ЕК з оцінкою «____» грудня 2020 р.

Науковий керівник: _____ к. т. н., Нагорний В.В.
(підпис)

Голова комісії: _____ Шифрін Д.М.
(підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Суми-2020

Сумський державний університет

Центр заочної, дистанційної та вечірньої форм навчання
Кафедра комп'ютерних наук
Секція інформаційних технологій проектування
Спеціальність 122 «Комп'ютерні науки»
Освітньо-професійна програма «Інформаційні технології проектування»

ЗАТВЕРДЖУЮ
Зав. секцією ІТП

В. В. Шендрик
«_____» _____ 2020 р.

ЗАВДАННЯ

на кваліфікаційну роботу магістра студентові

Нестерук Богдан Володимирович
(прізвище, ім'я, по батькові)

1 Тема проекту *Інформаційна технологія автоматизації тестування API та web-сервісів*

затверджена наказом по університету від « 16 » листопада 2020 р. № 1773-III

2 Термін здачі студентом закінченого проекту « 7 » грудня 2020 р.

3 Вхідні дані до проекту
1. Технічне завдання
2. Перелік функціональних вимог до технології

4 Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1. Аналіз предметної області
2. Постановка задачі
3. Проектування інформаційної технології
4. Розробка інформаційної технології

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Презентація (21 слайд)

6. Консультанти випускної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Аналіз предметної області			
Постановка задачі			
Проектування інформаційної технології			
Розробка інформаційної технології			

Дата видачі завдання _____

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів випускної проекту	Термін виконання етапів проекту	Примітка
1	Аналіз предметної області	01.10.2020 – 10.10.2020	
2	Аналіз технік тест-дизайну	11.10.2020 – 25.10.2020	
3	Ідентифікація ідеї та мети проекту	25.10.2020 – 26.10.2020	
4	Аналіз технологій	26.10.2020 – 05.11.2020	
5	Вибір інструментів реалізації	05.11.2020 – 10.11.2020	
6	Планування WBS	11.11.2020 – 12.11.2020	
7	Планування OBS	13.11.2020 – 13.11.2020	
8	Розробка календарного плану	13.11.2020 – 15.11.2020	
9	Визначення ризиків	15.11.2020 – 15.11.2020	
10	Проектування інформаційної технології	15.11.2020 – 20.11.2020	
11	Розробка модулю тестів WEB UI	20.11.2020 – 24.11.2020	
12	Розробка модулів тестів REST API та DB	24.11.2020 – 29.11.2020	
13	Реалізація взаємодії з сервісами CI	29.11.2020 – 02.12.2020	
14	Підготовка презентації	03.12.2020 – 04.12.2020	
15	Захист дипломного проекту	16.12.2020	

Магістрант _____ Нестерук Б.В.

Керівник роботи _____ к. т. н., Нагорний В.В.

РЕФЕРАТ

Тема кваліфікаційної роботи магістра: «Інформаційна технологія автоматизації тестування API та web-сервісів».

Актуальність теми. Початкова фаза автоматизації тестування веб-сервісів потребує значних ресурсів для розробки архітектури проекту, тестових наборів та допоміжних інструментів. На даний момент серед бібліотек автоматизації тестування не існує комплексного рішення даних питань.

Мета і задачі. Метою роботи є створення інформаційної технології автоматизації тестування API та веб-сервісів у вигляді каркасного проекту для забезпечення швидкого старту автоматизації тестування графічних інтерфейсів, API та баз даних.

Структура та обсяг роботи. Пояснювальна записка складається зі вступу, 4 розділів, висновків, 3 додатків, включає 84 сторінки, 4 таблиці, 28 рисунків, 34 джерела.

Основний зміст роботи. В першому розділі дано визначення автоматизації тестування, проведено огляд поширених підходів з автоматизації. Наведено порівняльну таблицю існуючих систем автоматизації. Другий розділ присвячений формулюванню мети, задач та вибору інструментів реалізації. Виконується планування робіт для реалізації проекту. Третій розділ описує поетапне проектування інформаційної технології. Побудовані контекстна діаграма та діаграми декомпозиції 1-го та 2-го рівнів у нотації IDEF0, описана взаємодія між акторами та ресурсами, наведена загальна архітектура технології. У четвертому розділі описується процес реалізації механізмів параметризації, абстрагування, запуску тестових наборів. Проводиться поєднання компонентів технології у суцільну структуру.

Ключові слова. АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, ЗОВНІШНЯ ПАРАМЕТРИЗАЦІЯ, ПАРАМЕТРИЗОВАНІ ТЕСТОВІ СЕСІЇ, ТЕСТУВАННЯ API, ТЕСТУВАННЯ UI, БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ, ТЕСТОВІ ЗВІТИ, УПРАВЛІННЯ ТЕСТОВИМ СЕРЕДОВИЩЕМ.

ЗМІСТ

ВСТУП

Наразі технології веб-сервісів вийшли далеко за рамки статичних енциклопедій чи інтернет-магазинів. Мережева архітектура “клієнт-сервер” стала домінуючою ідеєю для реалізації розподілених інформаційних систем та невід’ємною частиною сучасного життя, будучи втілена у мільйони проектів [1]. В суспільстві, де 92% розрахунків відбувається у безготівковій онлайн формі, а випробування двигунів та медичних імплантів проводяться інженерами у симуляторах фізичних процесів, найменший дефект може спричинити незворотні економічні наслідки, спростувавши репутацію компанії-замовника та розробників програмного забезпечення.

Для уникнення неочікуваної поведінки системи потрібні не лише потужні мови програмування, останні версії бібліотек розробки та зниження залежностей між компонентами системи. Тестування програмних продуктів проходить паралельно на усіх рівнях роботи: від unit-тестів для функцій, що створюються програмістами, до системних тестів без втручання у внутрішню архітектуру веб-сервісу (тестування чорного ящика, black-box testing [2]). Останні проектуються інженерами із забезпечення якості, що підходять до взаємодії з системою з точки зору користувача, іноді додаючи творчий підхід дослідника (exploratory testing [3]) та включаючи в тестування випадки, що важко передбачити при розробці програмного коду.

Забезпечення якості (quality assurance, QA) програмного забезпечення є невід’ємною частиною процесу розробки від етапу створення специфікації до підтримки продукту після його видачі. Включаючи активності по розробці тестової документації, ітеративне виконання тестових сценаріїв та обробку результатів, забезпечення якості саме по собі є затратним процесом з точки зору часових, людських, а отже – фінансових ресурсів. Завдання автоматизації тестування – підвищувати його ефективність, беручи на себе такі аспекти як:

- виключення людського фактору (уникнення помилок при введенні даних та проходженні тестових сценаріїв);

- повтори тестів цілодобово за розкладом, або при оновленні коду програмного продукту;
- паралельні сесії тестування;
- збір та оптимізації результатів (виключення заздалегідь неінформативних даних з репортів);

Починаючи автоматизацію на проекті, команда розробників та QA-інженерів, як правило, стикається з питанням: залучати сторонніх інженерів-автоматизаторів, чи поступово розроблювати архітектуру проекту автоматизації самостійно. Зазвичай, замовнику програмного продукту не цікаві додаткові витрати на розширення команди QA, тож додання автоматизованих тестів розтягується на місяці перебору підходів, випробувань бібліотек та шаблонів автоматизації, що неминуче відображається на якості як самих авто-тестів, так і на процесі забезпечення якості на проекті в цілому.

Ідея даної роботи – розробити технологію для швидкого старту автоматизації будь-якої веб-орієнтованої системи, незалежно від стеку технологій основного продукту: починаючи автоматизацію тестування, інженер підключає дане рішення, та, наслідуючи реалізовані інструменти згідно інструкціям, розширює проект своїми тестовими сценаріями, кожен з яких можна запустити окремо, або в наборі з іншими тестами за розкладом на різних середовищах. Дана робота відноситься в першу чергу до функціональних тестів системного рівня (Web UI, REST API, Database testing).

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність питання

Сучасне суспільство неможливо уявити без інтенсивного використання інформаційних технологій. Ми живемо в еру інформації, де кожен економічний процес, як і будь-який зв’язок між окремими людьми або групами людей, повністю залежить від якості передачі даних та знань. Стрімкий розвиток інформаційних ресурсів та сервісів підтримується не тільки суспільним розвитком, а й активним розвитком технічної складової інформаційних технологій, нових пристроїв та програмного забезпечення.

В останнє десятиріччя веб-сервіси, реалізовані за клієнт-серверною архітектурою, вийшли на передній план розподілених інформаційних систем. Десктопні (ті, що потребують встановлення безпосередньо на робочу станцію) додатки мають ряд суттєвих недоліків: потреба в інсталяції, оновлення версії продукту на кожній станції, боротьба з підробкою ліцензій, залежність від операційної системи та інше. Тому, перехід до клієнт-серверної технології поряд з наданням доступу через веб-інтерфейс браузерів, або REST API-точок доступу [4] є природним продовженням надання інформаційних послуг користувачам. У цьому варіанті головне програмне забезпечення розгортається на одному сервері, або кластері серверів, що адмініструють професіонали. Оновлення проходить майже непомітно та одночасно для усіх користувачів, а видача ліцензії заміщається підпискою на послуги.

Купуючи підписку на послугу, клієнт сплачує не за довічне користування певною версією продукту; оплата підписки надає можливість використання найновішої версії, але обмежений період часу. Перевірка користувача на право використання послуг також виконується на серверній частині замість комп’ютера користувача, тому підробка або злом стає неможливим.

Заміщення ліцензії підпискою, до речі, допомогло багатьом корпораціям утриматись на плаву в період кризи світового масштабу 2020 року: під час

весняно-літнього спаду економіки клієнти, маючи ліцензії на певну версію продукту, просто відмовлялися від купівлі ліцензій на нові версії. В той час, як закінчення строку дії підписки неминуче означає припинення роботи, а отже необхідність нової передплати, тобто надходження коштів власникам програмних продуктів.

Однією з найважливіших тенденцій сучасних веб-орієнтованих інженерних систем є те, що їх вплив на організацію основного виробництва об'єктивно посилюється, а технічний рівень і складність підвищується. Наглядним прикладом подібних систем можуть служити продукти корпорації Dassault Systemes [5]: розробки в областях медицини, машинобудування, авіації та конструкцій проходять безліч ітерацій у симуляторах фізичних процесів перш, ніж буде створений фізичний прототип серцевого клапану, тормозної системи автомобіля або системи опорних ферм торгівельного центру.

Симуляційне програмне забезпечення встановлене на потужних серверах, адже деякі види симуляції потребують використання 400 та більше ядер процесорів одночасно. Водночас, інженерам надається можливість увійти до системи через будь-який веб-браузер, де вони можуть створити свій власний простір, завантажити файли моделей, конфігурувати процес експерименту та аналізувати результати.

Такий підхід не лише економить час та фінанси: найчастіше деякі дані неможливо зібрати у фізичному світі через недоліки обладнання, або сам процес експерименту. Уявімо собі випробування моделі тормозної системи автомобіля, що вийшла с ладу через перегрів. В реальному світі надзвичайно складно точно дослідити напрям потоків повітря повз модель або розповсюдження зони перегріву, адже після завершення експерименту частини усієї системи холонуть за лічені секунди. З іншого боку, експерименти в симуляторах є дискретними, тобто точний стан як моделі так і середовища можна побачити у будь-який момент експерименту. А, змінюючи параметри моделі, експеримент може бути повторений дизайнерами та інженерами сотні разів для знаходження найефективнішої конфігурації.

Зрозуміло, що приведена вище схема може використовуватись одночасно багатьма клієнтами, що, найчастіше, є конкурентами. Беручи за приклад автомобілебудування, BMW, Bentley та Tesla Motors розробляють кузови своїх автомобілів, використовуючи одні й ті ж самі ресурси симуляторів повітряних потоків.

Беручи до уваги приклади приведені вище, стає зрозумілою вартість помилки (втрата, помилкове поширення, неактуальність даних) в роботі таких продуктів, та важливість забезпечення стабільності в роботі кожного з варіантів використання системи.

З розвитком та ускладненням клієнт-серверних додатків збільшується відповідальність QA-інженера та час, потрібний на проходження тестових сценаріїв, а, отже, збільшується ризик помилки або непоміченого дефекту. А сама ідеологія «клієнт-сервер» передбачає комунікацію між різними клієнтами (ВЕБ-браузер, скрипт, що викликає API тощо) та програмним забезпеченням на сервері, що збільшує різноманіття використовуваних технологій та шаблонів проектування. Окрім цього, безліч випадків неможливо протестувати без допомоги спеціальних інструментів: схеми фінансових розрахунків, аналіз даних в базах знань, або тестування швидкодії веб-сервісу, як правило, потребують великих об’ємів даних для перебору та перевірки очікуваних результатів.

Саме цьому автоматизація функціонального та нефункціонального тестувань виходить на новий рівень та с кожним роком поширюється між інтерпрайз-проектами. Наряду з unit-тестами, інженери з автоматизації тестування проектують системні тести, що імітують дії користувача на сайтах, оперують викликами API, аналізують швидкість відповідей веб-сервісу залежно від умов середовища та кількості даних для обробки.

Автоматизація процесу тестування в змозі вирішити проблему людських ресурсів: автоматизований сценарій можна повторити безліч разів, запустити паралельно безліч сесій в різних середовищах, запускати вночі, автоматично при додаванні нового коду тощо [6]. При кожному старті автоматизований тест завжди виконує описані дії в однаковому порядку, викреслюючи людський

фактор (непомітна помилка в введених даних, що приводить до іншого результату в кінці тесту).

З іншого боку, для створення авто-тестів потрібні знання мов програмування, шаблонів автоматизації, підходів з налаштування середовища. На прикладі функціональних авто-тестів через інтерфейс веб-браузера, необхідно організувати роботу з елементами сторінок таким чином, щоб тест знаходив їх та проводив операції над ними (кліки мишкою, ввід-зчитування тексту, або drag-and-drop) стабільно в різних веб-браузерах, версіях та на різних операційних системах. Окрім цього, при зміні дизайну сторінки тест повинен легко адаптуватись до змін. Найчастіше, один і той же сценарій потрібно повторити кілька разів з різними вхідними, а отже, і різними очікуваними даними. Це потребує вищої кваліфікації робітників, додаткових затрат на створення архітектури та постійної підтримки уже існуючих автоматизованих сценаріїв.

Тестування оплачує замовник, а підтримка створених тестів – завдання команди із забезпечення якості. Хоча обидві сторони зацікавлені у стабільних тестах та однозначних результатах, найчастіше, команді розробки складно аргументувати замовнику необхідність додаткових витрат на проектування архітектури авто-тестів.

Майже кожен аспект автоматизації має безліч рішень, але організація їх в одну ефективну та гнучку систему, найчастіше, є складним завданням, що займає місяці опрацювання документації та проб завдяки різноманіттю технологій та бібліотек, якими потрібно оперувати. Тому було вирішено розробити інформаційну технологію, яка дозволить QA-команді зосередитися саме на логіці тестових сценаріїв замість вирішення проблем інфраструктури, що і визначає актуальність даної роботи. Розроблювана технологія повинна забезпечувати:

- автоматизацію тестування на системному рівні (WEB UI, API, DB);
- перевірені шаблони з організації даних та тестових сценаріїв;
- зручну систему оновлення версій та залежностей компонентів;
- організацію сценаріїв у черги, набори, групи, задання залежностей;
- зберігання тестових даних окремо від коду (параметризацію);

- детальні вихідні звіти, готові для показу замовнику;
- роботу в системах безперервної інтеграції;
- роботу тестів в різних середовищах.

Така технологія з ліцензією на вільне використання [7] може вирішити вище перелічені проблеми інженерів-автоматизаторів, а, будучи втіленою у бібліотеку з відкритим програмним кодом, вирішить проблему актуалізації та розширення завдяки спільноті таких ресурсів, як gitlab.com, stackoverflow.com.

1.2 Аналіз методів вирішення

Тестуванням програмного забезпечення є перевірка відповідності між реальною та очікуваною поведінкою програмного продукту, що здійснюється на кінцевому наборі тестів, обраному певним чином. Його метою є підвищення ймовірності того, що програмний продукт, призначений для тестування, буде працювати правильно та при будь-яких обставинах буде відповідати всім описаним вимогам. Інша мета тестування – надання актуальної інформації про стан продукту на даний момент [8].

Кожен процес тестування можна віднести до певного рівня інформаційної системи:

- модульне тестування (Unit Testing) – перевіряє функціональність в частинах продукту, які доступні і можуть бути протестовані окремо один від одного (модулі програм, об’єкти, класи, функції). Як правило, авто-тести на даному рівні складаються автором коду, що тестується;
- інтеграційне тестування (Integration Testing) – перевіряється взаємодія між компонентами системи після проведення модульного тестування;
- системне тестування (System Testing) – тут основним завданням є перевірка як функціональних, так і нефункціональних вимог до системи в цілому. При цьому виявляються дефекти, такі як неправильне використання ресурсів системи, непередбачені комбінації даних

користувача, несумісність з середовищем, непередбачені сценарії використання;

На модульному (unit-tests) рівні тестування веб-сервісів мало чим відрізняється від тестування інших типів додатків: розробник коду створює авто-тести, що тестують створені функції, використовуючи звичні йому технології: мову програмування, менеджер тестів. І хоча створення unit-тестів не потребує втручання сторонніх інженерів із забезпечення якості, воно не може гарантувати відстеження регресії після змін в інших модулях продукту, або в роботі суміжних сервісів.

Водночас, підходячи зі сторони користувача системи в цілому, інженери-автоматизатори мають змогу знайти помилки в поведінці, непомітні для unit-тестів, але, найчастіше, не мають інформації про першопричину помилки: наприклад, повернений з серверу код 500 свідчить про те, що програмний код на стороні сервера не зміг опрацювати запит, але навряд чи в відповіді сервера буде представлений повний стек викликів методів та дані, що оброблялися. Тому, надання інформації про вхідні дані тесту, момент помилки, логи браузера та сервера є не менш важливим аспектом автоматизації тестування, ніж сам код авто-тестів.

Автоматизація тестування саме системного рівня має рішуче значення для виявлення регресії та забезпечення цілісності усієї системи, так як проходження варіанту використання програмного продукту «від початку до кінця» може складати кілька сотень кроків, та ділитися на суміжні варіанти дій, що робить виконання таких сценаріїв вручну практично неможливим на регулярній основі. Окрім цього, ризик втручання людського фактору збільшується пропорційно кількості перевірок.

Підготовка до автоматизації тестових процесів починається наряду з іншими роботами з розробки програмного продукту (аналіз та вибір технологій, оточення для прогону тестів), але самі автоматизовані сценарії реалізуються інженерами лише після створення тестового плану та опису найбільш пріоритетних тестових випадків, адже сценарії автоматизованих тестів завжди

повинні спиратися на тестову документацію з чітко визначеними кроками та очікуваним станом після кожної дії. Інакше, з часом, після змін в бізнес-логіці, актуалізація авто-тесту стане неможливою через незрозумілі агенти взаємодії (елементи сторінок, методи API тощо), що описані в коді самого тесту без посилань на затверджену документацію.

Перед складанням тестових сценаріїв інженер із забезпечення якості з'ясовує пріоритети та необхідні види тестів з командою розробки та замовником, визначаються групи тестів, набір середовищ, тип запуску тестових прогонів. Для формалізації процесу забезпечення якості складається тест план [9] тестування програмного продукту – офіційний документ, що містить опис активностей із забезпечення якості з датами та строками завершення, видами тестувань, також описує середовища, де проводяться тестування та визначає аспекти, які не підлягають тестуванню.

Видами тестувань, які доречно автоматизувати в першу чергу, є:

- функціональне тестування (Functional Testing) – розглядає заздалегідь описану поведінку і ґрунтується на аналізі специфікації функціональності компонента або системи в цілому;
- тестування безпеки (Security testing) – стратегія тестування, яка використовується для перевірки безпеки системи, а також для аналізу ризиків, пов'язаних із забезпеченням цілісного підходу до захисту веб-сервісу, атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних;
- тестування взаємодії (Interoperability Testing) – функціональне тестування, що перевіряє здатність програмного продукту взаємодіяти з одним і більше компонентами або системами, і включає в себе тестування сумісності і інтеграційне тестування;
- тестування навантаження (Load testing) – це тестування, що імітує роботу певної кількості бізнес користувачів на якомусь загальному ресурсі;

- об’ємне тестування (Volume Testing) – отримання оцінки продуктивності при збільшенні обсягів даних в базі даних програми;
- регресійне тестування (Regression testing) – це вид тестування, спрямований на перевірку змін, зроблених в програмному продукті, або навколишньому середовищу (лагодження дефекту, злиття коду, міграція на іншу операційну систему, базу даних, веб-сервер), для підтвердження того факту, що існуюча раніше функціональність працює як і раніше. Регресійний можуть бути як функціональні, так і нефункціональні тести;
- тестування інтерфейсу користувача (GUI Testing) – функціональна перевірка інтерфейсу на відповідність вимогам – розмір, шрифт, колір, положення елементів;

Одиницею процесу тестування є тестовий сценарій (Test Case) – артефакт, що описує сукупність дій, конкретні умови та параметри, необхідні для перевірки реалізації функцій або частин проекту, що тестується. Як правило, кожній дії в сценарії відповідає певний очікуваний результат. Окрім цього, тестовий сценарій завжди відноситься до певного розділу (Feature) продукту, наприклад розділ «Звітність» (“Reporting” Feature) та до певної історії користувача (User story) продукту, наприклад, «Звіт про відпустки» (“Vacation report” story). Тестові сценарії також включаються до так званих груп (Group) – наскрізні мітки, що дозволяють знайти тести не певного розділу продукту, а тести в усіх розділах, але певної логіки. Наприклад, тестовий сценарій з розділу «Звітність» може належати до груп «Адміністративний модуль» та «Валідація форм». Таким чином, при зміні програмного коду, що відповідає за перевірку введених даних, можна виконати перевірки сценаріїв, що торкаються заповнення форм. Сценарій може бути залежним від іншого: наприклад, немає сенсу виконувати тест на перевірку обов’язкових полів на формі, якщо позитивний сценарій на відкриття та заповнення форми не пройшов.

Тестові сценарії і є та документація, що лягає в основу кожного з автоматизованих тестів (таблиця 1.1). Атрибути сценарію, такі, як Feature, User

story, Groups, Dependencies переносяться у метадані авто-тесту таким чином, що його можна локалізувати та поставити на виконання на основі доданих міток.

Таблиця 1.1 – Приклад тестового сценарію

Feature: Reporting						
User story	**Name**	**Steps**	**Expected result**	**Groups**	**Dependencies**	**Automated**
Vacation report	Vacation_report_3: Missing end date	**Preconditions:** A user with account manager is at administration -> vacation reports page		Administration Validation	Vacation_report_1	
		Click the "New report" button	New report dialog is displayed			yes
		Select the type "Vacation"				yes
		Select the "user1" employee	The user is select in autocomplete search field			yes
		Set the start date "01.05.2020"				yes
		Keep the end date field empty				yes
		Click the "Generate report" button	The dialog is not closed Red stroke is displayed next to the end date field No requests were sent to the server			yes

Набір дій, що виконуються в тестових сценаріях, визначається після ретельного аналізу бізнес-логіки програмного продукту, а вхідні дані комбінуються таким чином, щоб покрити собою найбільш широкий набір умов, одночасне виконання яких повинне призвести до очікуваного результату.

Для упорядкування та поетапної перевірки складних бізнес вимог існують техніки тест-дизайну. Найпотужнішою та найпоширенішою серед них є Таблиця прийняття рішень (Decision table). Таблиця складається з «Умов» («Conditions») та «Дій» («Actions»), інформація про які береться з проектної документації. До таблиці додаються стовпчики з комбінаціями можливих умов. Для кожної комбінації записується очікуваний результат. Такий вид запису гарантує, що жодна комбінація не буде пропущена (таблиця 1.2). Також стає очевидним, що «значення 2» є надлишковим та може бути виключено з тестування, так як значення 3 і 4 повністю перекривають його.

Таблиця 1.2 – Таблиця прийняття рішень тестового випадку «Авторизація»

Умова	**Значення 1**	**Значення 2**	**Значення 3**	**Значення 4**
Ввод коректних даних в поле E-mail	+	—	+	—
Ввод коректних даних в поле Password	+	—	—	+
Ввод некоректних даних в поле E-mail	—	+	—	+
Ввод некоректних даних в поле Password	—	+	+	—
Дія				
Авторизація виконана	+	—	—	—
Видана помилка: "Введені невірні дані"	—	+	+	+

Для реалізації системного тестування вводиться поняття тесту «від а до я» (end-to-end, E2E). Мається на увазі набір сценаріїв, де кожен наступний залежить від попереднього, а їх скінченний набір повністю описує варіант використання системи від початку роботи (авторизація або реєстрація користувача) до аналізу результатів, згенерованих користувачем (перевірка отриманих даних, створених файлів та інших артефактів роботи веб-сервісу).

1.2.1 Автоматизація функціонального тестування через WEB UI

В даному випадку відбувається заміщення ручних дій тестувальника ПО автоматизованим програмним забезпеченням: драйвером браузеру [10], або скриптами, що виконуються в консолі браузера. Для серверної частини веб-сервісу найчастіше обидва варіанта залишаються однаковими, але для частини коду, що виконується на стороні браузера (клієнті), переважним є варіант з драйвером: в такому випадку сам браузер не помічає різниці між діями людини і авто-тесту, адже для натискання на гіперпосилання або введення тексту в поле викликаються одні й ті ж методи браузеру. Іншими словами, браузер не зможе відрізнити текст, введений з клавіатури, від тексту, що був вставлений в поле авто-тестом. При цьому, зворотною стороною є те, що кожен браузер та навіть різні його версії можуть по-різному оброблювати такі події, тобто розробник авто-тестів повинен автоматизувати вибір драйверу браузера залежно від операційної системи, типу браузера та його версії.

Основними тезами автоматизації через інтерфейс користувача в браузері (User Interface, UI) є:

- локалізація елементів веб-сторінки (кнопки, поля, тексти в HTML) завдяки набору селекторів, таких як ID, CLASS, CSS, XPATH;
- дії над елементами: введення тексту, кліки, отримання текстів;
- очікування певного стану елементу: показаний/схований, маючий певний текст або присутній в певній кількості;
- робота з логами браузерів: запити на сервер, відповіді, деталі помилок;

1.2.2 Автоматизація тестування відображення веб-сторінок

Даний тип фактично є додатком до попереднього: використовуються ті ж технології управління веб-браузерами, але в іншому ключі: авто-тест проходить у двох режимах:

- еталонний старт: інженер впевнений в робочому стані веб-сервісу та хоче зберегти зовнішній вигляд для наступних тестів;
- старт в режимі тестування: тест повинен перевірити відповідність актуального зовнішнього вигляду сторінки збереженому варіанту;

В обох випадках роботу виконує один и той же тест: приводить веб-сторінку до потрібного вигляду (відкриває форми, масштабує вікно, відсортовує список тощо) та робить її знімок. Різниця складається у тому, що тест або зберігає запис сторінки у відповідну директорію (шлях до неї повинен включати в себе тип ОС, назву браузеру, версію, розмір вікна, назву сторінки та інше), або порівнює отриманий знімок з раніше збереженим еталоном. Таким чином, при зміні поведінки веб-сервісу зміни потрібно зробити лише в одному тесті. Основними тезами такої автоматизації є:

- ігнорування динамічних областей сторінки (банери реклами, тексти новин тощо);
- необхідність прогону авто-теста у двох режимах: еталонний та тестовий;
- механізм порівняння еталонного та реального знімку, задання допустимого порогу;
- ігнорування таких елементів сторінки, як положення смуг прокрутки, курсору миші;

1.2.3 Автоматизація функціонального тестування через API

API – це Application Programming Interface, або «відкритий програмний інтерфейс», за допомогою якого одна програма може взаємодіяти з іншою. API дозволяє надсилати інформацію безпосередньо з однієї програми в іншу, минаючи інтерфейс взаємодії з користувачем. API може бути внутрішнім, приватним – коли програмні компоненти пов’язані між собою і використовуються всередині

системи. А може бути відкритим, публічним – в такому випадку воно дозволяє зовнішнім користувачам, або іншим програмам отримувати інформацію, яку можна інтегрувати в свої додатки.

Щоб програми спілкувалися між собою, їх API потрібно побудувати за єдиним стандартом. Одним з них є REST – стандарт архітектури взаємодії додатків і сайтів, що використовує протокол HTTP. Особливість REST в тому, що сервер не запам’ятовує стан користувача між запитами. Іншими словами, ідентифікація користувача (авторизаційний токен) і всі параметри виконання операції передаються в кожному запиті. Цей підхід настільки простий і зручний, що майже витіснив всі інші.

Тестування API проводять, ґрунтуючись на бізнес-логіці програмного продукту. Тестування API відноситься до інтеграційного тестування, а, значить, в ході нього можна відловити помилки взаємодії між модулями системи або між системами. Для тестування використовують спеціальні інструменти, де можна відправити вхідні дані в запиті і перевірити точність вихідних даних.

Для роботи з API програмного продукту розробник повинен надати документацію до кожного методу, або ж слідувати ідеології HAL [11], коли користувачу надається лише початкова точка доступу до API, а відповіді на кожний запит несуть у собі інформацію про доступні (зв’язані) ресурси та формат звернення до них.

Основними аспектами тестування REST API є:

- конфігурація та відправка HTTP-запитів на сервер;
- отримання відповіді;
- перевірка метаданих відповіді (статус код, хедери);
- перевірка структури відповіді (чи відповідає та еталонній JSON або XML схемі [12]);
- навігація по графу відповіді та перевірка певних полів (тип, розмір, очікувані значення);

1.2.4 Автоматизація тестування навантаження

Даний тип тестування може проводитися як комбінацією попередніх типів (контрольоване зростання кількості паралельних потоків тестів, їх повторення), так і спеціалізованими додатками, що оперують запитами до серверу, їх кількістю за одиницю часу. Так чи інакше, в даному типі тестів заміри часу виконання кожного з запитів виходять на перший план: як правило, для оптимізації роботи веб-сервісу потрібні не лише дані про час виконання певного запиту до серверу, а й інформації про те, як збільшення запитів одного типу впливає на обробку інших. Уявімо собі торгову інтернет-платформу напередодні святкових днів: запити до каталогів товарів можуть вплинути на швидкість оброблення запитів щодо оформлення замовлень, що нервує та відштовхує реальних клієнтів.

Іншим типом тестування навантаження є об’ємне тестування, коли заміри швидкості роботи проводяться на різній кількості даних. Тут ключовими моментами є проектування запитів до БД для створення тестових даних та ітеративне повторення тестів. При цьому, інженеру потрібно передбачити як консистентність(несуперечливість) даних, так і їх унікальність.

1.2.5 Спільні аспекти автоматизації тестування

Типи автоматизації тестування, перелічені в пунктах 1.2.1-4 потребують від інженера вирішення спільних технічних проблем:

- пріоретизації тестів та встановлення залежностей між ними;
- фільтрація тестів за розділами, групами, вибірковий старт;
- авторизація: отримання токену доступу, передача його між тестами, використання даних користувача в зашифрованому вигляді (логіни та паролі не повинні бути показані у коді, логах, звітах) [13];
- параметризація тестів: зберігання вхідних та очікуваних даних тестів в доступному та зрозумілому форматі, відстеження змін в даних, можливість перезапустити тест з різними вхідними даними;
- контроль оточення: розроблювані тести повинні приймати до уваги різні ОС, браузери, версії програмного продукту;

- звітність: час, крок сценарію, використані дані, запис екрану або відповіді серверу, лог серверу;
- робота з тестовими файлами: зберігання та завантаження артефактів;
- оновлення версій використовуваних бібліотек: найчастіше, заміна лише однієї бібліотеки потребує оновлення цілого ряду залежностей;
- безперервна інтеграція коду: програмний код авто-тестів повинен розвиватися разом з продуктом, що тестується. На різних версіях продукту повинні проходити відповідні версії авто-тестів;
- старт за розкладом або за вимогою: системні тести викликають обробку запитів у веб-сервісу та можуть тривати години. Необхідна можливість запланувати старт з певними параметрами оточення для ефективного регулярного отримання результатів.

1.2.6 Аналіз існуючих систем автоматизації

На даний момент деякі з вище озвучених аспектів мають безліч програмних рішень (як платних корпоративних, так і бібліотек з відкритим програмним кодом). Інші досі реалізуються індивідуально на кожному проекті, не маючи однозначного підходу. В будь-якому випадку, кожен існуючий на ринку автоматизації інструмент має доволі вузьку спеціалізацію та вирішує лише деякі з наведених вище питань.

Під час пошуку оптимального рішення було проаналізовано та протестовано більше 20ти IDE, фреймворків та бібліотек з автоматизації тестування. Нижче наведені найвідоміші та потужні рішення, які можуть претендувати на звання комплексного рішення для системної автоматизації веб-сервісів (інші варіанти були опущені, так як по суті являють собою лише бібліотеки вузької спеціалізації):

- **Serenity BDD:** фреймворк для приймального функціонального тестування веб-додатків через API браузерів. Є платним професійним рішенням. Має докладні звіти. Процес параметризації тестів та повторне

використання коду тестів з різними вхідними даними є складним, непрозорим та в значній мірі обмеженим процесом.

- **Robot Framework:** загальний фреймворк для автоматизації тестування, реалізований на Python. Потребує встановлення додаткових бібліотек для тестування web. Механізми параметризації, звітності також потребують окремої реалізації. Вимагає інсталяції на робочу станцію.
- **Katalon Studio**: інтегроване середовище (IDE) для Web, API, mobile тестування. Має плагіни для роботи в системах безперервної інтеграції. Має безкоштовну версію з обмеженими можливостями та відсутністю підтримки. Повна версія коштує $69 за місяць використання. Задає жорсткі рамки формату тестових сценаріїв.
- **SOAP UI** – інтегроване середовище (IDE) для тестування API. Має безкоштовну версію з обмеженими можливостями. Підходить для функціонального тестування, тестування безпеки та тестування навантаження. Не має власного рекордера тестів, механізму параметризації. Звіти недосконалі.
- **Postman** – IDE для розробки (в першу чергу) та тестування API. Фактично не є інструментом тестування. Потребує інсталяції. Має всі необхідні інструменти для організації запитів в колекції та їх виконання на різних оточеннях, але механізм параметризації та звітності недосконалий, що не є проблемою для розробки API, але стає критичним для інженерів із забезпечення якості.

Порівняння наведених рішень знаходиться в таблиці 1.3. Знак «+» відповідає критеріям, що реалізовані в самому рішенні та не потребують доробки або застосування сторонніх бібліотек.

Таблиця 1.3 – Порівняння поширених фреймворків автоматизації тестування веб-сервісів та REST API

Критерій / Фреймворк	Serenity BDD	Robot Framework	Katalon Studio	SOAP UI	Postman
Налаштування та ліцензія					
Потребує інсталяції	-	+	+	+	+
Відкритий програмний код	+	+	-	+	+
Тестування API					
Перевірка схеми відповіді	-	+	-	+	+
Тестування навантаження	-	-	-	+	+
Навігація по графу відповіді	-	-	+	-	
Тестування WEB					
Взаємодія з елементами сторінки	+	-	+	-	-
Робота з фреймами	+	-	-	-	-
Реалізація шаблону "Об'єкт сторінки"	+	-	-	-	-
Тестування зовнішнього вигляду сторінки	-	-	-	-	-
Організація тестових наборів					
Фільтрація, приоретизація тестів	+	+	-	-	+
Параметризований рестарт тестів	-	-	-	-	+
Контроль оточення					
Вбудований механізм звітів	+	+	+	-	-
Робота з різними наборами оточення	-	+	-	+	+
Робота з тестовими файлами	+	-	-	-	+
Безперервна інтеграція коду	-	-	-	-	-

З наведеного вище аналізу можна зазначити:

- жоден сучасний фреймворк не надає комплексного рішення для автоматизації функціонального тестування веб-сервісів;
- більшість потребують інсталяції, що робить їх платформо-залежними, тобто неактуальними в певних ОС;
- деякі бібліотеки мають потужні методи перевірки відповідей або контенту сторінок, але механізми параметризації та впорядкування тестів потребують реалізації самим інженером;
- деякі інструменти не мають детальних звітів, а їх використання на системах безперервної інтеграції не передбачено.

Даний статус-кво є результатом різноманіття технологій, від мов програмування до ідеологій клієнт-серверних архітектур. Окрім цього, провідні IT компанії постійно займаються комерційним розвитком цих технологій, намагаючись зробити їх власним продуктом. На жаль, іноді це призводить до

зміни ліцензії на використання продукту та його переходу на платне використання, як, наприклад, сталося с продуктами Serenity BDD (колишній Thucydides) та Katalon Studio.

Починаючи активності з автоматизації тестування, кожній команді QA-інженерів доводиться самостійно створювати програмний проект, реалізуючи пункти перелічені в розділах 1.2.1-1.2.5. Але слід приймати до уваги, що створення системних тестів не повинно залежати від технологій програмного продукту, що тестується: адже тести лише використовуюсь інтерфейси продукту, так само як це робить користувач-людина. Тобто проект, що містить та стартує авто-тести, може бути запущений окремо та незалежно від основного програмного продукту.

Ця ідея дає змогу спроектувати універсальну платформо-незалежну технологію, в якій перелічені вище пункти уже будуть присутні та протестовані. В такому випадку, інженеру-автоматизатору замість створення процесу «з нуля» потрібно буде лише наслідувати певні класи та слідувати інструкціям, щоб отримати робочу інфраструктуру та зосередитися на розробці бізнес-логіки тестів свого проекту.

2. ПОСТАНОВКА ЗАДАЧІ

2.1 Мета та задачі

Метою даної роботи є створення інформаційної технологія автоматизації тестування API та веб-сервісів. Розроблюване рішення являє собою модульний платформо-незалежний проект.

Дана робота потребує вирішити наступні задачі:

- дослідити актуальність проблеми;
- провести аналіз існуючих рішень;
- визначити функціональні на нефункціональні вимоги до розробки;
- провести моделювання бізнес-процесів;
- розробити програмне рішення у вигляді бібліотеки з відкритим програмним кодом;
- провести апробацію, налагоджування;

Практичне значення одержаних результатів полягає у тому, що запропонована технологія вирішить проблеми інженерів-автоматизаторів, такі, як:

- покриття веб-сервісів функціональними регресійними тестами;
- організація тестових сценаріїв у групи, задання пріоритетів;
- параметризація, повторне використання коду, варіація вхідних даних;
- механізм взаємодії з елементами управління в веб-браузерах;
- управління середовищем (заміна веб-браузерів та їх версій, серверів тестування);
- механізм запитів до API та верифікації відповідей;
- управління безперервною інтеграцією через динамічні скрипти конфігурацій розгортання проекту;
- оновлення та заміна компонентів системи (бібліотек-залежностей);

Технічне завдання дипломного проекту зазначене у додатку Б.

2.2 Вибір інструментів реалізації

2.2.1 Мова програмування

Мова програмування Java є однією з найпопулярніших, посідаючи перші місця в рейтингах мережевих технологій. Низький поріг входження, безпечність, управління пам’яттю, найбільша спільнота підтримки – це лише декілька пунктів, що роблять Java сучасним інструментом розробки програмного забезпечення.

Програмні продукти на Java є платформо-незалежними. Зі спеціального програмного забезпечення повинен бути встановлений лише пакет Java Runtime Environment. Це робить її вірним вибором для починаючих інженерів-автоматизаторів, адже як правило, авто-тести повинні запускатися як на локальній машині розробника, так і на серверах безперервної інтеграції. Найчастіше, це комбінація ОС Windows – Linux або MacOS – Linux. До недоліків Java в контексті автоматизації тестових наборів можна віднести:

- надлишковість синтаксису при розробці класів (реалізація JavaBean-класу з 4 полями займає приблизно 30 строк коду, так як потрібно явно описати 4 гетери, 4 сетери та конструктори);
- відсутність підтримки функціонального програмування (можливість передачі функцій замість змінних з’явилася починаючи з версії 1.8 і хоча це вирішує ряд питань, все ще є досить обмеженою та громіздкою);
- робота з колекціями: фільтрація, перетворення списків, масивів та мап у Java – це десятки строк коду заради однієї операції, а наявність різних типів в колекції робить роботу з нею майже неможливою без втрати типів об’єктів;

Перелічені вище (та інші) недоліки вирішує Groovy [14] – об’єктно-орієнтована мова програмування для розробки програм на платформі Java. Groovy поширюється через Apache License v 2.0. Він повністю сумісний з Java, тобто з класів Groovy можна звертатися до коду на Java, і фактично є функціональним розширенням Java. Компілюється в байт-код та виконується в JVM без залежності від платформи.

Основні властивості Groovy:

- підтримка статичної та динамічної типізації;
- підтримка перевантаження операторів;
- власний синтаксис для списків і асоціативних масивів (мап);
- вбудована підтримка регулярних виразів;
- вбудована підтримка різних мов розмітки, таких як XML і HTML;
- простий для розробників Java, оскільки синтаксис Java і Groovy майже ідентичні;
- можна використовувати існуючі бібліотеки Java;
- лаконічний та потужний Механізм перевірки виразів на істинність.

Отже, Groovy включає в себе уві переваги Java, додаючи лаконічність конструкцій при розробці коду, та розширюючи функціональність мови Java.

2.2.2 Інструмент побудови проекту

При розробці любого програмного продукту потрібно вирішити ряд питань:

- завантаження та кешування бібліотек залежностей;
- компіляція класів та управління ресурсами (копіювання тестових файлів, очистка артефактів попереднього запуску);
- запуск завдань тестування, генерації звітності;
- генерація документації;
- побудова виконуваного файла (jar, war);
- публікація та управління версією (розробка нового релізу повинна надати користувачам змогу перейти до нього з часом, користуючись попередньою версією);

Для автоматизації даних процесів існують системи автоматизації побудови проекту. В випадку Java/Groovy найпопулярнішими є Maven та Gradle [15].

Gradle [16] заснований на графі завдань (task), які можуть залежати один від одного. Конфігурація Gradle задається в файлах build.gradle, що по суті являють собою Groovy-скрипти. Розробник має повний контроль над реалізацією завдань, так як Gralde використовує предметно-орієнтовану мову (DSL [21]) на основі

мови Groovy замість традиційної XML-подібної форми представлення конфігурації проекту.

Gradle було обрано для розроблюваної технології автоматизації тестування, так як він також дозволяє управляти побудовою модульних проектів: базовий build-файл перелічує модулі, що потрібно зібрати, та їх спільні завдання. А кожен модуль має свій власний файл конфігурації і по суті представляє собою незалежний артефакт, що можна зібрати, протестувати та опублікувати.

2.2.3 Управління запуском тестових методів

Кожна система автоматизації тестування повинна забезпечувати наступні можливості управління тестовим потоком:

- фільтрувати тестові сценарії за умовою: запускати сценарії певної Feature, User story, Group;
- керувати чергою авто-тестів та враховувати залежності тестів: при старті тесту перевіряти та запускати ті тести, від яких залежить даний;
- запускати кроки підготовки та очищення середовища (pre-conditions та post-conditions) незалежно від результати тесту;
- виключати з тестової сесії тести за умовою (наприклад, тести, що залежать від певного середовища);
- розмежувати логіку тесту (програмний код) та дані тесту (вхідні параметри та очікувані результати): параметризація тестів забезпечує покриття різних випадків одним тестовим методом через повторне використання коду тесту;

Дані пункти реалізуються за допомогою так званих тестових фреймворків (Test runners). Вони задають базову архітектуру, дозволяючи інженеру-автоматизатору реалізувати власну логіку кожного з пунктів перелічених вище.

Spock Framework є найліпшим рішенням для Groovy, так як використовує його динамічність: можливості делегування та трансформації коду.

Spock пропонує писати специфікації, які описують очікувані особливості (властивості, аспекти), системи, що тестується. Такі системи називають «системою за специфікацією» (system under specification, SUS) [20].

Специфікація представлена як клас Groovy, який наслідує від spock.lang.Specification. Назва специфікації повинна описувати частину системи (Feature) або операції системи (User story), описані специфікацією. У випадку end-to-end тестів системного рівня, один клас Groovy – це одна специфікація Spock, а також визначений end-to-end набір тестових методів, кожен з яких залежить від попереднього. Нижче наведені можливості Spock, що задіяні в рамках даної технології.

Методи «до» та «після» (pre-post conditions, fixture methods)

Методи pre-post conditions відповідають за налаштування та очищення середовища, в якому запускаються методи функцій. Їх прикладами є: відкриття головної сторінки програмного продукту, авторизація, підготовка файлової системи або БД. Особливістю таких методів є те, що вони виконуються незалежно від результату тесту: якщо тест не вдався та припинив виконання, необхідно передбачити перехід на початкову сторінку веб-сервісу, видалення створених сутностей тощо.

Блоки тестових фаз

Spock має вбудовану підтримку для реалізації кожної з концептуальних фаз тестового методу. Існує шість видів фаз:

- «налаштування» (setup) – перші кроки тесту для задання вхідної точки;
- «коли» (when) – дії, що повинні призвести до очікуваного стану;
- «тоді» (then) – список дій для перевірки очікуваного стану;
- «очікувано» (expect) – аналог then, коли фаза when непотрібна;
- «очищення»(cleanup) – завершальні кроки тесту, виконується після тесту незалежно від результату;
- «де» (where) – спеціальний блок, що надає параметри тесту. Буде описаний нижче;

Блоки then та expect мають важливу особливість: кожен вираз в них неявно огортається в метод Groovy assert, тобто його результат перевіряється на істинність. На рисунку 2.1 наведено приклад простого тесту на Groovy + Spock, де в стек додається один елемент та перевіряється стан стеку. На рисунку 2.2 наведено приклад помилки, коли одна з умов була змінена на невірну (очікуваний розмір стеку «два», замість «один»).

```
when:
stack.push('I am an element')

then:
!stack.empty
stack.size() == 1
stack.peek() == 'I am an element'
```

Рисунок 2.1 – Приклад тесту на Groovy + Spock

```
Condition not satisfied:

stack.size() == 2
|     |      |
|     1      false
['I am an element']
```

Рисунок 2.2 – Приклад виводу інформації про помилку при використанні Groovy power assertion

Параметризація тестів

Spock надає можливість повторювати тестовий сценарій один, два, або безліч разів завдяки блоку where. Цей блок йде останнім в тестовому методі та повинен надати масив даних для повторення тесту. При кожній ітерації на вхід тестовому методу подається наступний блок параметрів.

Більшість фреймворків пропонують описувати дані зразу після тестового методу. Нажаль, усі приклади роботи параметризації, незалежно від тестового фреймворку, є примітивними та націленими на unit-тести. Без реалізації власного рішення вони слабо підходять для функціональних тестів системного рівня (проблема існує як для WEB UI, так і для API тестувань). Причиною тому є

кількість вхідних параметрів тестового сценарію. Представимо тест системного рівня, що завантажує геометричну модель, задає її параметри, стартує експеримент та перевіряє отримані дані симуляцій. Подібні тести можуть приймати десятки параметрів та повторюватися кілька разів. А дані для тесту можуть формуватися за межами проекту автоматизації (братися з тестових планів або інших джерел). В такому випадку зберігання даних зразу в коді тестового метода стає неможливим через неймовірне зростання об'єму класу та незручності в підтримки та оновленні тестових даних.

Механізм параметризації з використанням блока where, але з відокремленим сховищем тестових даних та автозв'язуванням тестових методів буде описаний у розділах 3 та 4. На даний момент запропонований підхід параметризації не має аналогів у доступних бібліотеках автоматизації тестування.

2.2.4 Управління WEB-браузерами

Selenium – це проект, в рамках якого розробляється серія програмних продуктів з відкритим вихідним кодом (open source) для автоматизованого управління web-браузерами. Головним розділом проекту є Selenium Webdriver.

Найчастіше, під Selenium Webdriver розуміють:

- специфікацію програмного інтерфейсу для управління браузером;
- референтні реалізації цього інтерфейсу для декількох браузерів;
- набір клієнтських бібліотек для цього інтерфейсу на декількох мовах програмування [22];

Selenium Webdriver – є драйвером web-браузера в буквальному сенсі. Це програмна бібліотека, що не має графічного інтерфейсу користувача, та дозволяє іншим програмам взаємодіяти з браузером: отримувати від браузера дані і змушувати браузер виконувати певні команди.

Найголовніша відмінність Selenium Webdriver від всіх інших драйверів полягає в тому, що це «стандартний» драйвер, а всі інші – «нестандартні». І це не проста фігура мови. Організація W3C дійсно прийняла Webdriver за основу при розробці стандарту інтерфейсу для управління браузером [23].

Selenium Webdriver сам по собі не є інструментом тестування. Це бібліотека низького рівня абстракції, що формалізує стандарт управління браузером, тобто визначає мінімальний набір команд, який повинен бути реалізований в кожному браузері та підтримуватися від версії до версії. Він надає авто-тестам доступ до браузеру і на цьому його функції закінчуються.

Як і любий драйвер, з однієї сторони Selenium Webdriver надає розробнику інтерфейс по управлінню web-браузером (кліки, ввід-зчитування тексту, положення вікна, очікування елементів та інше) на певній мові програмування. З іншої, передає команди в браузер по протоколу, зрозумілому конкретному браузеру. Наприклад, у випадку тестування через Firefox необхідно завантажити реалізацію web-драйверу Geckodriver, який буде взаємодіяти з web-браузером Firefox по протоколу Marionette. Web-драйвери регулярно оновлюються розробниками самих web-браузерів, що гарантує актуальність та стабільність команд. Взаємодія між програмним кодом тестів та браузерами за допомогою Selenium Webdriver представлена на рисунку 2.3.

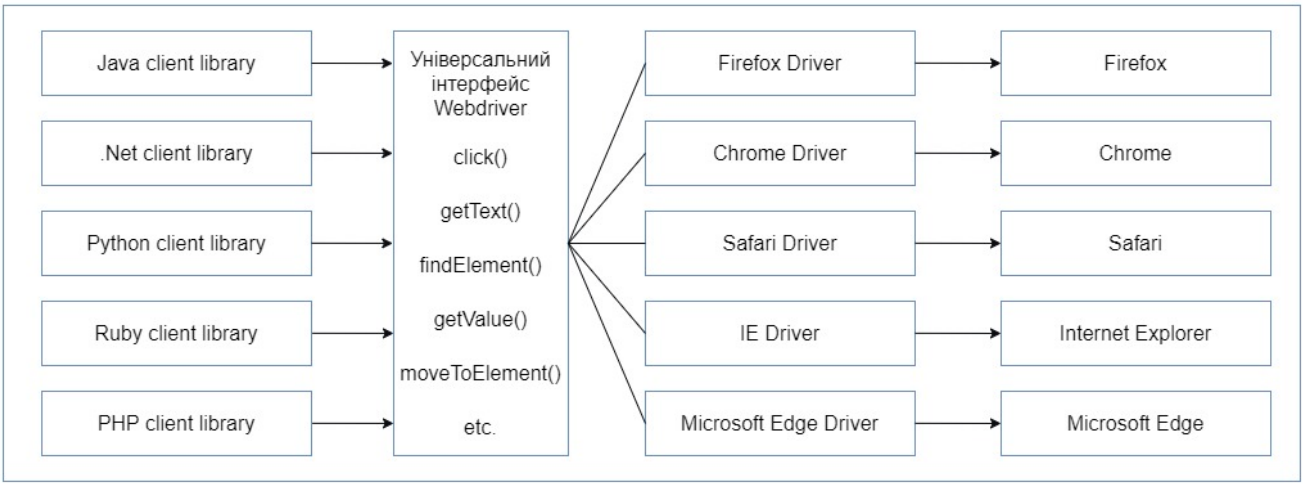

Рисунок 2.3 – Взаємодія компонентів Webdriver – Browser

Webdriver взаємодіє з додатком у вікні браузера через елементи сторінки (WebElement), які, в свою чергу, визначаються через «локатори».

Веб-елемент – це індивідуальна сутність, що генерується в HTML сторінці на основі логіки програмного продукту. Прикладами є: заголовки, кнопки, гіперпосилання, поля введення, текстові блоки. Елементи в HTML визначаються через ім’я тега (h1, div, select, frame та ін.), атрибути (name, class та ін) і вміст

(текст, значення). У них також можуть бути дочірні елементи (таблиці, де в кожній з строк є кілька комірок, випадаючий список).

Локатор веб-елемента – це об’єкт, який знаходить і повертає веб-елементи на сторінці по заданому запиту.

Selenium Webdriver надає вісім видів локаторів, кожен з яких представлений окремим програмним класом, що відображає суть локатора:

- By.ById;
- By.ByName;
- By.ByClassName;
- By.ByTagName;
- By.ByCssSelector;
- By.ByXPath;
- By.ByLinkText;
- By.ByPartialLinkText;

Слід наголосити, що для взаємодії з елементами сторінки та самим вікном браузера Selenium Webdriver використовує API самого веб-браузера, тобто для браузера, а отже і для програмного продукту, що тестується, не існує видимої різниці між діями користувача та авто-тесту. Це гарантує очікувані результати для однакових кроків, виконаних вручну та засобами автоматизованого сценарію.

Окремою властивістю Selenium Webdriver є можливість робити знімки сторінки та зберігати її стан (HTML, скрипти, стилі), що робить можливим реалізацію регресійного тестування інтерфейсу користувача: знімки сторінок, записані у режимі еталонного запуску, можуть бути порівняні з актуальним виглядом сторінок під час тестових сесій.

Оскільки при оновленні веб-браузеру виходить новий реліз веб-драйверу, необхідно передбачити його завантаження перед тестуванням. Це стає проблемою у випадку підтримки різних браузерів, їх версій та ОС. Вирішити дане питання на локальній робочій станції можна за допомогою бібліотеки з відкритим програмним кодом Webdriver Manager [24]. Вона читає з конфігурації версії веб-драйверів, які потрібно завантажити, та звертається до публічних сховищ

бінарних файлів для завантаження необхідних веб-драйверів на основі ОС та заданих версій.

Для автоматизації вибору веб-драйверу на віддалених серверах існує комплексне рішення Selenoid, яке буде розглянуте нижче.

Бібліотека GEB [25] – сучасний інструмент, написаний мовою Groovy, що аналогічно Spock використовує динамізм та лаконічність Groovy для визначення об’єктів сторінок, елементів, взаємодії з ними. Найбільш важливими розширеннями функцій Selenium в GEB є:

- реалізація об’єктів типу Module – окремих функціональних компонентів для повторного використання на різних сторінках. Приклад – меню навігації, фільтри, таблиці, що присутні в різних частинах UI програмного продукту;
- реалізація об’єктів типу Page – структур, що визначають цілу сторінку веб-сервісу та задають правила навігації, перевірок наявності, операцій на сторінці;
- Обгортка Navigator для веб-елементів. Вона значно розширює можливості пошуку, наслідування, фільтрації елементів.

2.2.5 Засоби тестування REST API-сервісів

Суть тестування та перевірки сервісів REST-API наступна: клієнт надсилає HTTP-запит до кінцевої точки API і перевіряє очікувані значення відповіді, а також схему відповіді, заголовки тощо.

Головною проблемою є навігація по об’єкту відповіді для поступу до потрібних полів, адже вони можуть бути вкладені, знаходитися всередині списків, тощо.

Отримуючи відповідь у вигляді JSON або XML, розробнику тестів необхідно перетворити його у Java POJO (старі добрі об’єкти Java). Далі з цією структурою можна працювати як зі звичайним об’єктом. Як результат, інженер-автоматизатор отримує повноцінний об’єктно-орієнтований підхід для перевірки відповідності відповіді API описаній структурі об’єкта.

Http-клієнт REST Assured являє собою інструмент для відправки та отримання даних по REST API. Будучи написаним на Groovy, він розширює можливості java, пропонуючи писати авто-тести у Behavior-Driven-Development (BDD) форматі «given-when-then» [27].

На рисунках 2.4-2.5 наведено приклади типової відповіді API у форматі JSON та авто-тесту, що перевіряє дану відповідь. Тест написаний з дотриманням правил, зазначених вище.


```
{
"lotto":{
      "lottoId":5,
      "winning-numbers":[2,45,34,23,7,5,3],
      "winners":[
         {
            "winnerId":23,
            "numbers":[2,45,34,23,3,5]
         },
         {
            "winnerId":54,
            "numbers":[52,3,12,11,18,22]
         }
      ]
   }
}
```


Рисунок 2.4 – приклад API-відповіді у форматі JSON

```
@Test public void
lotto_resource_returns_200_with_expected_id_and_winners() {

    when().
            get("/lotto/{id}", 5).
    then().
            statusCode(200).
            body("lotto.lottoId", equalTo(5),
                 "lotto.winners.winnerId", hasItems(23, 54));
}
```

Рисунок 2.5 – API-тест у форматі «given-when-then»

Як видно з прикладу, в секції when тест робить запит до конкретного методу API, а далі в секції then перевіряє код статусу відповіді та значення конкретних полів об’єкту відповіді. Слід зауважити, наскільки явно відбувається навігація про графу відповіді: для отримання поля winnerId розробник тесту вказує шлях до нього через множину батьківських об’єктів, розділяючи їх крапкою.

Тести API також необхідно організовувати у черги та набори. Слід зауважити, що механізм параметризації та фільтрації тестів буде універсальним для усіх модулей в рамках розроблюваної технології.

2.2.6 Засоби тестування баз даних

Робота в базою даних може буди як окремим етапом тестування програмного продукту (перевірка зв'язків, обмежень, цілісності даних), так і допоміжними активностями при системному тестуванні (наповнення баз тестовими даними, об'ємні тести). У будь-якому випадку необхідно вирішувати наступні питання:

- створення бази та її заповнення даними;
- формування запитів, підключення до БД;
- порівняння результату, таблиць;
- чистка бази після виконання тестів.

Бібліотека DbUnit [28] дозволяє автоматизувати ці дії, надаючи набор типів та методів для роботи з базами (методи операцій з таблицями, такі як запис, читання, видалення, та типи для зберігання даних з БД у вигляді Java-об'єктів). Використовуючи DbUnit ми можемо привести сценарії взаємодії в базою даних до формату окремих тестових методів, яким модна задати залежності, черги та параметри за допомогою механізмів Spock, описаних вище.

2.2.7 Вихідні тестові артефакти, звітність

У випадку системних тестів web-сервісів, звіт невдалого тесту повинен зберігати вхідні параметри, деталі запиту / дії в UI, очікувані дані та суть розбіжності, середовище (номер ревізії, сервер, час и т.д).

Для функціональних тестів WEB UI розроблювана технологія надає:

- знімок сторінки веб-браузера;
- відео-запис тестової ітерації з моменту старту до моменту падіння;
- html-код сторінки, де відбулася помилка;

- перелік кроків тесту з реально використаними даними (назви кнопок, введення тексту);
- лог помилок на стороні клієнту та серверу;

Це дає інженеру вичерпну інформацію для аналізу та відтворення помилки. Для відображення деталей у кінцевому форматі використаємо Allure Report Tool [29] – рішення з відкритим кодом від Yandex. Allure має методи для прикріплення потрібних файлів до тестової ітерації та відображення їх у фінальному звіті. Він також дозволяє сортувати тести по розділам, групам.

2.2.8 Безперервна інтеграція та запуск тестувань

Системи безперервної інтеграції (Continuous integration, CI) – сервіси, ціль яких – запускати певні завдання згідно заданої конфігурації. Найвідомішим сервісом є Jenkins [30], що має безліч плагінів та одну з найбільших ком'юніті користувачів. А прикладами завдань для CI можуть бути: побудова нової версії програмного продукту після злиття коду, старт юніт-, або функціональних тестів за розкладом або за вимогою. Такий процес дає змогу отримувати інформацію про поточний стан програмного продукту якнайшвидше.

Головним аспектом CI є формування самої конфігурації зборки: визначення етапів, потрібних для побудови програмного продукту, проекту тестів, завантаження потрібних артефактів, старт самих тестів та публікація результатів. Найсучаснішим підходом створення конфігурацій є так звані магістральні скрипти (Pipeline Scripts). Їх основні переваги:

- мають потужний DSL для описання етапів (Stage), кроків (Step), параметрів середовища та локації зборки, що дає розробнику всю міць мов програмування;
- зберігаються в окремому файлі замість розподілених в плагінах налаштувань, а отже не втрачаються при зміні CI-серверу;
- зберігаються в репозиторії проекту, тобто при зміні самого проекту розробник легко адаптує скрипт CI зборки.

Jenkins Pipeline також надає можливість конфігурувати багатогалузеві (Multibranch pipeline) зборки [31]. Як правило, при розробці програмного продукту актуальними є декілька гілок системи контроля версій. Для системи контроля версій GIT це develop/master, candidate, feature. Команді QA важливо розуміти стан програмного продукту в кожній з них. Multibranch pipeline дозволяє будувати та вести історію зборок окремо по кожній з гілок, що є безперечною перевагою для процесу забезпечення якості: інженер має змогу аналізувати стабільність окремих тестів в різних випробуваннях по кожній з ревізій, що тестуються.

2.2.9 Супровідна документація

Успіх технології також залежить від того, наскільки зрозумілою та структурованою вона є. Зберігання та – головне – актуалізація документації є не менш важливим аспектом, ніж працездатність програмного коду. Найбільш успішні продукти зберігають документацію поряд з самим програмним кодом. Такі інструменти, як AsciiDoctor [32] дозволяють поєднувати опис функцій системи з прикладами коду та посиланнями на певні строки коду.

AsciiDoctor – це текстовий процесор з відкритим вихідним кодом та інструментарій публікацій для перетворення формату AsciiDoc в HTML5, DocBook, PDF та інші формати. Перевагою такого підходу є те, що документація розроблюється, тестується та оновлюється паралельно з самим проектом. При доданні чи зміні функцій розробник оновлює документацію та нова її версія належить до нового релізу, в той час як попередній реліз зберігає актуальну для себе версію. В даній технології інструкція користувача буде розроблена засобами AsciiDoctor, а процес її публікації до репозиторію в форматах HTML5 та PDF буде автоматизований засобами системи зборки проекту Gradle.

3 ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

3.1 Збір вимог

Програмний каркас, або фреймворк – інфраструктура програмних рішень, що полегшує розробку складних систем. Спрощено дану інфраструктуру можна вважати своєрідною комплексною бібліотекою, але при цьому вона має ряд обмежень, що задають правила створення сутностей та написання коду. Розроблювана технологія представлятиме собою таку комплексну платформо-незалежну бібліотеку з відкритим кодом для забезпечення реалізації та виконання автоматизованих сценаріїв для тестування веб-сервісів через графічний інтерфейс користувача, REST API, або базу даних. Рішення повинно відповідати ітеративним методологіям розробки програмного забезпечення та концепції GIT-flow: надавати можливість розробки авто-тестів та їх запуску на різних етапах життєвого циклу програмного продукту, що тестується, та на різних його версіях.

В розроблюваній технології можна виділити наступні області застосування:

- розробка коду авто-тестів інженерами-автоматизаторами:
    - управління залежностями тестових методів;
    - доступ до параметрів тестів, їх зміна залежно від середовища;
    - набір класів-помічників по роботі з датами у різних форматах, файлами ресурсів та ін.;
    - пре- та після- дії (підготовка середовища незалежно від результату тесту);
- параметризація старту автоматизованих сценаріїв:
    - збирання тестового артефакту з певної ревізії чи тегу VCS;
    - задання черг тестування, вибір тестових наборів;
    - старт за умовами, за розкладом, за вимогою;
    - задання параметрів тестового середовища (тестовий сервер, браузер, розміри вікна, назва бази даних);
- публікація та зберігання тестових результатів:

- однозначний опис помилки у форматі «очікувано – отримано»
- супровідні артефакти (відеозапис екрану, логи браузера, сервера, параметри тесту, стек-трейс помилки, кроки тесту);

Групи цільової аудиторії розроблюваної технології:

- розробник програмного продукту (DEV);
- інженер із забезпечення якості програмного продукту (QA);
- керівник проекту (PM)

3.2 Структурно-функціональне моделювання

3.2.1 Функціональна модель системи у IDEF0-нотації

Функціональна модель представляє систему функцій з необхідним ступенем деталізації, які в свою чергу відображають свої взаємовідносини через об’єкти системи. Вона являє собою ієрархію взаємопов’язаних діаграм, кожна з яких представляє підсистему або її окрему компоненту. Вершина цієї структури містить загальний опис системи, який деталізується на наступних рівнях декомпозиції.

IDEF0 [33] – методологія функціонального моделювання (англ. Function modeling) і графічна нотація, призначена для формалізації і опису бізнес-процесів. Відмінною особливістю IDEF0 є її акцент на підпорядкованість об’єктів. В IDEF0 розглядаються логічні відносини між роботами, а не їх часова послідовність (потік робіт). Контекстна діаграма розроблюваної технології представлена на рисунку 3.1 Діаграма декомпозиції являє собою детальний опис етапів всередині системи від початку циклу до його завершення. Етап зображується прямокутником, всередині якого подається назва у формі дієслова. Стрілками показані вхідні артефакти, правила, інструменти виконання та вихідні артефакти. Усі вони, у свою чергу, можуть бути вхідними для іншого блоку. Діаграма декомпозиції першого (A0) та другого рівнів розроблюваної технології показана на рисунках 3.2-3.7.

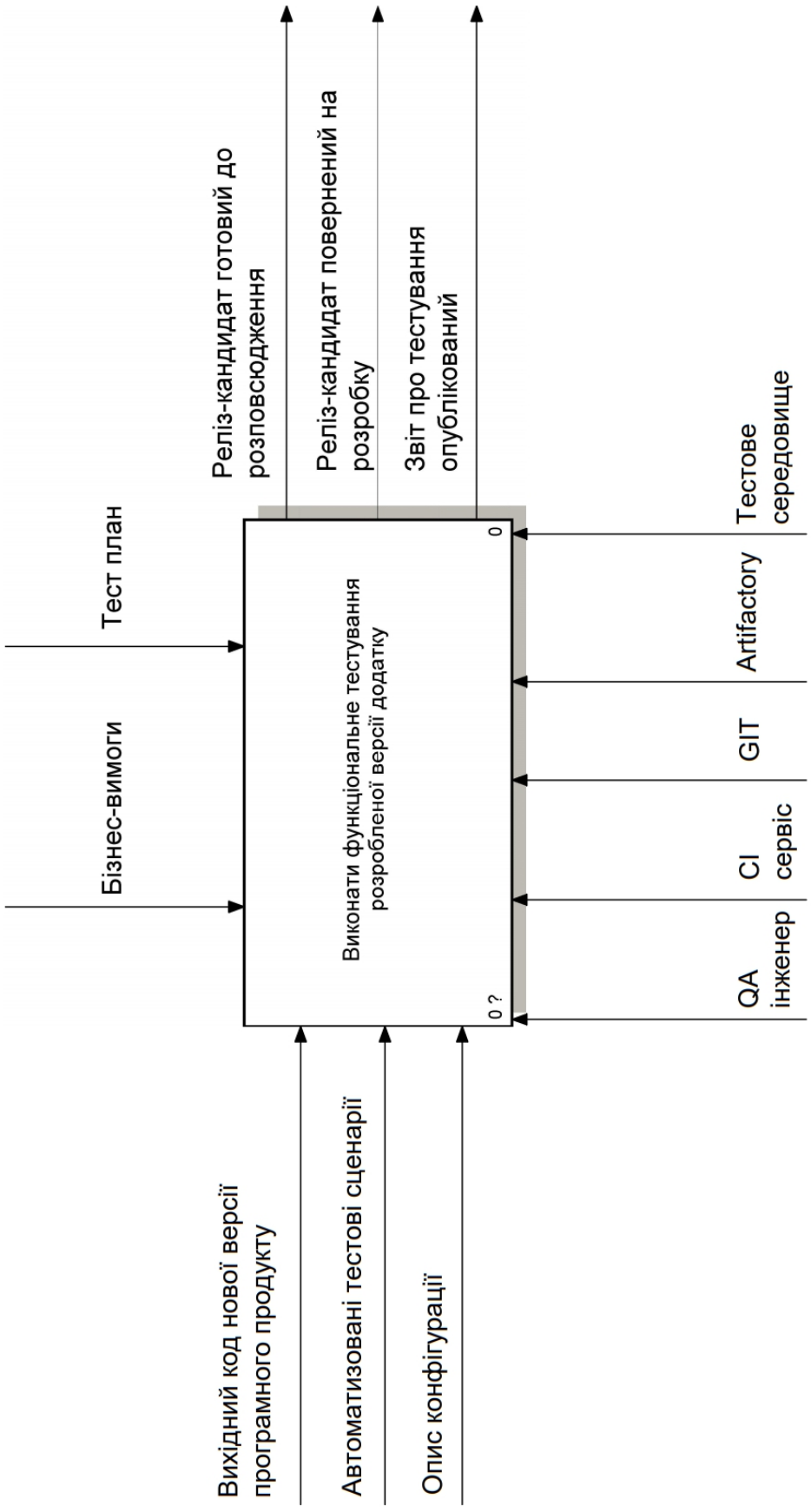

Рисунок 3.1 – Контекстна діаграма

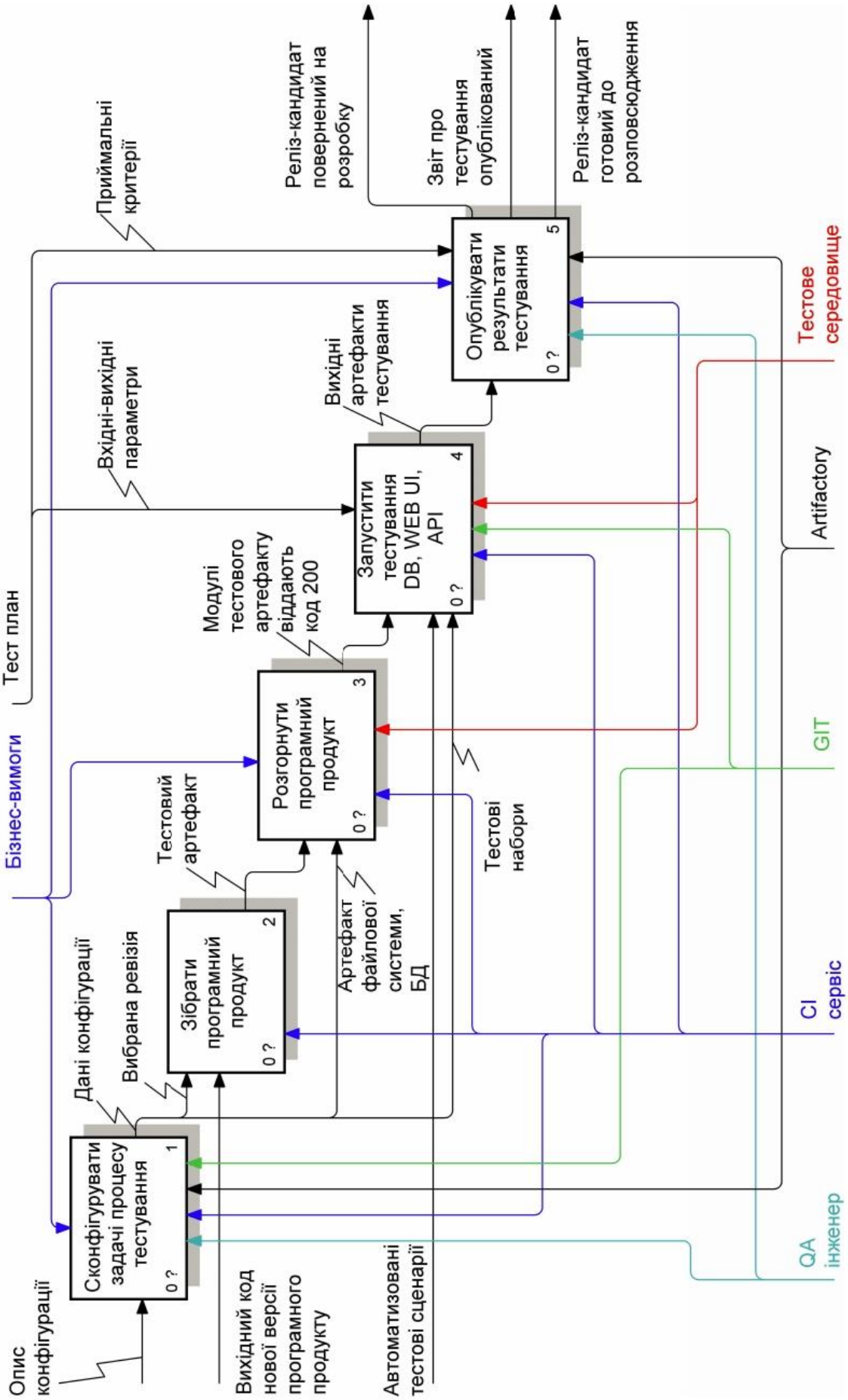

Рисунок 3.2 – Декомпозиція першого рівня (A0)

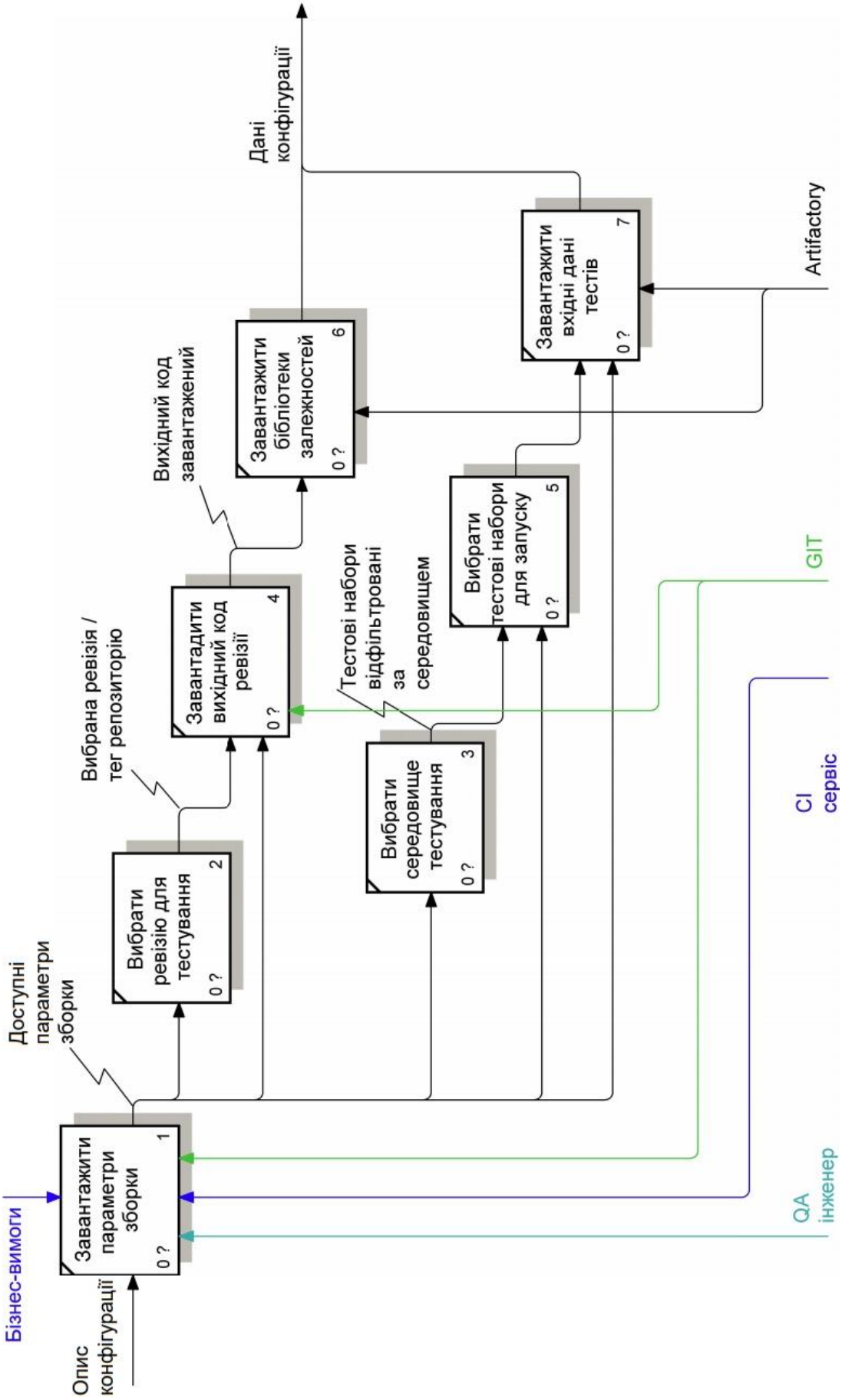

Рисунок 3.3 – Декомпозиція другого рівня, блок A0 - 1

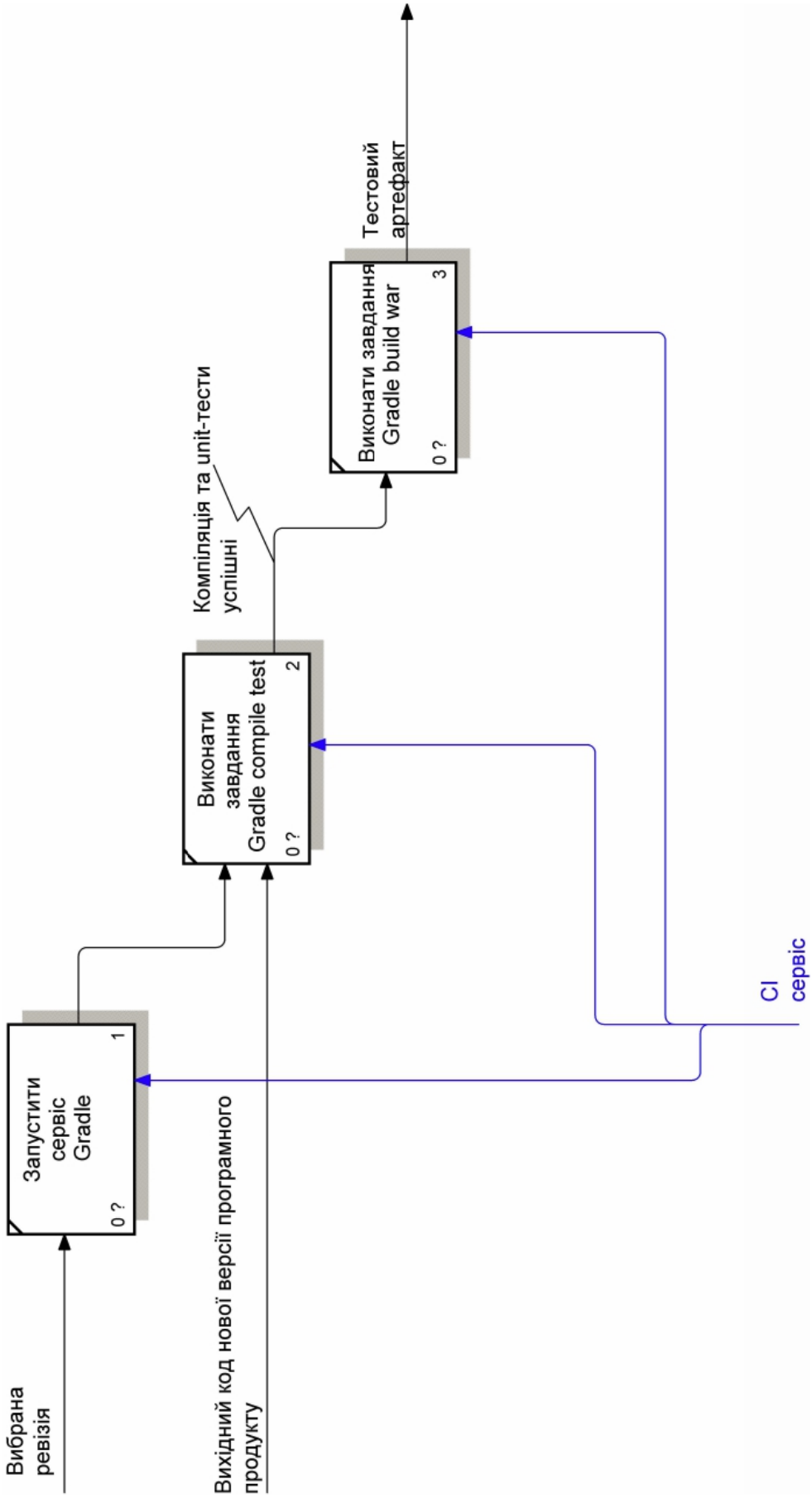

Рисунок 3.4 – Декомпозиція другого рівня, блок A0 - 2

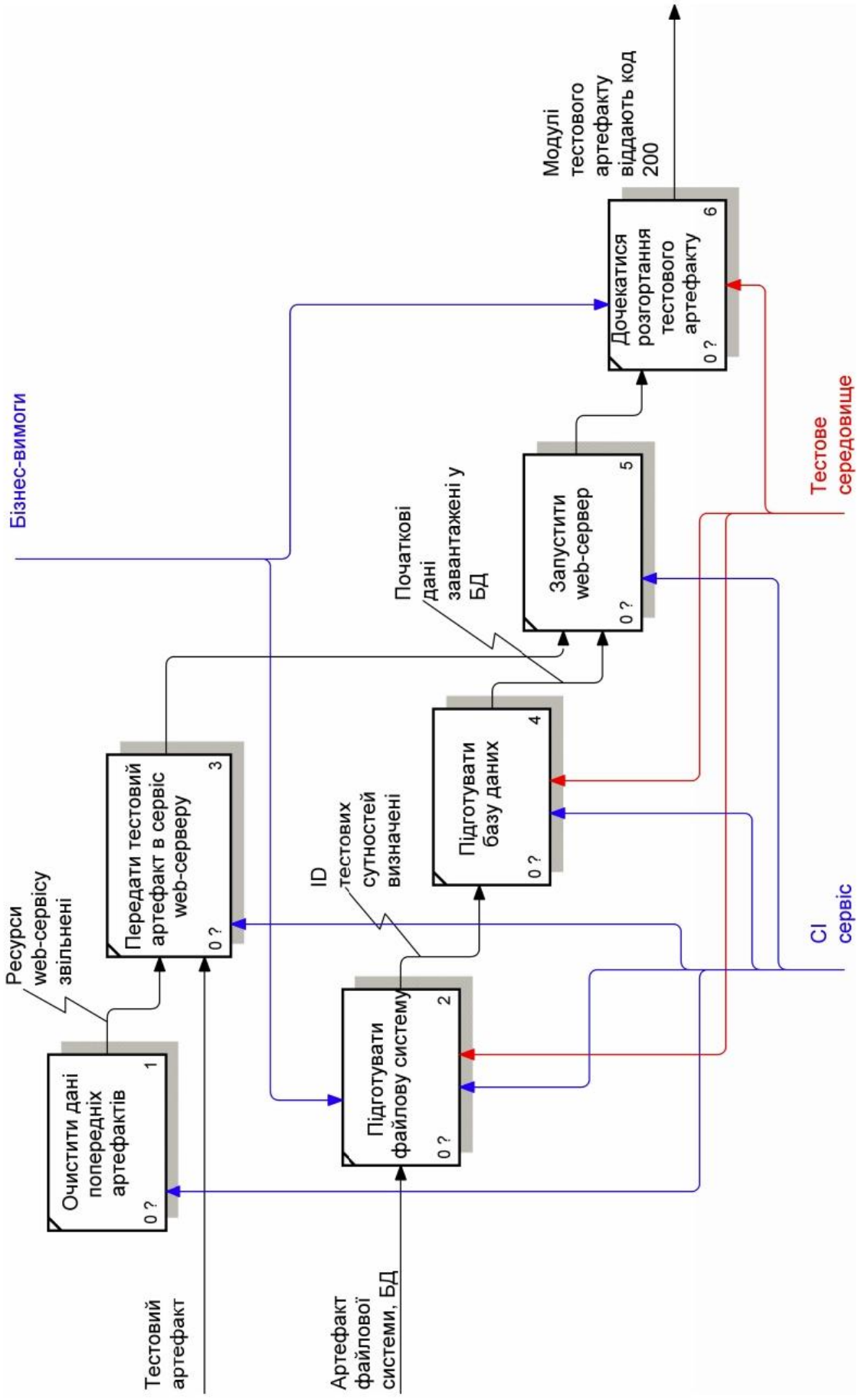

Рисунок 3.5 – Декомпозиція другого рівня, блок A0 - 3

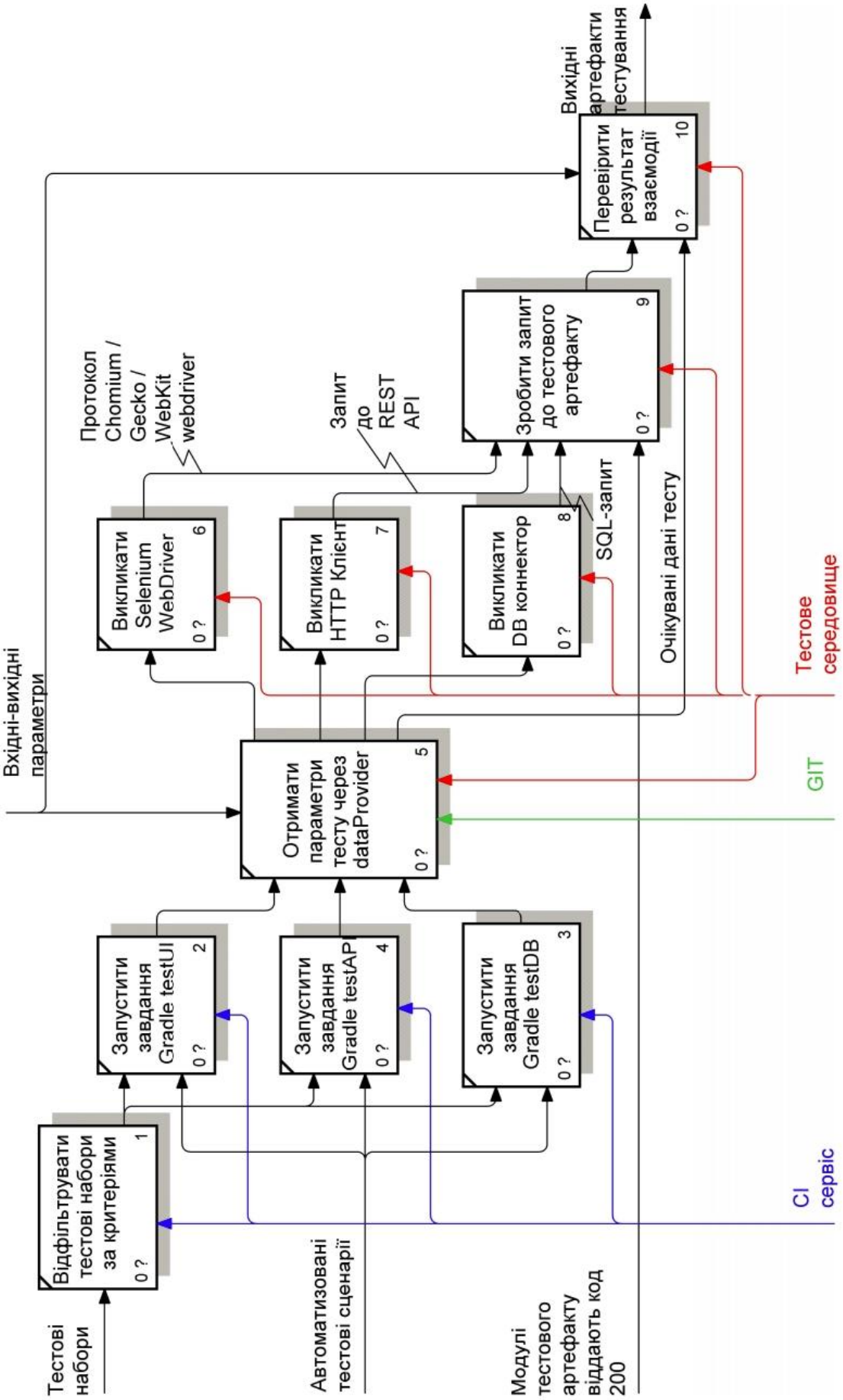

Рисунок 3.6 – Декомпозиція другого рівня, блок A0 - 4

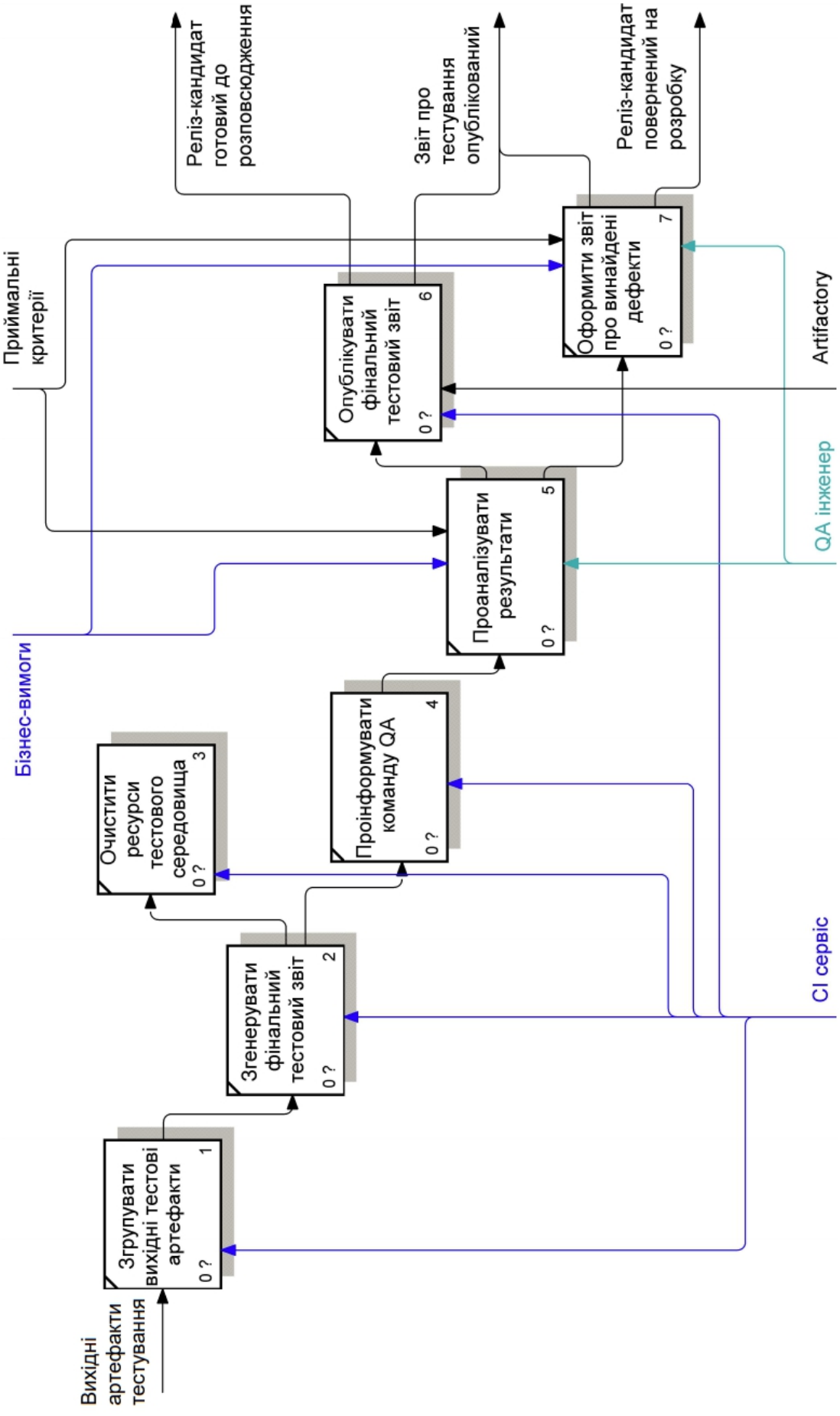

Рисунок 3.7 – Декомпозиція другого рівня, блок A0 - 5

3.2.2 Варіанти використання розроблюваної технології

Варіанти використання (Use cases) – це опис дій, які може здійснювати система у відповідь на зовнішні дії користувачів або інших програмних систем. Вони дозволяють точніше уявити, як повинна працювати система.

Варіанти використання відображають функціональність системи. Діаграми варіантів використання описують функціональне призначення системи або те, що система повинна робити.

Візуальне відображення варіантів використання розроблюваної технології показане на рисунку 3.8.

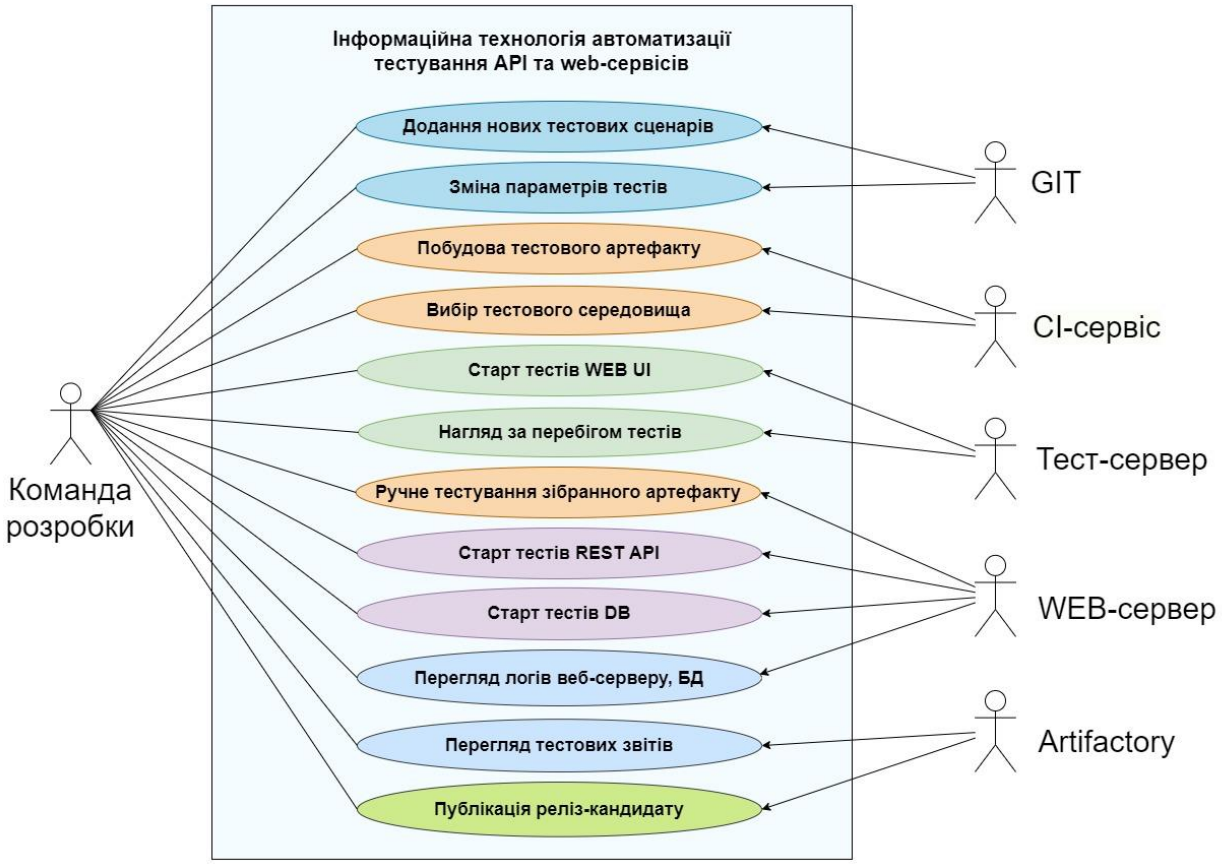

Рисунок 3.8 – Варіанти використання

3.2.3 Діаграма взаємодії

Діаграма взаємодії – одна з моделей опису поведінки взаємодіючих груп об’єктів в UML. Як правило, кожна окрема діаграма взаємодії описує поведінку тільки в межах одного варіанта використання. На такій діаграмі прийнято відображати екземпляри об’єктів та повідомлення, якими ці об’єкти обмінюються один з одним в рамках даного варіанта використання. Діаграма взаємодії допомагає візуалізувати процеси в системі з урахуванням часу та послідовності виконання дій. Діаграма взаємодії акторів та компонентів розроблюваної технології показана на рисунку 3.9.

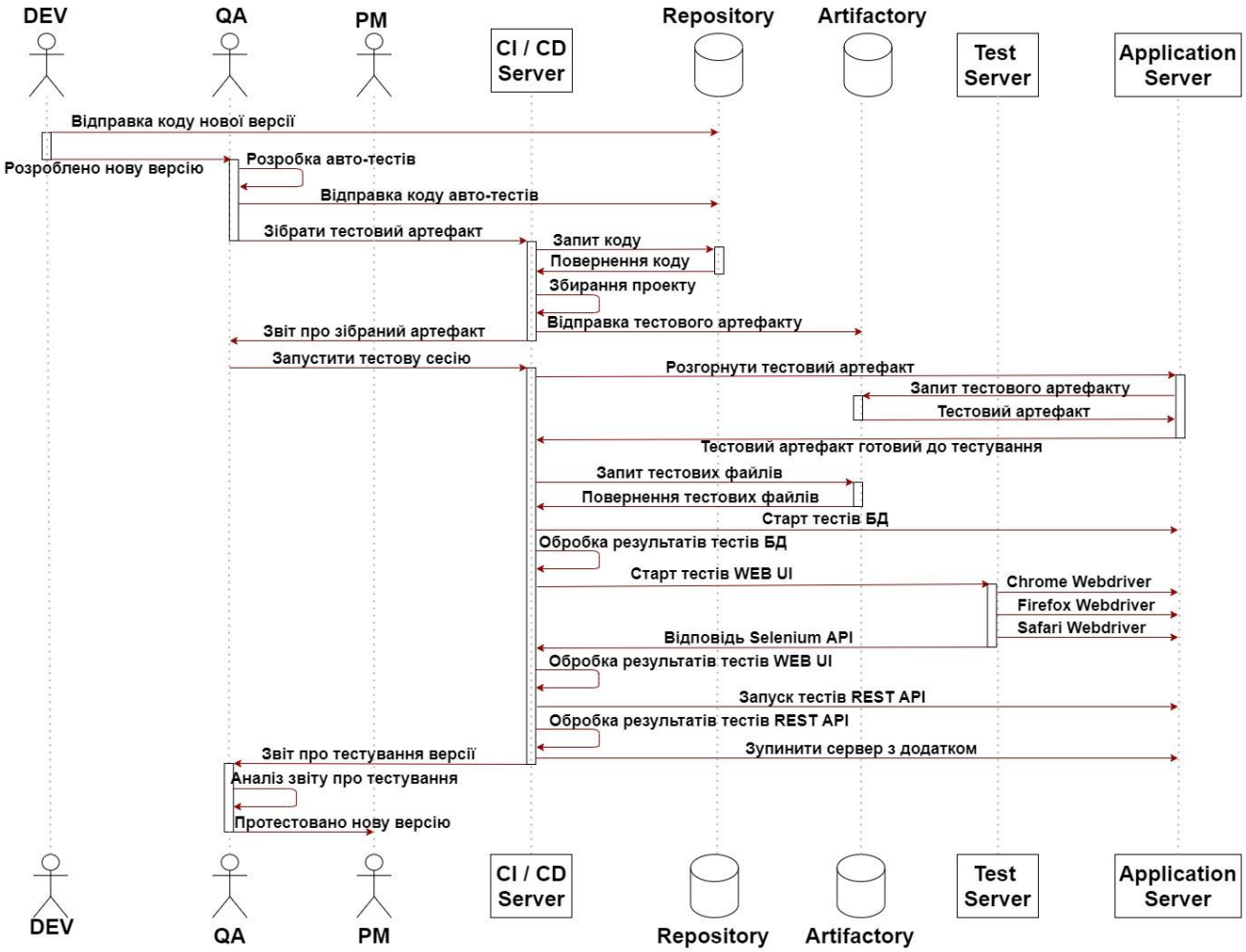

Рисунок 3.9 – Діаграма взаємодії

3.3 Проектування структури розроблюваної технології

Розроблювана технологія включає в себе кілька рівнів, кожен з яких надає необхідну абстракцію для зменшення залежностей та керування потоками даних під час тестової сесії.

3.3.1 Рівень абстракції, тестових сценаріїв.

Включає в себе тестові класи, що згруповані за бізнес логікою або типом тестування. На даному рівні відбувається фільтрація, пріоретизація тестових наборів, а також постачання параметрів в тестові методи. Самі тестові методи оперують поняттями лише бізнес-логіки, а не реалізації програмного продукту. Беручи за приклад тести WEB UI, код тестового методу нагадує історію користувача (User story). Тобто тест описує, хто працює з додатком, які дії виконує та що очікує. Але в тесті не йдеться про пошук елементів сторінки, їх очікування та обробки виключень. Це дає змогу зіставляти код тестів зі специфікацією продукту та вносити зміни в поведінку тестів відповідно до змін в бізнес-логікі продукту. Досягається дана абстракція за допомогою поєднання шаблонів «Об’єкт сторінки» (Page object) та «Компонент сторінки» (Component object).

3.3.2 Рівень інкапсуляції тестових дій.

Суть полягає у тому, що замість взаємодії з елементами веб-сервісу прямо у тілі тестового методу, усі необхідні елементи та операції над ними приховуються усередину окремого класу, що описує одну сторінку, або її частину. Далі методи такого об’єкту можна викликати з тесту, або використати даний об’єкт як частину іншого класу, що описує іншу сторінку. Приведемо приклад: панель навігації, найчастіше, присутня на майже усіх сторінках веб-сервісу, отже це кандидат в компонент сторінки, який можна описати окремим класом та ініціалізувати у батьківському класі декількох сторінок. Описавши локатори лише потрібних його елементів та необхідні дії (відкриття меню за назвою, перехід по пунктам меню), ми отримуємо функціональний об’єкт з відкритим інтерфейсом для використання у тестових методах. Метод «перейти в меню за назвою» може бути використаним

у сотнях тестових сценаріїв, але коли розробники змінять пункти меню, його положення чи поведінку, адаптацію потрібно буде провести лише в тілі одного методу одного класу. Це дає надзвичайні переваги та економію часу при актуалізації тестових сценаріїв після оновлення поведінки програмного продукту.

3.3.3 Рівень взаємодії з веб-браузером.

Даний рівень використовується лише в тестах типу WEB UI, так як потребує взаємодії з графічним інтерфейсом веб-сервісу та передачі команд за протоколами веб-браузерів. Сюди надходять виклики API клієнтської бібліотеки Selenium з рівня реалізації (від компонентів сторінок). Варіанти даної бібліотеки підтримуються для декількох найпопулярніших мов програмування, що дає змогу розробляти авто-тести без прив’язки до конкретної технології. Незалежно від мови програмування, кожна клієнтська бібліотека Selenium реалізує загальний інтерфейс WebDriver, де представлені стандартизовані методи управління веб-браузером. Для задання параметрів самого веб-браузеру (назва та версія, розміри вікна, налаштування дозволів та інше) тести передають в Selenium об’єкт типу «можливості» (Capabilities).

З іншої сторони бібліотеки Selenium команди від тестів та компонентів сторінок трансформуються в формат, зрозумілий конкретному веб-браузеру (команди передаються по відповідному протоколу). Саме цьому кожен браузер та кожна його версія потребують свій веб-драйвер для трансляцій команд, адже програмний код самого веб-браузеру змінюється від версії до версії. Веб-драйвер є двійковим платформо-залежним файлом, а отже потрібен механізм для автоматичного завантаження веб-драйверів залежно від конфігурації тестової сесії.

Для вирішення питань налаштування тестового середовища згідно вимог Capabilities, завантаження веб-драйверів, доступу до вікна браузеру під час тестування використовується інструмент Selenoid. Він надає широкий асортимент докер-контейнерів, в яких присутні конкретні версії веб-браузерів та відповідні їм файли веб-драйверів. Також, Selenoid надає потужні механізми управління

параметрами браузеру та збору його логів. Після закінчення тестової сесії контейнер зупиняється, що наряду з безпекою (усі результати взаємодії з веб-сервісами залишаються в файловій системі контейнеру) є ефективним засобом управління ресурсами пам’яті тестового серверу.

За допомогою концепції об’єктів Capabilities можна емулювати різні фізичні пристрої, такі, як мобільні телефони, планшети, стаціонарні монітори. У поєднанні з різними типами та версіями браузерів це дає безмежні можливості з перевірки функціональності та інтерфейсу тестового артефакту.

Якщо необхідно виконати старт тестів на локальній робочій станції (для потреб розробки, зручності пошуку помилки), автоматичне завантаження потрібної версії веб-драйверу здійснюється засобами бібліотеки WebDriver Manager, що реалізує завантаження бінарних файлів веб-драйверу з відкритих репозиторіїв на основі типу операційної системи, назви та встановленої версії веб-браузеру. В обох випадках (віддалене тестове середовище, або локальний старт авто-тестів) інженеру-автоматизатору не потрібно витрачати час на пошук веб-драйверів – дана технологія автоматизує вирішення даного питання.

3.3.4 Рівень взаємодії з тестовим артефактом.

Тут відбувається взаємодія з елементами сторінок веб-сервісу засобами веб-браузера: натискання кнопок, введення значень, очікування повідомлень тощо. Слід зауважити, що на даному етапі межи між реальним користувачем та автоматизованим тестом для програмного продукту не існує: веб-драйвер оперує сторінками через ті ж методи браузеру, що й реальні пристрої типу клавіатура чи миша.

Також очевидно, що для самих автоматизованих сценаріїв та компонентів сторінок абсолютно неважливо, які технології використовуються в реалізації програмного продукту та для його розгортання: адже тести отримують доступ до веб-сервісу через вікно веб-браузеру та HTML-елементи. Як результат, даний підхід дозволяє однаково ефективно автоматизувати взаємодію з любими веб-

додатками, від соціальних мереж типу facebook до спеціалізованих інженерних рішень типу продуктів Dassault Systemes.

Що до тестування REST API та DB, на рівні абстракції тестові методи оперують такими об’єктами як Специфікація запиту (Request spec), Специфікація відповіді (Response spec) та об’єктів наборів даних (Data set). В них формуються вимоги до відправлених даних та до очікуваних результатів. На прикладі тестів типу REST API в об’єкті response spec можна зберігати очікуваний код відповіді 200 або 401, тип очікуваних даних та інші перевірки, що можна використати більш ніж в одному тесті. При цьому, якщо веб-сервіс на конкретний запит почав повертати дані у іншому форматі, достатньо внести зміни лише в одному об’єкті response spec, щоб тестові набори знову стали актуальними (аналог шаблону page object в тестах WEB UI). Далі, минаючи рівень взаємодії з веб-браузером, такі тести відправляють запити до API веб-серверу, або серверу бази даних відповідно, та, отримавши відповідь, проводять її аналіз: перевіряють структуру отриманої відповіді, або формат та значення отриманих даних.

Незалежно від типу тестування, після взаємодії з програмним продуктом дані повертаються назад на рівень тестових наборів, де тестовий фреймворк приймає рішення: тестовий крок успішний, чи необхідно згенерувати помилку з інформацією про розбіжність між очікуваними та реальними даними. Для кожного з тестових класів на цьому рівні підключаються декілька слухачів, що залежно від вимог та типу тестування прикріпляють до тестового звіту додаткові дані про стан екрану, запис виконання, параметри тесту, запит та відповідь, логи браузеру тощо. Загальна структура даної інформаційної технології показана на рисунку 3.10.

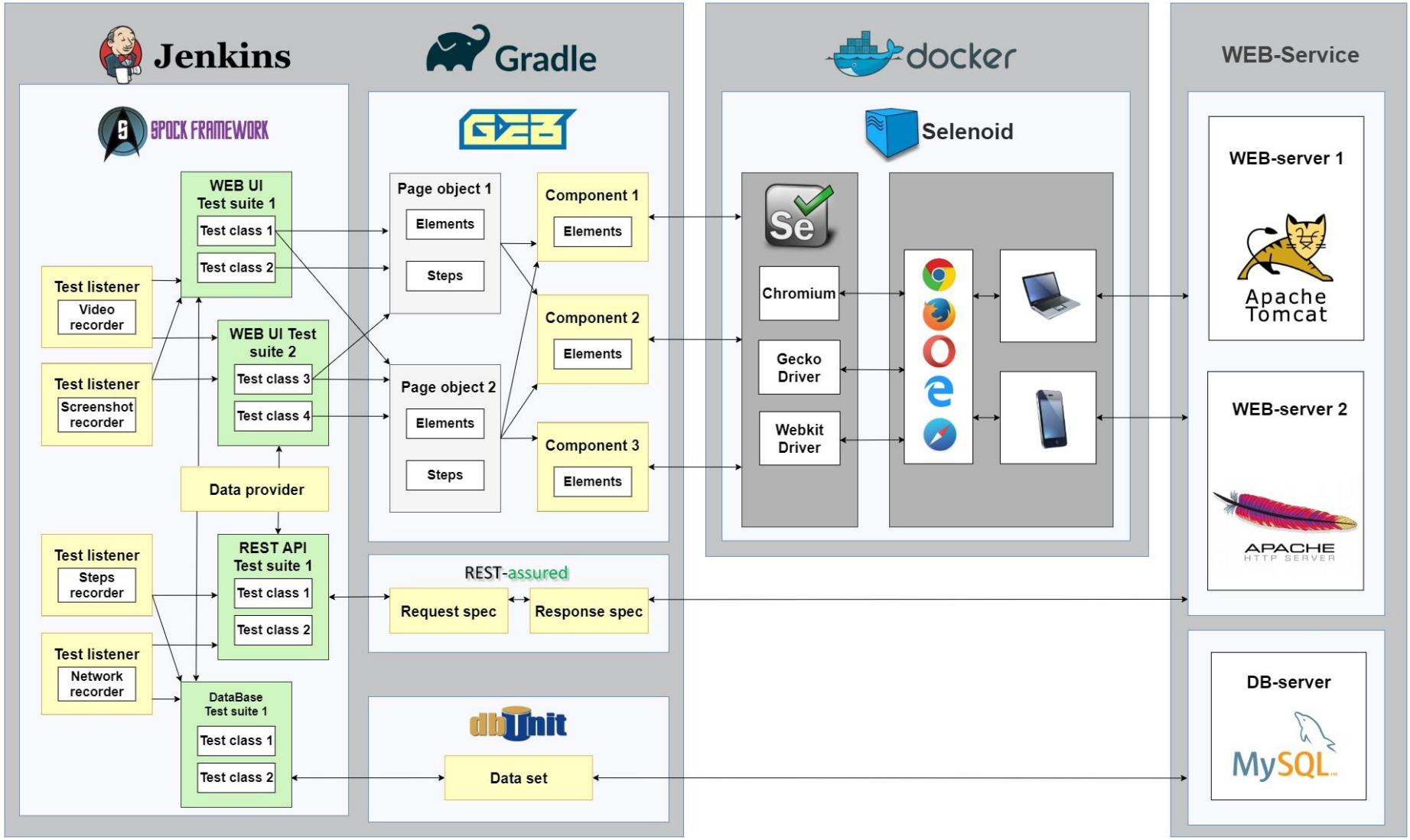

Рисунок 3.10 – Загальна структура інформаційної технології

4 РОЗРОБКА ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

Технологія, що розроблюється в рамках даної роботи, повинна являти собою платформо-незалежний каркасний проект (фреймворк), який, будучи підключеним як залежність до проекту автоматизації тестування, надаватиме інженерам готову архітектуру та усі необхідні механізми для розробки автоматизованих сценаріїв відповідно до бізнес-логіки програмного продукту, що тестується. Для досягнення даної мети, розробка повинна виконуватися на мові, що дозволяє виконувати програми на різних платформах без зміни вихідного коду.

Вихідний код проекту, включаючи усі внутрішні залежності, повинен існувати у відкритих репозиторіях у вигляді зібраного артефакту, який можна завантажити та підключити в інші проекту за допомогою менеджерів залежностей. Також, приймаючи до уваги динамічність сфери IT-розробки, розроблений артефакт повинен враховувати версійність, тобто розширення та оновлення функцій наряду зі зворотною сумісністю.

На основі наведених вимог та проведеного аналізу було вирішено розробити технологію у вигляді бібліотеки на мові Groovy, яка компілюється в Java байт-код та може бути виконана в Java Virtual Machine (JVM) на будь-якій операційній системі. Збирання проекту відбуватиметься за допомогою системи збирання проектів Gradle, заснованому на ідеології та динамічності Groovy. Його предметно-орієнтована мова програмування (DSL) дозволяє створити завдання (tasks) для кожної з потреб даної технології: від завантаження залежностей та підготовки тестового оточення до запуску тестових сценаріїв та агрегації вихідних результатів.

4.1 Збирання проекту

Gradle – система автоматичного збирання, яка далі розвиває принципи, закладені в Apache Ant та Apache Maven і використовує предметно-орієнтовану мову на основі мови Groovy замість традиційної XML-подібної форми представлення конфігурації проекту. На відміну від Apache Maven, заснованого на концепції життєвого циклу проекту, Gradle використовує спрямований ациклічний граф для визначення порядку виконання завдань (DAG [34]). Приклад такого графу завдань в контексті збирання проекту наведено на рисунку 4.1.

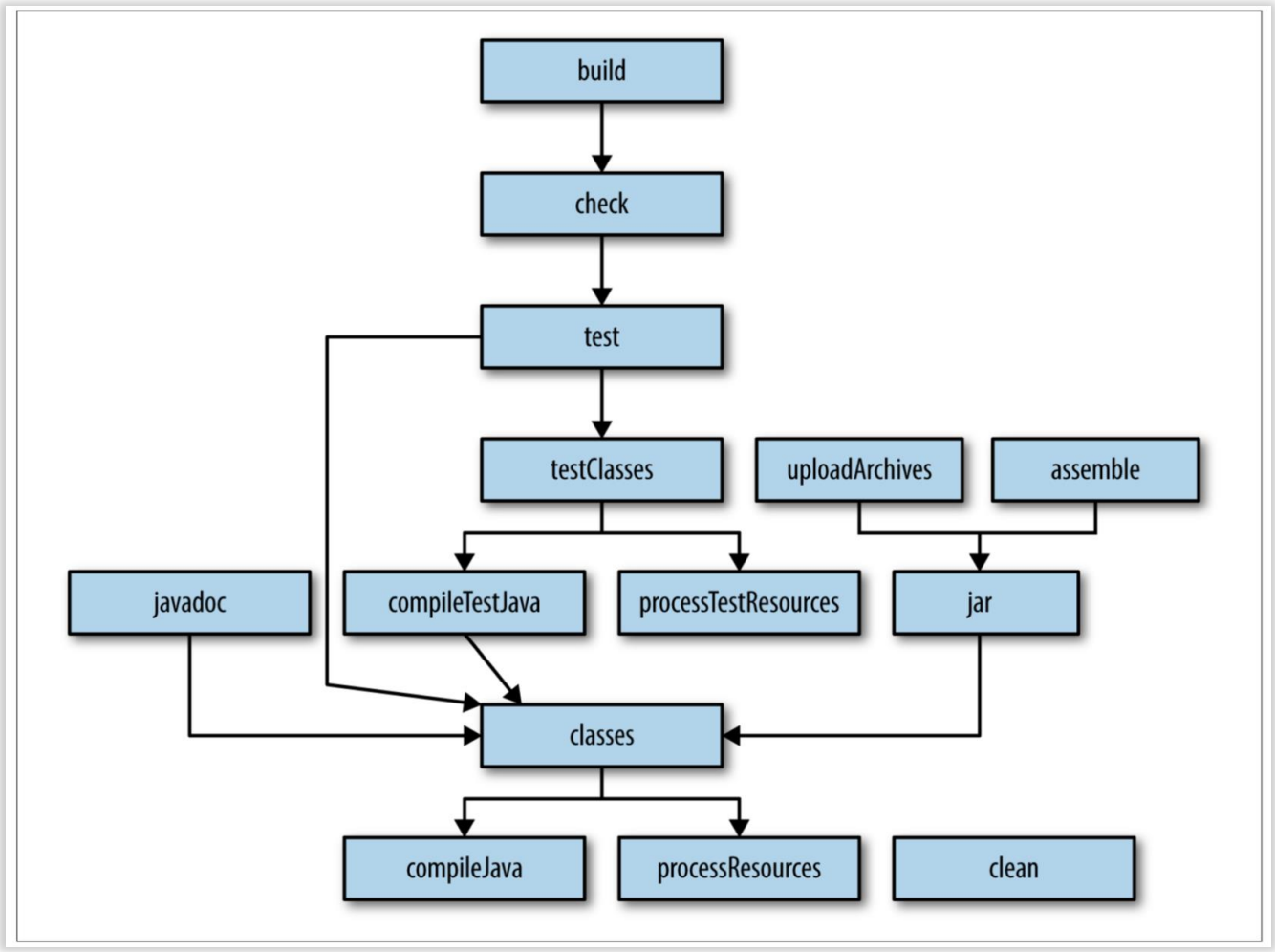

Рисунок 4.1 – Орієнтований ациклічний граф завдань в конфігурації Gradle

Gradle було розроблено для побудови мультипроектів, які можуть розростатися; він підтримує інкрементальне збирання, тобто визначає, які частини було змінено, і виконує тільки ті задачі, які залежать від цих частин. Основним

інструментом при роботі з Gradle є gradle.build файл, де на мові Groovy конфігуруються області опису залежностей, завдань та їх відносин. Далі наведено приклад опису залежностей розроблюваної технології та реєстрація репозиторіїв для пошуку та завантаження:

```
repositories {
    mavenLocal()
    jcenter()
    maven { url 'https://repo.grails.org/grails/core' }
}

dependencies {
    compile 'org.codehaus.groovy:groovy:2.5.2'
    compile 'ch.qos.logback:logback-classic:1.1.7'
    compile "org.aspectj:aspectjweaver:1.9.2"
    testCompile group: 'io.rest-assured', name: 'rest-assured', version: '4.2.0'
    compile group: 'io.rest-assured', name: 'json-schema-validator', version: '4.3.2'

    testCompile group: 'org.duracloud', name: 'unit-test-db', version: '3.7.4'
    testCompile group: 'io.github.bonigarcia', name: 'webdrivermanager', version: '3.0.0'

    testCompile 'org.spockframework:spock-core:1.3-groovy-2.5'
    testCompile "org.seleniumhq.selenium:selenium-java:3.141.59"
    testCompile group: 'org.monte', name: 'screen-recorder', version: '0.7.7'
    driver group: 'mysql', name: 'mysql-connector-java', version: '8.0.15'

    testFiles "ua.edu.sumdu.bnesteruk.curiosity:testFiles:1.9.4.1-SNAPSHOT@zip"
}
```

У цьому ж файлі конфігуруються завдання підготовки тестового середовища (очистка/створення файлової системи, завантаження знімку бази даних) завдання запуску тестових та публікації результатів. Фрагмент коду нижче показує приклад завдання для старту тестів типу WEB UI:

```
task uiTest(type: Test, dependsOn: ['fetchTestFiles']) {
    group 'verification'
    outputs.upToDateWhen { false }
    exclude '**/api/**'

    if (project.hasProperty('excludeTests')) {
        exclude project.property('excludeTests')
    }
    if (project.hasProperty('includeTests')) {
        include project.property('includeTests')
    }

    testLogging.showStandardStreams = true
    testLogging.exceptionFormat = 'full'
}
```

В даному методі, пакет тестів REST API ігнорується, також додається можливість додаткової фільтрації тестових класів. А для самого завдання uiTest

вказана залежність від завдання fetchTestFiles: таким чином, при старті тестів gradle автоматично виконає завантаження необхідних тестам вхідних артефактів: файлів та параметрів:

```
task fetchTestFiles() {
    group 'environment'
    doLast {
        copy {
            from zipTree(file(configurations.testFiles.find { it.name.endsWith('.zip') }))
            into "$buildDir/resources/test/files"
        }
    }
}
```

Завдання запуску інших тестів мають подібну структуру та виключаюсь тести WEB UI чи REST API відповідно.

На рисунку 4.2 показаний ланцюг орієнтованого графу при запуску завдання uiTest:

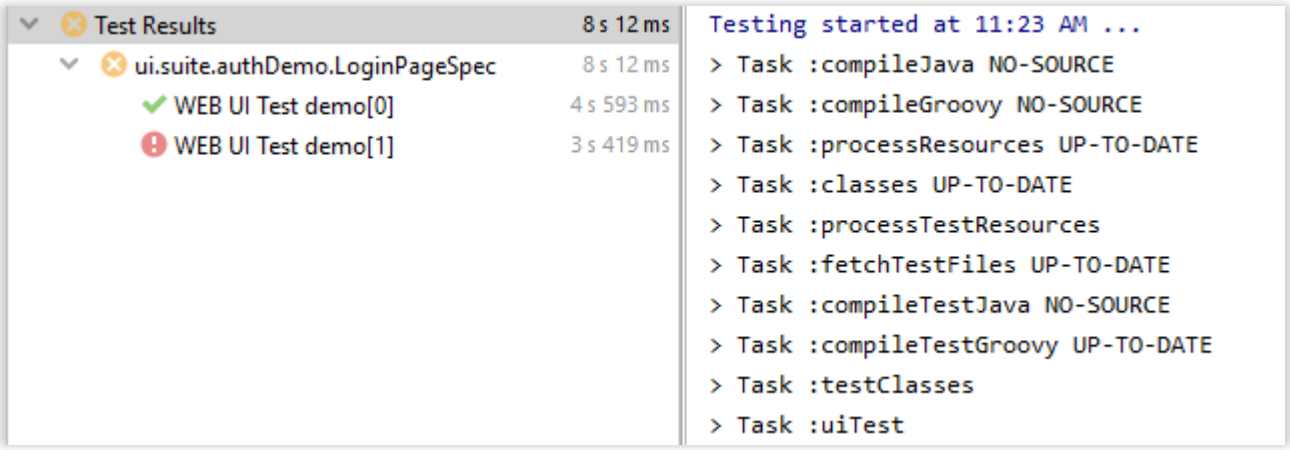

Рисунок 4.2 – Виконання залежних завдань Gradle перед запуском WEB UI тестів

4.2 Реалізація модуля WEB UI тестування

4.2.1 конфігурація Selenium Webdriver

Основна функція даного модуля – надавати тестам механізм взаємодії з веб-сторінками незалежно від типу та версії веб-браузера. Бібліотека GEB, будучи обгорткою для Selenium Webdriver, дозволяє керувати створенням та подальшою поведінкою об’єкту веб-драйверу залежно від середовища. Файл конфігурації GebConfig.groovy містить опис головних аспектів поведінки:

– базовий URL (хост розгортання програмного продукту, що тестується);
– імена та версії веб-браузерів, які підтримуються авто-тестами;
– менеджери завантаження веб-драйверів;
– початкові габарити вікна, таймаути очікування елементів тощо;

GebConfig, будучи скриптом Groovy, надає інженерам можливість реалізувати будь-яку поведінку тестової сесії та зберігати у об’єкті конфігурації необхідні параметри. Нижче наведено приклад обрання веб-браузера залежно від переданої з тестів назви:

```
driver = {
    WebDriver driver
    switch (browserName?.trim()?.toLowerCase() ?: "") {
        case "firefox":
            driver = new FirefoxDriver(setUpFirefoxOptions())
            break
             /* Other drivers go here */
        default:
            Closure<ChromeOptions> setUpGoogleChromeOptions = {
                WebDriverManager dm = WebDriverManager.chromedriver()
                if (chromeDriverVersion) {
                    dm.version(chromeDriverVersion)
                }
                dm.setup()

                ChromeOptions options = new ChromeOptions()
                ['--lang=en-GB', '--no-sandbox', '--disable-dev-shm-usage'].each {
                    options.addArguments(it)
                }
                options
            }
            driver = new ChromeDriver(setUpGoogleChromeOptions())
    }
    driver
}
```

На прикладі можна побачити, як при створенні об’єкту ChromeDriver викликається метод **setUpGoogleChromeOptions()**. Саме тут в гру вступає менеджер завантаження веб-драйверів. На основі поточної операційної системи, типу та версії браузеру, автоматично завантажується відповідний виконуваний файл веб-драйверу. Також, тут задаються необхідні опції, що будуть додані за замовчуванням при відкритті веб-браузеру (мова інтерфейсу, директорія завантаження файлів тощо).

4.2.2 Реалізація загальних атрибутів тестової сесії WEB UI

Базовим класом для усіх тестів, що взаємодіють з графічним інтерфейсом веб-сервісу, є BaseE2ESpec.groovy. Саме тут ініціалізуються атрибути, спільні для усіх WEB UI тестів, а саме:

- визначаються формати дат та часу (для синтаксичного аналізу та взаємодії за датами в інтерфейсі);
- завантажується список тестових користувачів (визначається автоматично залежно від тестового оточення з урахуванням ролей);
- завантажуються тестові параметри відповідно до поточного пакету та назви тесті (детальніше – у розділі 4.4 – провайдер тестових даних);

Приклад конфігурації базових атрибутів наведено у наступному фрагменті коду. Слід зауважити, що даний метод виконується один раз, перед першою ітерацією першого тестового метода у класі, тобто усі тестові методи отримують ідентичні інструкції щодо оточення:

```
def setupSpec() {
    loadSpecData()
    users = getTestUsers()
    Class className = getClass()
    suiteName = className.package.name[(className.package.name.lastIndexOf(".") + 1)..-1]
    specName = className.name[(className.name.lastIndexOf(".") + 1)..-1]

    String dateFormat = Environment.instance.get 'runDateFormat'
    String timeFormat = Environment.instance.get 'runTimeFormat'
    Date now = new Date()
    runDateTime = LocalDateTime.ofInstant(now.toInstant(), ZoneId.systemDefault())
    runDateTime = runDateTime.minusMinutes(1)
    runDate = new SimpleDateFormat(dateFormat).format(now)
    runTime = new SimpleDateFormat(timeFormat).format(now)
    suiteId = Helper.randomId()

    pages = to ApplicationPages
}
```

Будучи нащадком класу geb.spock.GebReportingSpec, клас BaseE2ESpec та усі його нащадки ідентифікуються як тестові методи при виконанні завдання Gradle uiTest, тобто інженеру-автоматизатору не потрібно створювати додаткові файли тестових наборів, на відміну від більшості існуючих тестових фреймворків.

4.2.3 Абстракція бізнес-логіки в тестових методах

Дана технологія автоматизації пропонує створювати тестові сценарії у форматі, зрозумілому не тільки автору тесту, а й усій команді розробки чи замовнику. Тестові сценарії повинні оперувати лише високорівневими термінами бізнес-логіки програмного продукту, що тестується: фактично, в тілі тестового методу повинна відображатися історія користувача, описана у специфікації до продукту. Даний рівень абстракції досягається використанням шаблону «Об’єкт сторінки» (Page Object): усі аспекти взаємодії з елементами веб-сторінок (наприклад, опис локатору, очікування, зчитування/задання значення) приховуються у середину окремих класів, які, в свою чергу, надають тестам набір легких для читання та розуміння методів взаємодії з частинами веб-сторінки. Таким чином, тестові методи не знають нічого про структуру веб-сторінки, так само, як і про логування дій та помилок. Це надає незаперечні переваги при розробці та актуалізації тестових наборів:

- дії тесту зрозумілі кожному, хто з ним працює;
- тест відображає певну частину специфікації, а отже може бути однозначно пов’язаним з конкретним розділом та історією користувача;
- оновлення локаторів веб-елементів чи поведінки сторінки відбувається лише в одному об’єкті веб-сторінки. При цьому усі тести, що користуються методами даного об’єкту, автоматично отримують оновлену та вірну поведінку.

Сама назва тестового методу описує суть його роботи та відповідає тестовому випадку (test case) з тест-плану. Наряду з тестовим класом та пакетом тесту, це ідентифікує тестовий сценарій серед технічних вимог:

- Пакет тесту – розділ (feature);
- Клас тесту – історія користувача (user story);
- Назва тесту – тестовий випадок (test case);

Далі наведено приклад тестового сценарію у форматі, що пропонує розроблювана технологія.

```
def 'Add Case Monitor'(Map data) {
    setup: "Open monitoring panel"
    pages.monitoringTab.openCaseMonitorsFor(SIMULATION_FINE)
    when:
    pages.tableTab.selectTabByName(TableTabs.MONITORING)
    then:
    caseMonitorPanelModule.caseMonitorsPanelIsDisplayed()
    when: "Particular monitor is selected"
    caseMonitorPanelModule.selectMonitorName(data.monitorName)
    then: "Monitor's parameters are presented in its options"
    caseMonitorPanelModule.timeRangeMinValue == (data.defaultMinTimeRange)
    caseMonitorPanelModule.timeRangeMaxValue == (data.defaultMaxTimeRange)
    when:
    caseMonitorPanelModule.setSignal(data.signalValue)
    caseMonitorPanelModule.setRunningAverage(!data.signalValue)
    then: "No data displayed at the chart"
    caseMonitorPanelModule.noDataMessage == data.noDataMessage

    where:
    data << getTestData()
}
```

Так же, як і реальний користувач, тест оперує поняттями взаємодії з додатком в цілому, не переймаючись локалізацією веб-елементів всередині сторінки. Навіть людина, яка не ознайомлена зі специфікою програмного продукту, може зрозуміти, які дії виконуються в даному тестовому сценарії.

4.2.3 Реалізація шаблонів «Об’єкт сторінки», «Компонент сторінки»

Для викликів методів об’єктів сторінок в тестових методах створено об’єкт типу ApplicationPages, де зареєстровані окремі сторінки програмного продукту. Він і сам являється об’єктом типу GEB Page Object, тому має секцію content, де задаються його елементи. В даному випадку – сторінки, що тестуються:

```
class ApplicationPages extends BasePage {
    static content = {
        loginPage(cache: true) { browser.at LoginPage }
        dashboard(cache: true) { browser.at DashBoard }
        administration(cache: true) { browser.page Administration }
        reporting(cache: true) { browser.page Reporting }
        userSettings(cache: true) { browser.page UserSettings }
        projectPage(cache: true) { browser.page Project }
        runPage(cache: true) { browser.page Run }
        filesTab(cache: true) { browser.at FilesTab }
        monitoringTab(cache: true) { browser.at MonitoringTab }
        sessionsTab(cache: true) { browser.at SessionTab }
        resultsTab(cache: true) { browser.at ResultsTab }
    }
}
```

ApplicationPages ініціалізується у методі setupSpec базового класу, тобто на момент початку любого з тестів усі об’єкти сторінок вже готові до використання.

Наступний фрагмент коду демонструє об’єкт сторінки, реалізований в стилі GEB.

```
class MonitoringTab extends TableTabs {
    static at = {
        waitFor(10, 1) { jobsTable.isDisplayed() }
        selectedWorkspaceTab(MONITORING).isDisplayed()
    }
    static content = {
        jobsTable { $("#jobsTable") }
        tableHeader { jobsTable.$("thead").module(new JobTableRow(cellElement: "th")) }
        caseMonitorPanelModule { module CaseMonitorPanel }
        dateFilterLink { $('.th-job-start .dropdown-toggle') }
        dateFilterOption { String value -> dateFilterMenu.find('label', text: value) }
    }
    List<JobTableRow> getSelectedRows() {
        getSelectedRows(jobsRows)
    }
    @Step
    boolean waitForAllFinished(LocalDateTime dateTime, Map moreCriteria = [:], Long
timeOutSeconds = 120) {
        waitForAllFinished(moreCriteria << [dateTime: dateTime], timeOutSeconds)
    }
    @Step
    def clickJobRunLink(JobType jobType, Map moreCriteria = [:]) {
        JobTableRow jobRow = filterJobs(moreCriteria << [type: jobType]).first()
        jobRow.runLink.click()
    }
    @Step
    void openCaseMonitorsFor(JobType type, LocalDateTime date = null) {
        filterJobs(type: type, dateTime: date).first().caseMonitorIcon.click()
    }
}
```

Основними секціями подібних класів є:

- **at** – блок перевірки показу веб-сторінки. Виконується при кожному переході на дану веб-сторінку. Це економить час у випадку, якщо програмний продукт показав неочікувані результати: авто-тест припинить виконання зі зрозумілою помилкою;
- **content** – область визначення потрібних для взаємодії елементів. На відміну від аналогічних областей в класах Webdriver, тут кожен вираз є функцією, тобто для локалізації контенту розробнику авто-тестів доступні усі можливості мови Groovy. З прикладу видно, що елементами

можуть бути як конкретні області HTML, так і окремі класи компонентів, так и функції обробки даних.

– **steps** – перелік методів, що представляють кроки, необхідні тестам від даної сторінки; Кожен такий крок буде відображений у фінальному звіті з урахуванням переданих у метод даних;

Як видно з прикладу, майже все в класі є функціями, тобто виконується безпосередньо під час виклику, вирішуючи проблеми застарілих посилань на веб-елементи (StaleElementReferenceExpection), що є однією з головних проблем при роботі напряму з API Selenium Webdriver.

На рисунку 4.3 продемонстровано розташування пакетів з компонентами та об’єктами сторінок в структурі програмного проекту. Як видно, пакет page розташований на одному рівні з пакетом тестів suite, та включає в себе дочірні пакети для окремих компонентів. Це дозволяє відобразити відносини реальних веб-сторінок продукту, що тестується, відповідно до класів з об’єктами сторінок.

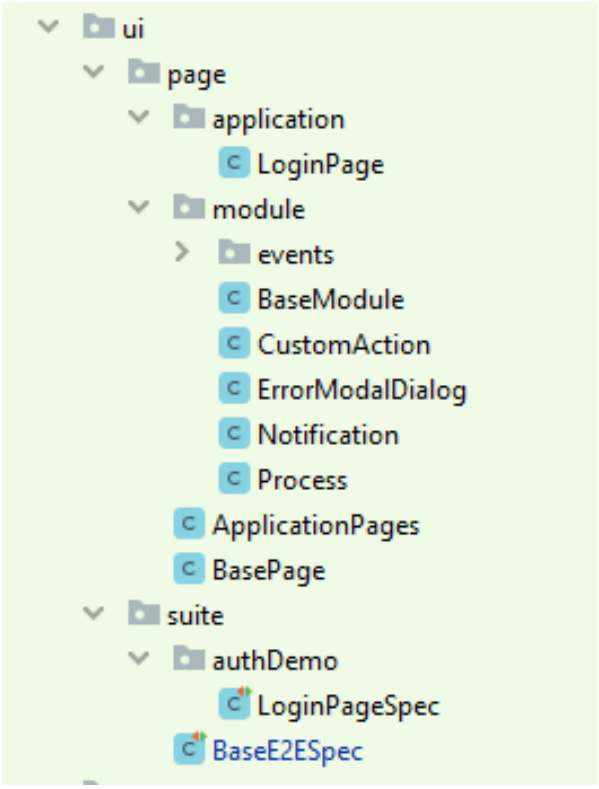

Рисунок 4.3 – Розташування об’єктів сторінок в структурі проекту

4.3 Реалізація модуля REST API тестування

Аналогічно модулю WEB UI, тестування API потребує базових налаштувань та підключення механізму параметризації тестів. Усі загальні атрибути ініціалізуються у базовому класі BaseApiSpec. Окрім конфігурації форматів дат та часу, тут задаються базові параметри запитів та очікувані характеристики відповідей:

```
def setupSpec() {
    loadSpecData()
    users = getTestUsers()
    RestAssured.baseURI = Environment.instance.get 'baseAPIUrl'
    requestSpec = new RequestSpecBuilder().build().
            config(RestAssured.config()
                    .jsonConfig(jsonConfig().numberReturnType(BIG_DECIMAL))).
            log().everything()

    responseSpec = expect().
            statusCode(200).and().
            contentType(ContentType.JSON)

    jsonSchemaFactory = JsonSchemaFactory.newBuilder()
            .setValidationConfiguration(
                    ValidationConfiguration.newBuilder().
                            setDefaultVersion(DRAFTV4).freeze()).freeze()
}
```

Об’єкти requestSpec та responseSpec в деякому сенсі є аналогами об’єктів сторінок в тестах WEB UI та приховують у собі деталі реалізації API. В наведеному фрагменті коду задаються правила розпізнання чисельних значень (BIG_DECIMAL замість INT), правила логування та очікуваний статус відповіді серверу. Далі, в самих тестових методах можна використовувати дані об’єкти без повторення даних налаштувань:

```
def 'Search books via google api by author with limit'(Map data) {
    expect:
    given().
            spec(requestSpec).
            pathParam("limit", data.limit).
            pathParam("searchQuery", data.query).
            expect().
            spec(responseSpec).
            when().
            get("?maxResults={limit}&q={searchQuery}").
}
```

Слід зауважити, що механізми запуску і фільтрації тестів, їх параметризації, логування та публікації результатів, є уніфікованими в рамках даної технології та однаково працюють для кожного з тестових модулів. Дня уникнення дублювання розділів, опис параметризації для модулей REST API та DB буде опущено. Деталі реалізації відповідних механізмів можна побачити у Додатку В – лістинг проекту.

4.4. Реалізація провайдера тестових даних

Завдання механізму провайдера даних (Data provider) – відокремити реалізацію тестового методу від значень, якими тест оперує. Це дає змогу повторно виконати один и той же тестовий метод для покриття різних варіантів використання продукту. Наведемо приклад класичного параметризованого тесту в реалізації Spock Framework:

```
def "maximum of two numbers"() {
    expect:
    Math.max(a, b) == c
    where:
    a | b || c
    3 | 5 || 5
    7 | 0 || 7
    0 | 0 || 0
}
```

Секція where дозволяє задати вхідні та очікувані дані у табличному вигляді. В даному випадку Spock розпізнає параметризацію та автоматично повторює тест три рази, використовуючи для кожної ітерації новий набір параметрів. Хоча майже кожен тестовий фреймворк надає подібні механізми параметризації тестів, усі вони слабо підходять для тестування на системному рівні, так як, найчастіше, дані пропонується зберігати поряд з самим тестом у тестовому класі. Це приводить до таких проблем, як:

- розростання об’єму тестового класу: для тестів системного рівня кількість вхідних та очікуваних параметрів може сягати сотні для однієї ітерації тесту;
- залежність тесту від тестового середовища: на різних серверах дані доступу користувачів можуть відрізнятися;

– неможливість оновлення даних без компіляції усього проекту: навіть для зміни одного очікуваного значення тесту необхідно мати доступ до репозиторію тестів, мати права на зміну коду на навички роботи с системою контроля версій;

Для уникнення подібних проблем, в рамках даної роботи був розроблений унікальній механізм провайдера даних, що дозволяє зберігати тестові параметри поза межами тестових класів та автоматично передавати дані у тести перед кожною ітерацією (рис. 4.4).

Test .groovy class

```
def 'Add Case Monitor'(Map data) {
    setup: "Open monitoring panel"
    pages.monitoringTab.openCaseMonitorsFor(SIMULATION_FINE)
    when:
    pages.tableTabs.selectTabByName(TableTabs.MONITORING)
    then:
    caseMonitorPanelModule.caseMonitorsPanelIsDisplayed()
    when: "Particular monitor is selected"
    caseMonitorPanelModule.selectMonitorName(data.monitorName)
    then: "Monitor's parameters are presented in its options"
    caseMonitorPanelModule.timeRangeMinValue == (data.defaultMinTimeRange)
    caseMonitorPanelModule.timeRangeMaxValue == (data.defaultMaxTimeRange)
    when:
    caseMonitorPanelModule.setSignal(data.signalValue)
    caseMonitorPanelModule.setRunningAverage(!data.signalValue)
    then: "No data displayed at the chart"
    caseMonitorPanelModule.noDataMessage == data.noDataMessage

    where:
    data << getTestData()
}
```

External .json resource

```
"Add Case Monitor": [
  {
    "projectName": "Project_e2e_jobs",
    "runName": "Run_Discretize_And_Simulate",
    "monitorName": "monitor-3",
    "averageValue": "-1.637 1/sec",
    "defaultMinTimeRange": 1000,
    "defaultMaxTimeRange": 4950,
    "confidenceIntervalValues": [
      "68.27% ( 1.0 sigma)",
    ],
    "additionalDataSetValues": [
      "Is signal converged",
      "Percent complete"
    ],
    "timeUnitsValues": [
      "dimlessTime"
    ],
    "signalUnitsValues": [
      "1/dimlessTime"
    ],
    "noDataMessage": "No data to display."
  }
```

Рисунок 4.4 – параметри тесту по за межами тестових класів

Суть ідеї полягає у тому, що ми можемо отримати дані про назву тестового методу, його клас та пакет, ще перед стартом тестової ітерації. Таким чином, ми отримуємо метадані про положення тесту в структурі проекту, та можемо зв’язати їх з положенням файлів даних тесту у ресурсах проекту. При цьому, файли даних не залежать від компіляції проекту, так як ресурси можна зберігати в окремих репозиторіях, або інших публічних ресурсах, та завантажувати за потребою (завдання fetchTestFiles від виконання якого залежать завдання uiTest, apiTest,

dbTest). Рисунок 4.5 показує відповідність структури тестових класів та ресурсів тестових даних.

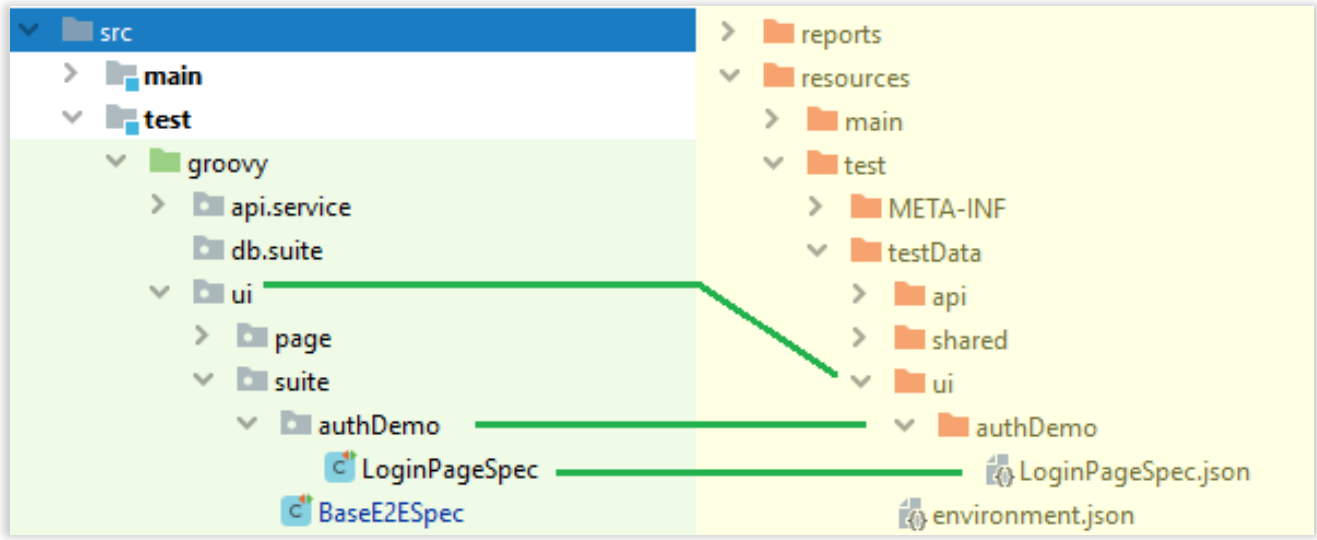

Рисунок 4.5 – Назва та пакет тестового класу відповідають назві та директорії файлу параметрів в ресурсах проекту

Більш того, такий підхід не має обмежень щодо типу файлів, в яких зберігаються дані. Розроблювана технологія надає можливість зберігати параметри тесту у форматах JSON та XLSX (Excel). Але реалізуючи інтерфейс TestDataReader, можна за потребою зчитувати файли інших форматів. Виклик методу зчитування параметрів доданий у setupSpec блоки базових класів кожного з модулів проекту, тобто кожен тестовий сценарій, реалізований в класах-нащадках, буде отримувати дані з файлів ресурсів, і інженер-автоматизатор може зосередитися саме на заповненні тестових даних, а не на передачі їх до тестів.

В наступному фрагменті коду показано, як метадані тесту отримуються з об’єкту тестового методу (SpecInfo), передаються у метод зчитування файлів ресурсів (getDataFile()) та приводяться до типу мапи (асоціативного масиву).

```
class TestDataUtil implements TestDataReader {
    private static final String UNICODE_CHARSET = 'UTF-8'
    private static final String DATA_FOLDER = 'testData'
    private static final Gson gson = new Gson()

    static Map<String, Object> loadSpecData(SpecInfo currentSpec) {
        Map<String, Object> data = [:]
        String fileName = currentSpec.filename.replace('groovy', 'json')
```

```
        String testPkgName = currentSpec.package.split('\\.').first()
        String suitePkgName = currentSpec.package.split('\\.').last()
        if (getDataFile(testPkgName, suitePkgName, fileName)?.exists()) {
            data = getJsonMappedData(testPkgName, suitePkgName, fileName)
        }
        return data
    }

    static <T> Map<String, T> getJsonMappedData(
            String pkgName, String suiteName, String fileName) {
        String testDataJson =
                getDataFile(pkgName, suiteName, fileName).getText(UNICODE_CHARSET)
        return gson.fromJson(testDataJson, new TypeToken<Map<String, T>>() {}.type)
    }

    static File getDataFile(String testPkgName, String suiteName, String fileName) {
        String dataResPath =
                Paths.get(DATA_FOLDER, testPkgName, suiteName, fileName).toString()
        URL dataResURL = TestDataUtil.getClassLoader().getResource(dataResPath)
        return dataResURL ? new File(dataResURL.toURI()) : null
    }
}
```

4.5 Проектування та реалізація слухачів тестових методів

Слухачі (Listeners) дозволяють відстежувати поведінку та результати тесту без втручання у код самого тесту (тестові методи не повинні перейматися логуванням. Їх справа – перевіряти програмний продукт). В рамках даної роботи були реалізовані наступні случахі тестової сесії:

- StepListener: показ дій тесту в реальному часі;
- RunListener: показ часу та параметрів тестової ітерації, параметрів тесту;
- OutputCapture: додання потоку виводу системи до фінального звіту;
- VideoRecorder: запис екрану під час перебігу тесту;
- ScreenshotRecorder: запис стану веб-сторінки під час виникнення помилки;

Результати роботи кожного з наведених слухачів додаються до провалених тестових ітерацій у фінальному звіті, та доступні для аналізу. Далі наведено фрагмент реалізації слухача StepListener. Деталі реалізації інших слухачів доступні в додатку В – Лістинг проекту.

```
@Aspect
@SuppressWarnings("unused")
public class StepListener extends RunListener{
    /* === Pointcut bodies. Should be empty === */
    @Pointcut("execution(* *(..))")
    public void anyMethod() {
```

```
    }
    @Pointcut("@annotation(io.qameta.allure.Step)")
    public void withStepAnnotation() {
    }
    /* Don't instantiate explicitly */
    private StepListener() {
    }
    @Before("anyMethod() && withStepAnnotation()")
    public void logStepStart(JoinPoint joinPoint) {
        Helper.log "[STEP]: ${getMethodName(joinPoint)}", 6
    }

    @After("anyMethod() && withStepAnnotation()")
    public void logAfterStep(JoinPoint joinPoint) {

    }
}
```

4.6 Тестування на сервері безперервної інтеграції

Функціональна технологія автоматизації тестування повинна забезпечувати старт та проходження тестових сесій не лише на локальній робочій станції, а й на віддалених тестових середовищах. Для досягнення цієї мети необхідна взаємодія з сервісами безперервної інтеграції та безперервного розгортування. Це системи, що керують потоком завдань збирання проектів (таких, як завдання Gradle).

Розроблювана технологія автоматизацій пропонує наступні можливості для інженерів із забезпечення якості:

- старт тестових сесій за розкладом або за вимогою;
- задання параметрів під час конфігурації тестової сесії:
    - вибір тестових наборів;
    - фільтрація класів тестів;
    - задання типу та версії веб-браузера для тестування;
    - задання параметрів тестового середовища;
- збирання тестового артефакту для ручного, або автоматизованого тестування;
- публікація тестових звітів та аналіз історії випробувань;

Для реалізації кожної з наведених функцій, а також для спрощення міграції конфігурації між різними серверами CI була використана технологія Jenkins Pipeline Script. Фактично, це Groovy-скрипт з визначеним DSL, де кожна дія

тестової сесії описується за допомогою понять етап (Stage) та крок (Step). В скрипті Pipeline також зазначаються тригери збирання, параметри, доступні для зміни, команди старту веб-серверів.

Скрипт збирання (jenkins.deploy_test.groovy) зберігається не на сервері безперервної інтеграції, а репозиторії проекту поряд з кодом тестів. Таким чином проект, що містить тестові набори, сам володіє інформацією про те, як конфігурувати завдання тестової сесії, і може бути без зайвих налаштувань розгорнутий на будь-якому сервері Jenkins CI. Єдине, що задається на стороні серверу CI – шлях до Pipeline Script файлу в репозиторії програмного коду.

За замовчуванням, в даній роботі пропонуються наступні етапи тестової сесії:

- **Checkout** – стягнення вихідного програмного коду з VCS-репозиторію;
- **Setup** – видалення попередніх тестових артефактів, очищення файлової системи;
- **Deploy** – зібрати тестовий артефакт, помістити його до контейнеру веб-серверу, запустити веб-сервер;
- **DB update** – виконати оновлення бази даних заготовленими даними;
- **Pause after deployment** – за потреби, поставити тестову сесію на паузу для виконання ручного тестування програмного продукту;
- **UI tests (Geb)** – запустити виконання тестів WEB UI;
- **REST API tests** – запустити виконання тестів REST API;
- **DB tests** – запустити виконання тестів DB;
- **Pause after tests** – за потреби, залишити тестовий артефакт розгорнутим для ручного аналізу середовища після завершення тестів;
- **Post steps reporting** – кроки обробки результатів тестової сесії: генерація та публікація фінального звіту, нотифікація команди розробки;

Приймаючи до уваги, що конфігурація кожного етапу – це лише метод в скрипті Groovy, любий етап легко змінити ра розширити для потреб конкретного проекту автоматизації.

4.6.1 Параметризація тестової сесії

Секція parameters відповідає за змінні параметри кожної тестової сесії, які можна задати до початку тестування. Наступний фрагмент коду показує приклад їх конфігурації:

```
parameters {
    choice(name: 'BROWSER_NAME', choices: "chrome\nfirefox\n")
    booleanParam(name: 'SET_DRIVER_POSITION', defaultValue: true)
    string(name: 'BROWSER_WIDTH', defaultValue: '1800')
    string(name: 'BROWSER_HEIGHT', defaultValue: '900')
    string(name: 'BROWSER_X_POS', defaultValue: '5')
    string(name: 'BROWSER_Y_POS', defaultValue: '5')
    string(name: 'BASE_URL', defaultValue: '')
    booleanParam(name: 'REFRESH_SCRIPT_ONLY', defaultValue: false)
    booleanParam(name: 'RUN_TESTS_GEB_E2E', defaultValue: true)
    booleanParam(name: 'RUN_TESTS_REST_API', defaultValue: true)
    booleanParam(name: 'RUN_TESTS_DB', defaultValue: true)
}
```

Після завантаження pipeline скрипту, кожен з описаних параметрів буде показаний на екрані старту тестової сесії, а його значення може бути зазначено інженером (рис. 4.6).

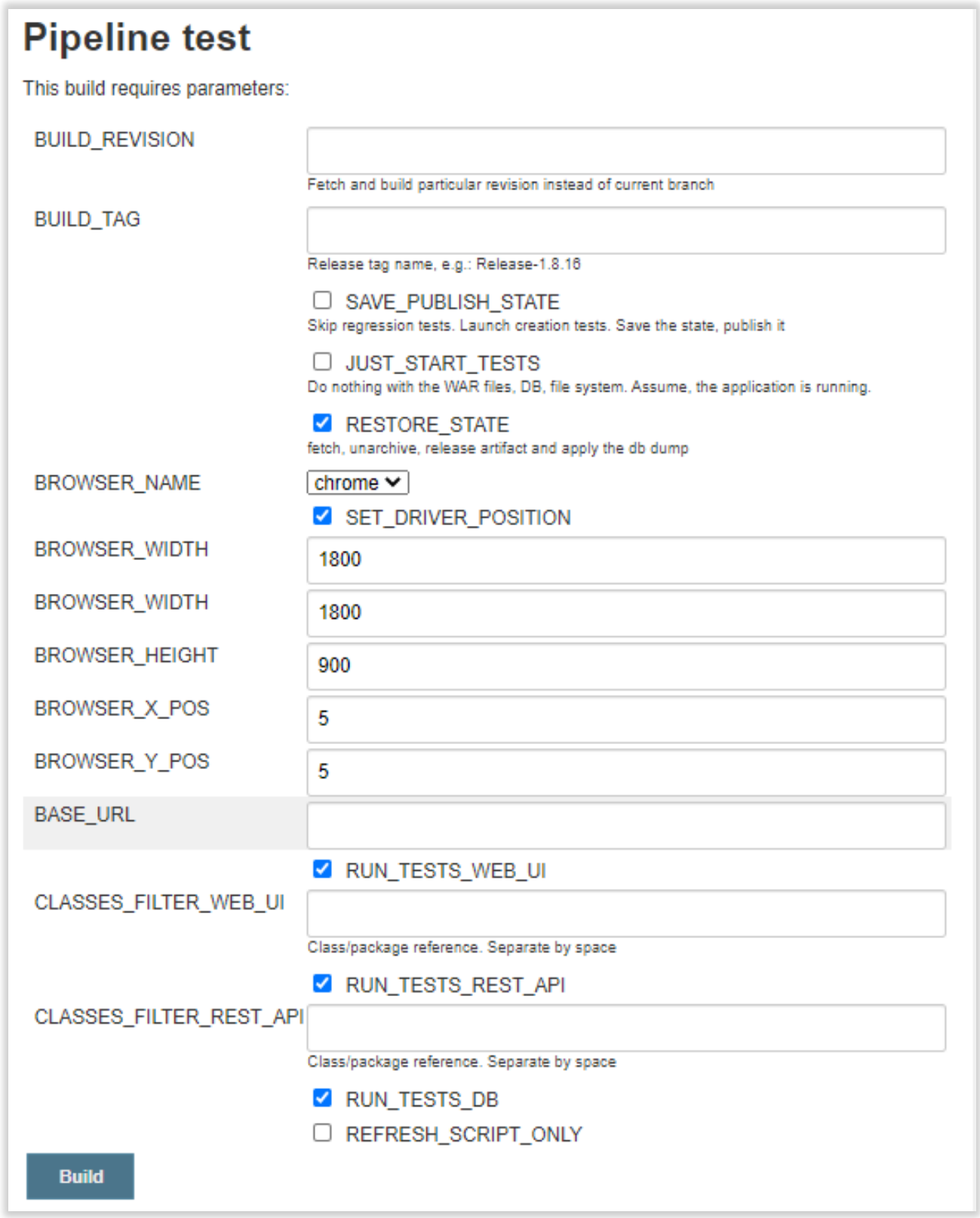

Рисунок 4.6 – параметри тестової сесії з Pipeline Script

4.6.2 Старт за розкладом

В даній технології реалізована можливість запускати тестові набори за розкладом. Для кожного запуску можливо задати власні параметри тестової сесії (наприклад, вранці запускати лише позитивні тести, а вночі – усі набори):

```
triggers {
    parameterizedCron env.BRANCH == 'develop' ? '00 1 * * * %BROWSER_NAME=chrome;' : ''
    cron env.BRANCH_NAME == 'develop' ? '00 18 * * 6' : ''
}
```

4.6.3 Запуск тестових сценаріїв

Стандарт в IT-індустрії на сьогоднішній день – це використання віртуальних контейнерів для запуску додатків. Docker [35] – інструментарій для управління ізольованими Linux-контейнерами. Він дозволяє не переймаючись вмістом контейнера запускати довільні процеси в режимі ізоляції і потім переносити сформовані для даних процесів контейнери на інші сервери, беручи на себе всю роботу зі створення, обслуговування і підтримки контейнерів.

Selenoid [36] – це легкий сервер, який запускає ізольовані браузери в Docker контейнерах та розповсюджується з ліцензією Apache License 2.0. Він має ряд важливих переваг, таких, як якісне управління ресурсами системи, перегляд процесу перебігу тесту в режимі реального часу через VNC-сесію, збереження логів браузеру. Саме через ці причини він був обраний для управління тестовим середовищем. На рисунку 4.7 зображено інформаційну панель управління Selenoid з запущеними контейнерами для веб-браузеру Firefox під час виконання тестів WEB UI. Приклад перегляду перебігу тесту наведено на рисунку 4.8. Як видно, інженеру-автоматизатору одночасно доступні функції перегляду вікна веб-браузеру та логу виконання тестового сценарію.

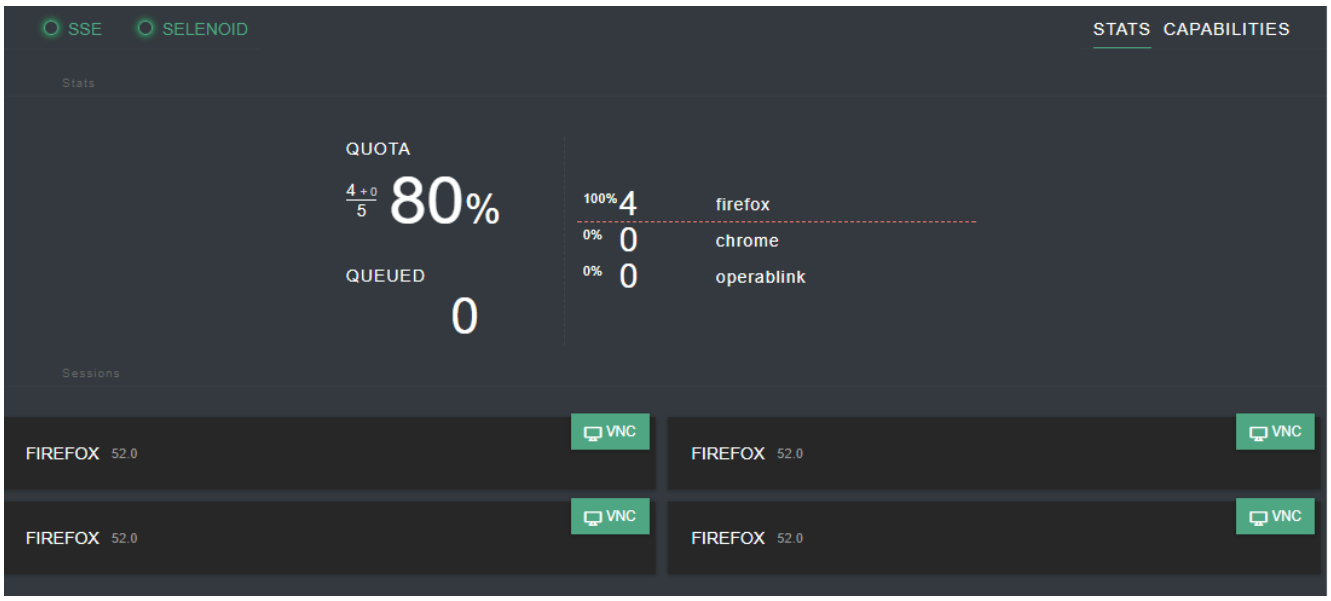

Рисунок 4.7 – Панель управління серверу Selenoid

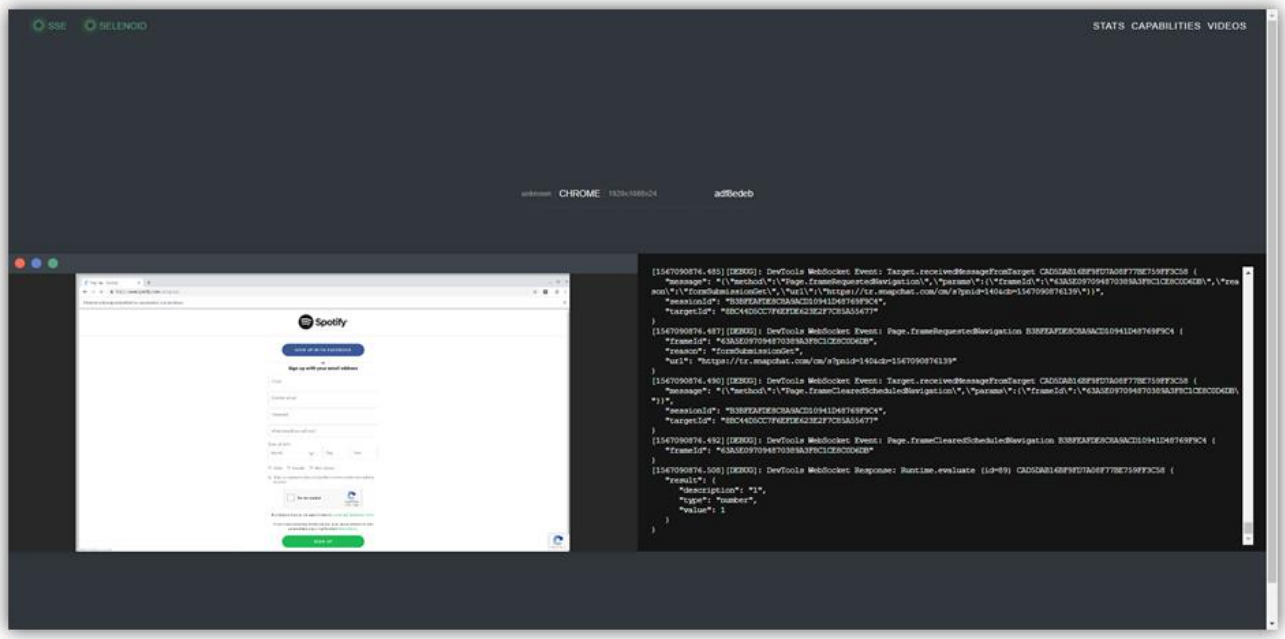

Рисунок 4.8 – Спостереження за виконанням тестового сценарію в режимі реального часу через VNC сесію в контейнері Docker

Наступний фрагмент коду pipeline script демонструє етап налаштування та старту тестових сценаріїв WEB UI (завдання REST API та DB конфігуруються подібним способом).

```
stage('UI tests (Geb)') {
    when {
        allOf {
            equals expected: true, actual: params.RUN_TESTS_WEB_UI
        }
    }
    steps {
        wrap([$class: 'Xvfb', screen: '1920x1080x24']) {
            sh "xhost +"
            sh '''
                docker-compose down && docker-compose up -d
                ./cm_linux_amd64 selenoid start --vnc --force
                --browsers 'firefox;chrome' --last-versions 1
                --args "-limit 10" --port 4444
                ./cm_linux_amd64 selenoid-ui download --force
                ./cm_linux_amd64 selenoid-ui start --port 8080
                ./cm_linux_amd64 selenoid status
                '''
            catchError(buildResult: 'UNSTABLE', stageResult: 'FAILURE') {
                script {
                    def filteredTestsOption = ''
                    if (params.CLASSES_FILTER_WEB_UI) {
                        params.CLASSES_FILTER_WEB_UI.split("\\s+").each {
                            filteredTestsOption += " --tests $it"
                        }
                    }
                    sh "./gradlew uiTest " +
                            "$filteredTestsOption -PignoreTestFailures=true"
                }
            }
        }
    }
}
```

Слід зауважити наступні особливості завдання:

– блок when дозволяє задати умову для виконання чи пропуску етапу тестової сесії (в даному випадку завдання виконається, якщо параметр RUN_TESTS_WEB_UI має значення true);

– перед запуском самих тестів здійснюється розгортування тестового середовища засобами Docker та Selenoid;

– якщо параметр CLASSES_FILTER_WEB_UI не пустий, відбувається розділення його значення на окремі частини (задані класи) та фільтрація тестів для запуску саме вказаних класів;

– після усіх проведених операцій відбувається виклик завдання Gradle uiTest, в яке, за наявності, передаються відфільтровані класи тестів:

4.7 Публікація тестових звітів

Під час виконання кожної ітерації тестових методів вихідні артефакти слухачів (відеозапис, виконані кроки, логи серверу та веб-браузерів тощо) зберігаються в директорію звітів Allure (рис. 4.9). Поряд з ними бібліотека Yandex Allure зберігає .json файл з даними про те, який тест виконувався та які артефакти необхідно прикріпити до нього у фінальному звіті.

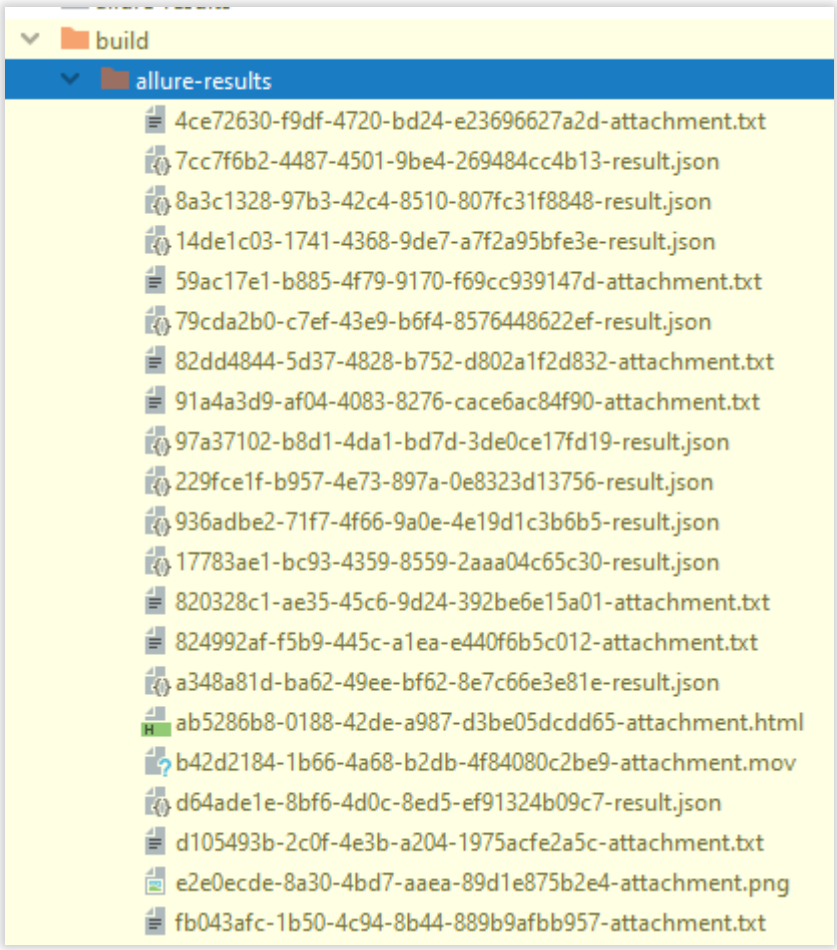

Рисунок 4.9 – вихідні артефакти слухачів тестів

Після виконання завдань тестової сесії, незалежно від її результату, на основі вихідних артефактів генерується фінальний звіт у HTML-форматі. Звіт надає дані про тестову сесію в цілому (рис. 4.10), та виконані тестові набори, розподілені за розділами та історіями користувачів (рис. 4.11).

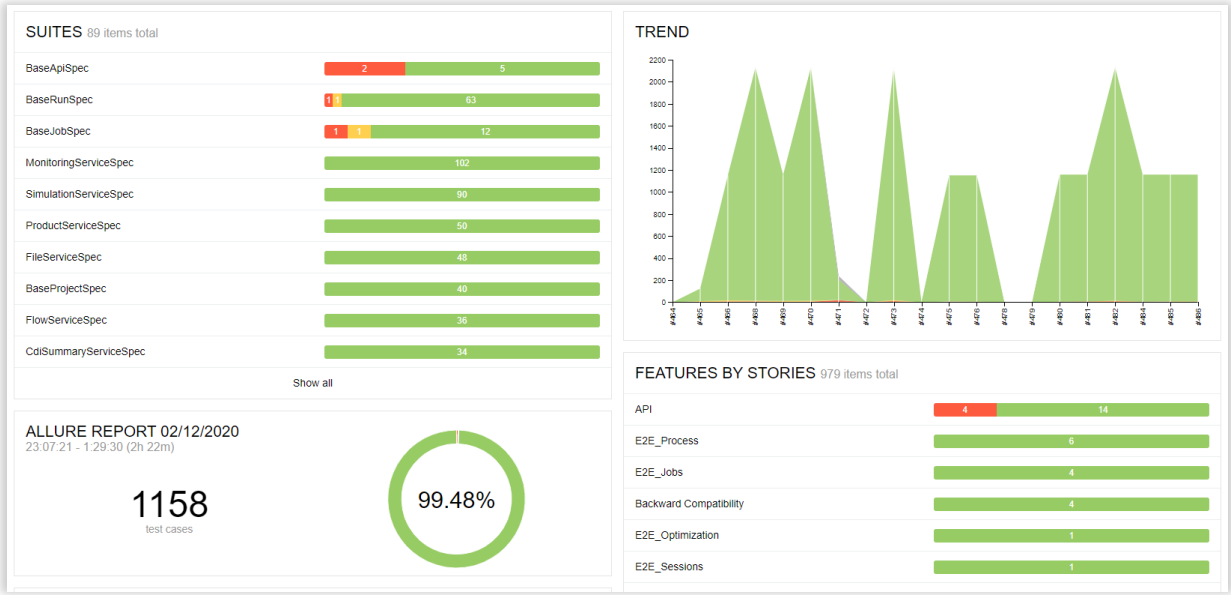

Рисунок 4.10 – Сумарна інформація про тестову сесію в Allure Report

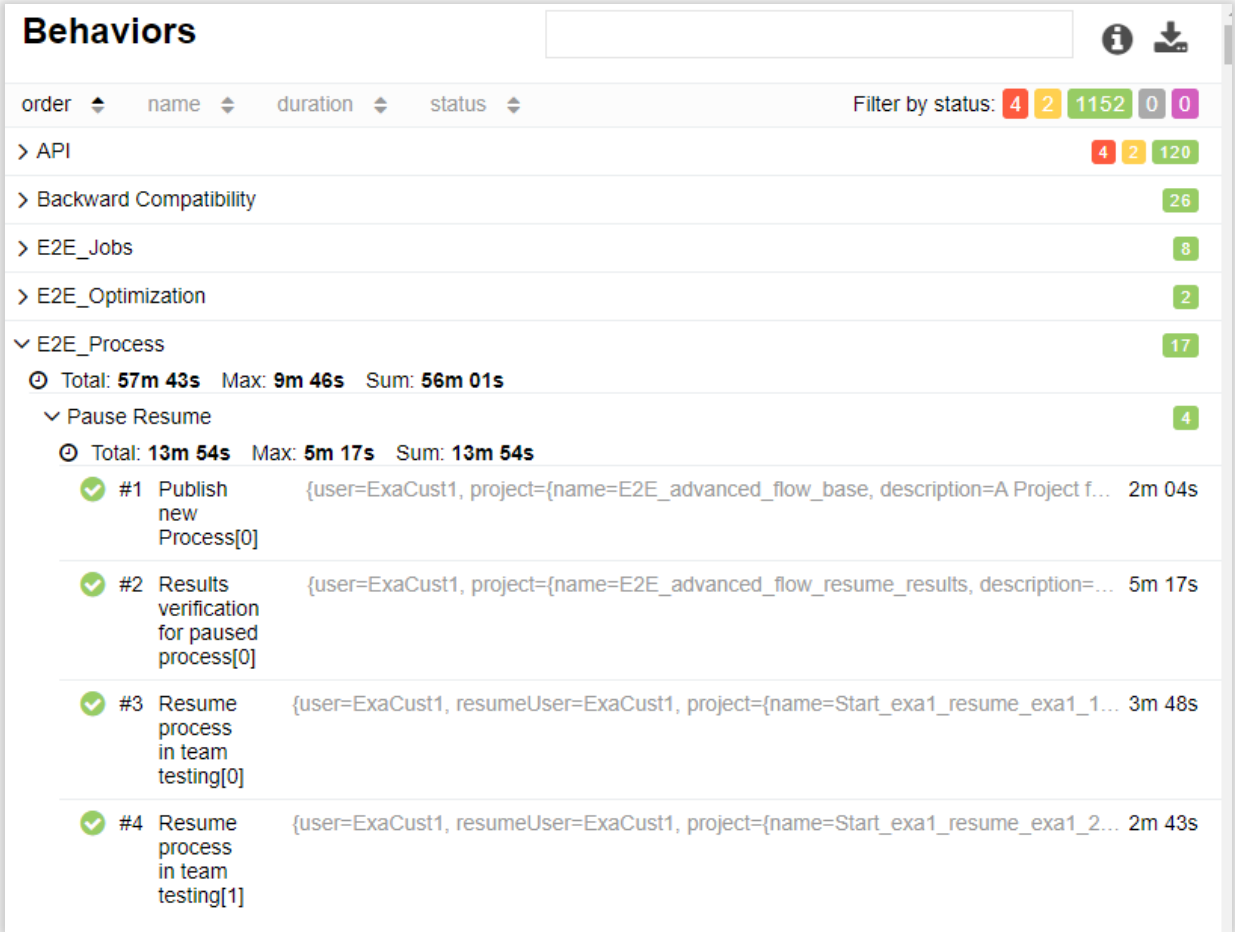

Рисунок 4.11 – Тестові сценарії показані у фінальному звіті відповідно до розділів тестового плану

В рамках даної роботи була реалізована майже максимальна деталізація знайденої помилки тестового сценарію. По кожному проваленому тесту завдяки тестовим слухачам до фінального звіту додаються:

- перелік кроків тесту з переданими в них даними;
- відеозапис екрану під час перебігу тесту (для тестів WEB UI);
- системний потік виводу;
- логи серверу та браузеру;
- відправлені запити та отримані відповіді серверу чи бази даних (для тестів REST API та DB);

Окрім цього, в звіті присутня суть помилки з очікуваними та актуальними даними, та стек-трейс викликів для швидкої навігації до тіла тестового методу. Приклад проваленого тесту з доданими до фінального звіту артефактами показаний на рисунках 4.12-4.13.

```
Condition not satisfied:

newTimeLimits.last() == data.expectedLastPoint['Time']
|              |      |  |    |                 |
|              3800.0 |  |    |                 3650.0
[3300.0, 3800.0]     |  |    [Time:3650.0, Running Average:-1.758, Signal:-1.568]
                     |  [timeUnits:timestep, signalUnits:1/sec, showSignal:true, showRunningAverage:true
                     false

        at org.spockframework.runtime.SpockRuntime.verifyCondition(SpockRuntime.java:38)
        at ui.suite.job.DiscretizeAndSimulate.$spock_feature_3_2(DiscretizeAndSimulate.groovy:148)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.spockframework.util.ReflectionUtil.invokeMethod(ReflectionUtil.java:200)
        at org.spockframework.runtime.model.MethodInfo.invoke(MethodInfo.java:113)
        at org.spockframework.runtime.extension.MethodInvocation.proceed(MethodInvocation.java:98)
```

Рисунок 4.12 – Очікувані, актуальні дані, строка порівняння та стек викликів у фінальному звіті

Рисунок 4.13 – Деталі провaленого тесту: кроки, відеозапис, знімок екрану

ВИСНОВКИ

В сучасному суспільстві кожна новітня галузь залежить від ефективності та якості програмних продуктів. Процес забезпечення якості складних інформаційних систем потребує автоматизації тестування для своєчасного виявлення дефектів та надання структурованої інформації для подальшого аналізу. Дефекти системного рівня, включаючи аспекти інтеграції та розгортання, можливо виявити лише моделюючи роботу з програмним продуктом «від початку до кінця», імітуючи дії реальних користувачів.

Розробка архітектури авто-тестів (їх зв’язків, взаємодії між потрібними інструментами, передача даних та публікація результатів) являється складним завданням, що потребує декількох ітерацій розробки засобами самої команди із забезпечення якості. Це напряму відображається на бюджеті, а головне – на якості програмного продукту, адже ресурси витрачаються не на написання ефективних, стабільних та інформаційних тестових сценаріїв.

Актуальність роботи зумовлена необхідністю розробити інформаційну технологію, що надає комплексне рішення для автоматизації тестування веб-сервісів на системному рівні, їх API та цілісності даних в базах даних.

В рамках даної роботи було проведено:

- аналіз області застосування автоматизації тестування:
  - виявлено види тестувань, що підлягають автоматизації в першу чергу;
  - обрано техніки оптимізації тестової документації та процесу тестування;
- порівняльний аналіз існуючих бібліотек та інструментів автоматизації:
  - досліджено недоліки існуючих рішень;
  - запропоновано методи вирішення недоліків існуючих рішень;
- обрано засоби реалізації:
  - для кожного логічного рівня технології застосовано новітні та динамічні інструменти, що наряду з простотою використання задовольняють усім потребам інженерів-автоматизаторів;
- спроектовано інформаційну технологію:

- передбачено окремі модулі по кожному типу тестування для незалежної розробки тестових сценаріїв різних напрямів;
- рівні параметризації, бізнес-логіки, реалізації взаємодії відокремлені один від одного, що зменшує залежності в коді та полегшує актуалізацію тестових наборів;
- передбачено окремі завдання для побудови тестового артефакту, роботи з базою даних, запуску тестувань;

– розроблено програмне рішення у вигляді каркасного проекту:
  - реалізований механізм автоматичної параметризації тестових методів на основі їх положення в проекті;
  - реалізований механізм автоматичного завантаження залежностей на основі тестового середовища;
  - підготовлено набір класів для наслідування, що надають засоби вирішення кожної з освітлених раніше проблем.

Результатом роботи є створена інформаційна технологія автоматизації, що забезпечує:

– організацію тестових сценаріїв у групи, задання пріоритетів;
– параметризацію тестів, зміну вхідних даних без зміни коду тестів;
– опис тестових сценаріїв в термінах бізнес-логіки;
– управління тестовими сесіями в системах безперервної інтеграції;

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Pethuru Raj, Anupama Raman, Harihara Subramanian, Architectural Patterns: Uncover essential patterns in the most indispensable realm of enterprise architecture: Packt Publishing 2017.

2. Vladimir Khorikov, Unit Testing Principles, Practices, and Patterns: O'Reilly Media, 2020.

3. James A. Whittaker, Exploratory Software Testing: Tips, Tricks, Tours: Addison-Wesley Professional, 2011.

4. Leonard Richardson, Mike Amundsen, RESTful Web API: O'Reilly Media, 2013.

5. Interfacing Abaqus with Dymola: A High Fidelity Anti-Lock Brake System Simulation [Електронний ресурс]: – Режим доступу (Назва з екрану): https://www.3ds.com/fileadmin/PRODUCTS-SERVICES/CATIA/PDF/Interfacing-Abaqus-with-Dymola.pdf

6. Paul M. Duvall, Steve Matyas Andrew Glover, Continuous Integration: Improving Software Quality and Reducing Risk: Addison-Wesley Professional, 2007

7. APACHE LICENSE, VERSION 2.0 [Електронний ресурс]: – Режим доступу: https://www.apache.org/licenses/LICENSE-2.0 – Назва з екрану

8. Glenford J. Myers, Corey Sandler, Tom Badgett, The Art of Software Testing: Wiley, 2011

9. Dr David Tuffley, Software Test Plans: A How To Guide for Project Staff Paperback: CreateSpace Independent Publishing Platform, 2011

10. Satya Avasarala, Selenium Webdriver Practical Guide Paperback: Packt Publishing, 2014

11. Hypertext Application Language [Електронний ресурс]: – Режим доступу: https://ru.wikipedia.org/wiki/Hypertext_Application_Language – Назва з екрану

12. Understanding JSON Schema [Електронний ресурс]: – Режим доступу: http://json-schema.org/understanding-json-schema/ – Назва з екрану

13. Mr Alan J Richardson, Automating and Testing a REST API: Compendium Developments, 2017
14. Apache Groovy Documentation [Електронний ресурс]: – Режим доступу: https://groovy-lang.org/documentation.html – Назва з екрану
15. Gradle vs Maven Comparison [Електронний ресурс]: – Режим доступу: https://gradle.org/maven-vs-gradle/ – Назва з екрану
16. Benjamin Muschko, Gradle in Action: Manning Publications, 2014
17. Frank Appel, Testing with JUnit: Paperback, 2015
18. TestNG [Електронний ресурс]: – Режим доступу: https://testng.org/doc/documentation-main.html – Назва з екрану
19. Rob Fletche, Spock: Up and Running: Writing Expressive Tests in Java and Groovy: O'Reilly Media, 2017
20. SUS [Електронний ресурс]: – Режим доступу: https://en.wikipedia.org/wiki/Specification_by_example – Назва з екрану
21. Domain Specific Language [Електронний ресурс]: – Режим доступу: http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html – Назва з екрану
22. Unmesh Gundecha, Satya Avasarala, Selenium WebDriver 3 Practical Guide: End-to-end automation testing for web and mobile browsers with Selenium WebDriver: Packt Publishing, 2018
23. WebDriver W3C Recommendation 05 June 2018 [Електронний ресурс]: – Режим доступу: https://www.w3.org/TR/Webdriver1/ – Назва з екрану
24. WebdDriver Manager [Електронний ресурс]: – Режим доступу: https://github.com/bonigarcia/Webdrivermanager – Назва з екрану
25. Luke Daley, Marcin Erdmann, Erik Pragt, The Book Of Geb [Електронний ресурс]: – Режим доступу: https://gebish.org/manual/current/ – Назва з екрану
26. Alex Siminiuc, Improve Selenium Code with Automation Patterns: Page Object Model Page Factory Page Elements Base Page Loadable Component: Kindle Store, 2017

27. John Ferguson Smart, BDD in Action: Behavior-driven development for the whole software lifecycle: Manning Publications; 2014
28. DbUnit [Електронний ресурс]: – Режим доступу: http://dbunit.sourceforge.net/ – Назва з екрану
29. Allure Test Report [Електронний ресурс]: – Режим доступу: http://allure.qatools.ru/ – Назва з екрану
30. Brent Laster, Jenkins 2: Up and Running: Evolve Your Deployment Pipeline for Next Generation Automation: O'Reilly Media, 2018
31. Getting started with Pipeline [Електронний ресурс]: – Режим доступу: https://www.jenkins.io/doc/book/pipeline/getting-started/ – Назва з екрану
32. Asciidoctor – A fast text processor & publishing toolchain for converting AsciiDoc to HTML5, DocBook & more [Електронний ресурс]: – Режим доступу: https://asciidoctor.org/ – Назва з екрану
33. IDEF0 [Електронний ресурс]: – Режим доступу: https://uk.wikipedia.org/wiki/IDEF0 – Назва з екрану
34. DAG [Електронний ресурс]: – Режим доступу: https://en.wikipedia.org/wiki/Directed_acyclic_graph – Назва з екрану
35. Sean P. Kane, Karl Matthias,Docker: Up & Running: Shipping Reliable Containers in Production: O'Reilly Media, 2018
36. Test Automation: Using Selenoid in the Docker Container [Електронний ресурс]: – Режим доступу: https://hackernoon.com/selenide-in-test-automation-through-selenoid-in-the-docker-container-ttw320f – Назва з екрану

ДОДАТОК А

1.1 Деталізація мети проекту методом SMART

Для конкретизації мети використаємо метод SMART. SMART – це мнемонічна абревіатура, що використовується в менеджменті та проектному управлінні для визначення цілей і постановки завдань. Аналіз за методом SMART наведено у таблиці А.1.

Таблиця А.1 − Деталізація мети методом SMART

Specific	Створення інформаційної технології для комплексної автоматизації web-сервісів на системному рівні
Measurable	Створити якісний програмний продукт, використовуючи наявні ресурси у обсязі, що не виходять за межі можливостей
Achievable	Створити програмний продукт на основі реально доступних ресурсних можливостей та практичного досвіду
Relevant	У наявності є всі необхідні технічні та програмні засоби Intellij IDEA, бібліотеки залежностей, доступ до інтернету
Time-bound	Програмний продукт створюється з обмеженням у часі на основі сформованого календарного плану

Specific (конкретна) – пояснює, що саме необхідно досягти.

Measurable (вимірювана) – пояснює, в чому буде вимірюватися результат. Якщо показник кількісний, то необхідно виявити одиниці виміру, якщо якісний, то необхідно виявити еталон відносин.

Achievable (досяжна) – пояснює, за рахунок чого планується досягти мети. І чи можливо її досягти взагалі.

Relevant (реалістична) – визначення істинності мети. Чи справді виконання даної задачі дозволить досягти бажаної мети? Необхідно впевнитися, що виконання даного завдання дійсно необхідно.

Time-framed (обмежена у часі) – визначення тимчасового тригера/проміжку, по настанню/закінченню якого повинна бути досягнута мета (виконана задача).

Після проведення аналізу методом SMART можна визначити кінцеву мету одним реченням: розробити інформаційну технологію автоматизації тестування API та web-сервісів на системному рівні, використовуючи наявні ресурси у обсязі, що не виходять за межі реально доступних апаратних та програмних можливостей, з обмеженням у часі на основі сформованого календарного плану.

Дана робота потребує вирішити наступні задачі:

- дослідити актуальність проблеми;
- провести аналіз практик організації та роботи з автоматизованими сценаріями;
- провести аналіз потреб інженерів із забезпечення якості програмного забезпечення для визначення критеріїв застосовності;
- сформулювати технічне завдання на розробку IT-продукту;
- спроектувати інформаційну систему;
- розробити інформаційну систему;
- провести тестування отриманої системи;
- розробити допоміжну документацію.

Практичне значення одержаних результатів полягає у тому, що запропонована бібліотека вирішить проблеми інженерів-автоматизаторів, такі, як:

- покриття додатку функціональними регресійними тестами;
- організація тестових сценаріїв у групи, задання пріоритетів;
- параметризація та повторне використання сценаріїв з різними наборами даних;
- механізм взаємодії з елементами управління в web-браузерах;
- управління типами web-браузерів та їх версіями;
- механізм запитів до API та верифікації відповідей;

- управління безперервною інтеграцією через динамічні скрипти конфігурацій розгортання проекту;
- оновлення та заміна компонентів системи (бібліотек-залежностей);
- спрощена підтримка та актуалізація тестів при зміні бізнес-логіки основного продукту;

1.2 Планування змісту структури робіт ІТ-проекту

1.2.1 Планування змісту структури робіт ІТ-проекту (WBS)

WBS – це ієрархічна структура, побудована з метою логічного розподілу усіх робіт з виконання проекту і подана у графічному вигляді. Це сукупність декількох рівнів, кожний з яких формується в результаті розподілу роботи попереднього рівня на її складові. Елементом найнижчого рівня є група робіт, або так званий робочий пакет (рис. А.1).

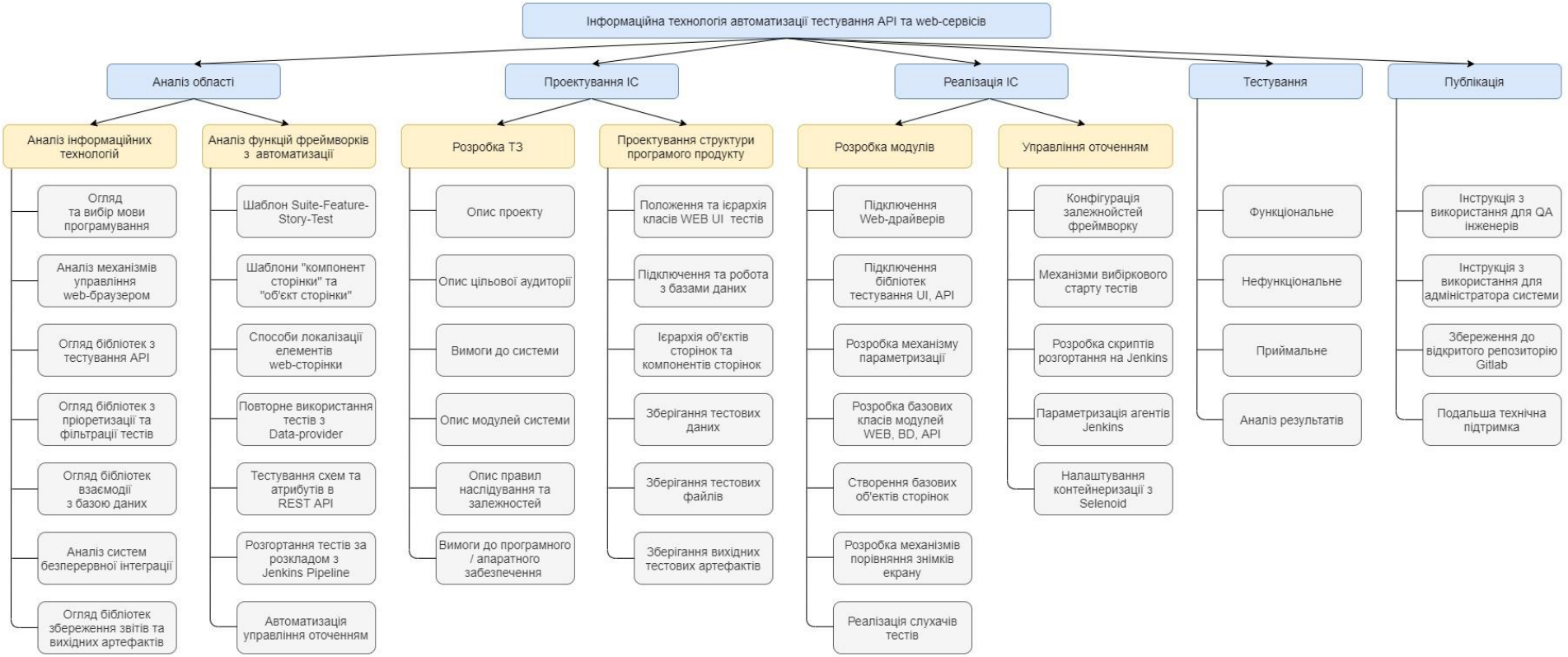

Рисунок А.1 – Ієрархічна структура робіт (WBS)

1.2.2 Планування структури організації

Організаційна структура проекту (OBS) є ієрархічною структурою управління проектом і показує відносини між учасниками проекту.

Організаційна структура проекту (OBS):

- створюється на рівні підприємства;
- дозволяє контролювати доступ користувачів до інформації відповідного рівня.

Елемент OBS, призначений на відповідний елемент – це керуючий проектом або відповідальний за всі роботи, які включаються в елемент (рис. А.2).

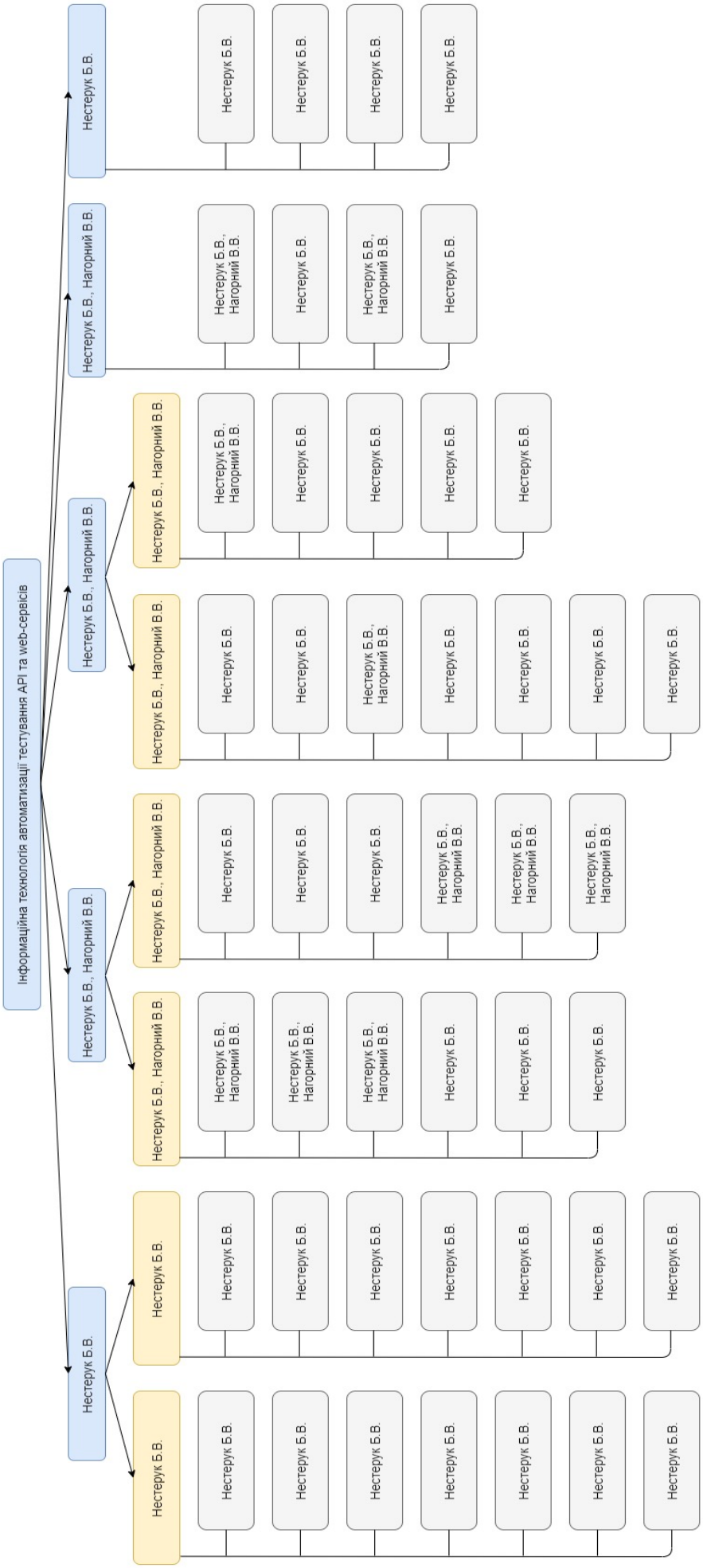

Рисунок А.2 – Організаційна структура проекту (OBS)

1.3 Побудова календарного графіку виконання ІТ – проекту

Діаграма Ганта (Gantt Chart) – інструмент для наочної ілюстрації календарного плану в проектному менеджменті. Діаграма Ганта надає графічну ілюстрацію розкладу та залежностей робіт, включаючи доступні ресурси розробників для подальшого планування, корегування та відстеження статусу конкретних завдань в проекті. Зліва від графіка розташований список завдань діяльності, а праворуч – шкала часу. Кожна дія представлено лінією, положення і довжина лінії відображає дату початку, тривалість і дату закінчення дії. Діаграма визначає завдання, які можуть виконуватися паралельно, і ті, які не можуть бути запущені або завершені до завершення інших завдань. Діаграма Ганта може допомогти виявити потенційні важкі місця і визначити завдання, які могли бути виключені з графіка проекту. Діаграму Ганта можна використовувати в управлінні проектами всіх розмірів і типів. Для побудови діаграми Ганта використовували програмний продукт GanttProject. Перелік завдань проекту згідно з ієрархічною структурою в вигляді діаграми Ганта показаний на рисунку А.3.

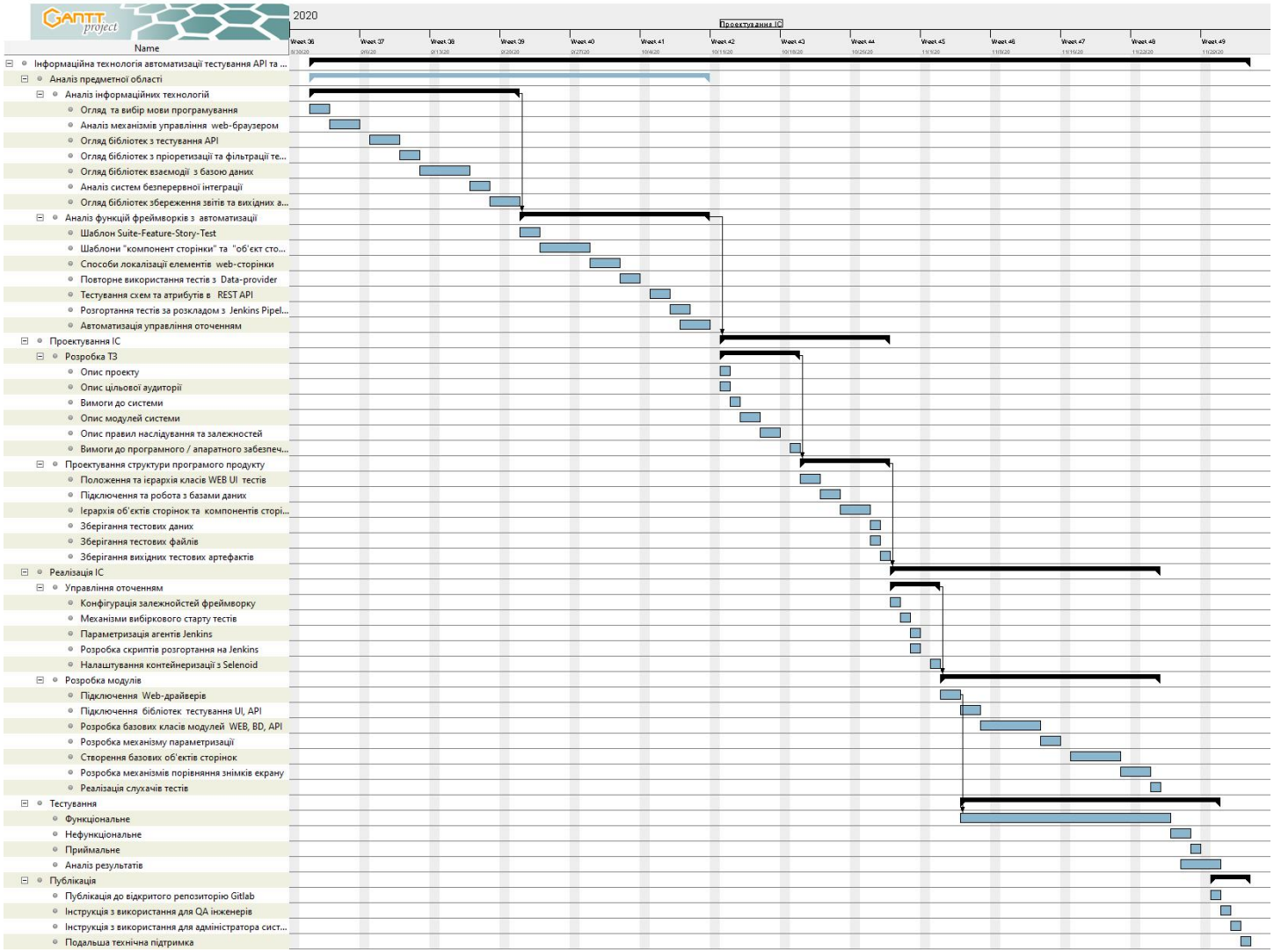

Рисунок А.3 – Діаграма Ганта

1.4 Планування ризиків проекту

При розробці будь-якого проекту необхідно передбачити всі можливі ризики, які можуть трапитись та провести необхідні заходи для їх уникнення чи зменшення збитків. Виділимо основні ризики під час розробки інформаційної системи:

1. вибір неоптимальних технологій;
2. поява альтернативного продукту;
3. ризик невірного технічного рішення;
4. недостатність кваліфікованого персоналу;
5. збій апаратного забезпечення при відсутності резервних копій даних;
6. виникнення незапланованих робіт та поява непередбачуваних витрат;
7. не знаходження спільної мови між керівником і виконавцем;
8. недотримання строків проекту;
9. втрата працездатності персоналу через сезонні хвороби та COVID-19;

Для класифікації ризиків використаємо шкалу ймовірності виникнення та величину втрат (таблиця А.2).

Таблиця А.2 – Оцінювання ймовірності виникнення та величини втрат

Оцінка	Ймовірність виникнення	Величина втрат
1	Слабоймовірна	Мінімальна
2	Малоймовірна	Низька
3	Ймовірна	Середня
4	Досить ймовірна	Висока
5	Майже можлива	Максимальна

Класифікуємо ризики згідно таблиці А.2 та розрахуємо індекс ризику за формулою $R = P_q \times I_q$, де:

R – індекс ризику (бали);

P_q – ймовірність виникнення ризиків відповідно до класифікації (бали);

I_q – величина втрат відповідно до класифікації ризику (бали).

Таблиця А.3 – Оцінка ймовірності виникнення, величини витрат та індексу ризику № Ризики Pq Iq R

№	Ризики	Pq	Iq	R
1	Вибір неоптимальних технологій	3	4	12
2	Поява альтернативного продукту	1	3	3
3	Ризик невірного технічного рішення	3	5	15
4	Недостатність кваліфікованого персоналу	1	4	4
5	Збій апаратного забезпечення при відсутності резервних копій	2	5	10
6	Виникнення незапланованих робіт та поява непередбачуваних витрат	4	2	8
7	Не знаходження спільної мови між керівником і виконавцем	2	5	10
8	Недотримання строків проекту	2	5	10
9	Втрата працездатності персоналу через сезонні хвороби чи COVID-19	2	4	8

На підставі отриманих значень індексу класифікуємо ризики: за ступенем впливу (таблиця А.4) та рівнем ризику (таблиця А.5).

Таблиця А.4 – Шкала оцінювання за ступенем впливу

№	Назва	Межі	Ризики що входять (табл. А.1.3)
1	Ігноруючі	$1 \leq R \leq 4$	2. Поява альтернативного продукту 4. Недостатність кваліфікованого персоналу
2	Незначні	$5 \leq R \leq 8$	6. Виникнення незапланованих робіт та поява непередбачуваних витрат 9. Втрата працездатності персоналу через сезонні хвороби чи COVID-19
3	Помірні	$9 \leq R \leq 10$	5. Збій апаратного забезпечення при відсутності резервних копій 7. Не знаходження спільної мови між керівником і виконавцем

			8. Недотримання строків проекту
4	Істотні	12≤R≤16	1. Вибір неоптимальних технологій 3. Ризик невірного технічного рішення
5	Критичні	20≤R≤25	Відсутні

Таблиця А.5 – Шкала оцінювання за рівнем ризику

№	Назва	Межі	Ризики що входять (табл. А.1.3)
1	Прийнятні	1≤R≤4	2, 4
2	Виправдані	5≤R≤10	5, 6, 7, 8, 9
3	Неприпустимі	12≤R≤25	1, 3

Критичні ризики згідно аналізу відсутні. Незначні, помірні, істотні потребують дій для зниження впливу та наслідків. Заплануємо активності:

- «6. Виникнення незапланованих робіт та поява непередбачуваних витрат» – ризик, що оцінки завдань проекту є неточними, або список завдань є неповним. Рішення: формування завдань розробки після повного формування та утвердження ТЗ;
- «9. Втрата працездатності персоналу через сезонні хвороби чи COVID-19» – ризик, що осінні загострення грипу та COVID-19 вплинуть на стан здоров’я розробника. Рішення: самоізоляція, робота з дому. Скорочення фізичних контактів з іншими людьми;
- «5. Збій апаратного забезпечення при відсутності резервних копій» – ризик повної або часткової втрати даних. Рішення – система контролю версій Git з репозиторієм в системах зберігання типу Gitlab, Github. Щоденне копіювання даних;
- «7. Не знаходження спільної мови між керівником і виконавцем» – ризик того, що очікування керівника відрізняються від реально розробленого продукту. Рушення: регулярні мітинги для синхронізації статусу, демонстрація розроблених модулей, teambuilding;

- «8. Недотримання строків проекту» – ризик не встигнути реалізувати усі функції системи до реальної дати здачі. Рішення: детальне планування завдань та фаз проекту. Розділення проекту на кілька релізів, доробка некритичних частин після видачі.
- «1. Вибір неоптимальних технологій» – ризик того, що обрана технологія не дозволить отримати необхідний результат. Для уникнення необхідно проводити детальний аналіз існуючих технологій, опираючись на предметну область та специфіку проекту;
- «3. Ризик невірного технічного рішення» – продукт не вирішує поставлені завдання, має неочікувану поведінку. Рішення: ретельне функціональне тестування на етапі реалізації. Детальна проробка ТЗ.

ДОДАТОК Б

Дана технологія використовується інженерами із забезпечення якості програмних веб-орієнтованих продуктів. Вона включає інструменти з розробки та виконання тестових сценаріїв для взаємодії з веб-сервісами на системному рівні через графічний інтерфейс користувача, REST API та інтерфейс бази даних.

Найменування: “Інформаційна технологія автоматизації тестування API та web-сервісів”.

Об’єкт, в якому використовують технологію: команда розробки та забезпечення якості веб-сервісів.

Б.1 ПІДСТАВА ДЛЯ РОЗРОБКИ

Розробка ведеться на основі дипломної роботи. Найменування організації: група ІТ.мз-91с, кафедра комп’ютерних наук, секція ІТП. Тема проекту: ”Інформаційна технологія автоматизації тестування API та web-сервісів”.

Б.1.1 Документи, на основі яких ведеться проектування

- Програмні засоби ЕОМ. Забезпечення якості. Терміни та визначення. ДСТУ 2844-94
- Програмні засоби ЕОМ. Показники і методи оцінювання якості. ДСТУ 2850-94.
- Програмні засоби ЕОМ. Документування результатів випробувань. ДСТУ 2851-94.
- Програмні засоби ЕОМ. Підготовлення і проведення випробувань. ДСТУ 2853-94.
- Системи оброблення інформації. Програмування. Терміни та визначення. ДСТУ 2873-94.
- Стандарти з управління якістю та забезпечення якості. ДСТУ 1809000-98.

- Системи оброблення інформації. Розроблення систем. Терміни та визначення. ДСТУ 2941-94.
- Інформаційні технології. Процеси життєвого циклу програмного забезпечення ДСТУ 3918-99.

Б.2 ПРИЗНАЧЕННЯ І МЕТА СТВОРЕННЯ ТЕХНОЛОГІЇ

Інформаційна технологія повинна представляти собою каркасний проект (фреймворк) у вигляді бібліотеки з відкритим програмним кодом та забезпечувати:

- розробку тестових сценаріїв для взаємодії з графічним інтерфейсом веб-сервісу у вікні веб-браузера;
- розробку тестових сценаріїв для взаємодії з відкритим програмним інтерфейсом REST API;
- розробку тестових сценаріїв для взаємодії з інтерфейсом бази даних веб-сервісу;
- параметризацію тестових методів та можливість зміни тестових даних незалежно від коду тестів;
- організацію тестових наборів, фільтрацію та виключення окремих тестів з набору;
- параметризований запуск тестової сесії на сервері безперервної інтеграції;
- перегляд звітів тестових сесій та деталей помилки окремих тестів;

Б.2.1 Мета створення інформаційної технології

Забезпечення швидкого старту автоматизації функціональних системних тестувань при розробці веб-орієнтованих програмних продуктів.

Б.2.2 Цільова аудиторія

У цільовій аудиторії можна виділити наступні групи: інженери із забезпечення якості, розробники програмних продуктів, менеджери проектів.

Б.3 ВИМОГИ ДО ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

Б.3.1 Загальні вимоги

Технологія має бути реалізована у вигляді java-бібліотеки та опублікована в публічних репозиторіях артефактів. Технологія повинна підтримувати версійність артефакту: при оновленні функцій попередня версія повинна бути доступною для завантаження та використання. Для кожної нової версії обов’язковим є перелік доданих функцій та функцій, що змінилися/не підтримуються.

Технологія не повинна вимагати від користувача встановлення додаткових інструментів чи бібліотек окрім віртуальної машини Java (JVM) та середовища розробки (IDE).

Б.3.2 Функціональні вимоги

Конфігурація залежностей та життєвого циклу здійснюється за допомогою систем автоматизованого збирання проектів. Окремі завдання повинні бути реалізовані для завантаження тестових артефактів, вхідних файлів даних та параметрів тестових методів.

Для кожного виду тестування (WEB UI / REST API / DB) створенні окремі модулі проекту. Структура пакетів тестів та класів реалізації взаємодії повинна бути уніфікована.

Параметризація усіх видів тестів відбувається засобами одного механізму провайдера даних. Технологія повинна підтримувати файли тестових параметрів у форматах JSON, XLSX. Потрібно реалізувати можливість підтримки додаткових форматів шляхом реалізації універсального інтерфейсу зчитування даних.

Для спрощення підтримки та реалізації, тестові сценарії оперують лише термінами бізнес-логіки та не розкривають механізмів взаємодії з елементами сторінок.

Технологія підтримує як локальний запуск тестових наборів, так і виконання тестових сесій за розкладом або за вимогою на серверах безперервної інтеграції.

Логування тестових дії, часу та тривалості тестової ітерації повинно бути наскрізним та реалізовано по за межами тестових сценаріїв.

Б.3.3. Вимоги до видів забезпечення

Вимоги до інформаційного забезпечення

Реалізація технології відбувається з використанням технологій та інструментів:

- Groovy;
- Gradle;
- Spock Framework;
- GEB;
- REST Assured;
- DB Unit;
- Selenium Webdrvier;
- Selenoid;
- Docker;
- Jenkins Pipeline;
- Allure Report;

Усі необхідні бібліотеки завантажуються автоматично за допомогою інструменту збирання проектів Gradle. Файл-обгортка для завантаження та установки Gradle знаходиться в репозиторії поряд з вихідним кодом розробленої технології.

Вимоги до програмного забезпечення

Програмне забезпечення робочої станції інженера-автоматизатора повинне задовольняти наступним вимогам:

- веб-браузер: Google Chrome, Mozilla Firefox, Safari, IE;
- компілятор: Java 8+;
- інтерактивна середа розробки (рекомендована): Intellij IDEA Community;

Вимоги до апаратного забезпечення

Апаратне забезпечення робочої станції інженера-автоматизатора повинне задовольняти наступним вимогам:

- не менше 5 ГБ вільного місця на диску;

- не менше 8 Гб оперативної пам’яті;
- не менше 8 ядер головного процессора.

Б.3.4 Публікація та використання

Вихідний код розроблюваної технології публікується у відкритому репозиторії Gitlab та доступний для використання за ліцензією Apache License 2.0. Режим доступу: https://gitlab.com/bohdan_nesteruk/masters_curiosity

ДОДАТОК В

api.service.BaseApiSpec

```
package api.service
import com.github.fge.jsonschema.cfg.ValidationConfiguration
import com.github.fge.jsonschema.main.JsonSchemaFactory
import io.restassured.RestAssured
import io.restassured.builder.RequestSpecBuilder
import io.restassured.http.ContentType
import io.restassured.specification.RequestSpecification
import io.restassured.specification.ResponseSpecification
import org.codehaus.groovy.runtime.StackTraceUtils
import org.spockframework.runtime.model.FeatureInfo
import org.spockframework.runtime.model.SpecInfo
import spock.lang.Shared
import spock.lang.Specification
import util.Environment
import util.Helper
import util.TestDataUtil
import util.data.TestUser
import java.text.SimpleDateFormat
import java.time.LocalDateTime
import java.time.ZoneId
import static com.github.fge.jsonschema.SchemaVersion.DRAFTV4
import static io.restassured.RestAssured.expect
import static io.restassured.config.JsonConfig.jsonConfig
import static io.restassured.path.json.config.JsonPathConfig.NumberReturnType.BIG_DECIMAL
import static util.UserUtil.getTestUsers

class BaseApiSpec extends Specification {
    @Shared
    ResponseSpecification responseSpec
    @Shared
    RequestSpecification requestSpec
    @Shared
    JsonSchemaFactory jsonSchemaFactory
    @Shared
    protected String suiteId
    @Shared
    protected LocalDateTime runDateTime
    @Shared
    private List<TestUser> users = []
    @Shared
    protected String suiteName, specName, runDate, runTime
    @Shared
    public Map<String, Object> specData
    def setupSpec() {
        loadSpecData()
        users = getTestUsers()
        Class className = getClass()
        suiteName = className.package.name[(className.package.name.lastIndexOf(".") +
1)..-1]
        specName = className.name[(className.name.lastIndexOf(".") + 1)..-1]
        String dateFormat = Environment.instance.get 'runDateFormat'
        String timeFormat = Environment.instance.get 'runTimeFormat'
        Date now = new Date()
        runDateTime = LocalDateTime.ofInstant(now.toInstant(), ZoneId.systemDefault())
        runDateTime = runDateTime.minusMinutes(1)
        runDate = new SimpleDateFormat(dateFormat).format(now)
```

```
        runTime = new SimpleDateFormat(timeFormat).format(now)
        suiteId = Helper.randomId()
        RestAssured.baseURI = Environment.instance.get 'baseAPIUrl'
        requestSpec = new RequestSpecBuilder().build().
                config(RestAssured.config()
                        .jsonConfig(jsonConfig().numberReturnType(BIG_DECIMAL))).
                log().everything()
        responseSpec = expect().
                statusCode(200).and().
                contentType(ContentType.JSON)

        jsonSchemaFactory = JsonSchemaFactory.newBuilder()
                .setValidationConfiguration(
                        ValidationConfiguration.newBuilder().
                                setDefaultVersion(DRAFTV4).freeze()).freeze()
    }
    protected getMethodName() {
        StackTraceUtils.sanitize(new Throwable()).stackTrace[2].methodName
    }
    protected List<Map<String, Object>> getTestData() {
        getTestData(getMethodName())
    }
    protected List<Map<String, Object>> getTestData(String methodName) {
        FeatureInfo spec = specificationContext.currentSpec.features.find {
            FeatureInfo info ->
                info.dataProviders.any {
                    it.dataProviderMethod.name == methodName
                }
        }
        specData.get(spec.name) as List<Map<String, Object>>
    }
    Map<String, Object> getSetupSpecData() {
        return specData.get('setupSpec') as Map<String, Object>
    }
    void loadSpecData() {
        SpecInfo currentSpec = specificationContext.currentSpec
        specData = TestDataUtil.loadSpecData(currentSpec)
    }
}
```

ui.suite.BaseE2ESpec

```
package ui.suite
import geb.spock.GebReportingSpec
import org.codehaus.groovy.runtime.StackTraceUtils
import org.openqa.selenium.Dimension
import org.openqa.selenium.Point
import org.spockframework.runtime.model.FeatureInfo
import org.spockframework.runtime.model.SpecInfo
import spock.lang.Shared
import ui.page.ApplicationPages
import util.Environment
import util.Helper
import util.TestDataUtil
import util.data.TestUser
import java.text.SimpleDateFormat
import java.time.LocalDateTime
import java.time.ZoneId
import static util.UserUtil.getTestUsers
class BaseE2ESpec extends GebReportingSpec {
```

```
    @Shared
    protected ApplicationPages pages
    @Shared
    protected String suiteId
    @Shared
    protected LocalDateTime runDateTime
    @Shared
    private List<TestUser> users = []
    @Shared
    protected String suiteName, specName, runDate, runTime
    @Shared
    public Map<String, Object> specData
    def setupSpec() {
        loadSpecData()
        users = getTestUsers()
        Class className = getClass()
        suiteName = className.package.name[(className.package.name.lastIndexOf(".") +
1)..-1]
        specName = className.name[(className.name.lastIndexOf(".") + 1)..-1]

        String dateFormat = Environment.instance.get 'runDateFormat'
        String timeFormat = Environment.instance.get 'runTimeFormat'
        Date now = new Date()
        runDateTime = LocalDateTime.ofInstant(now.toInstant(), ZoneId.systemDefault())
        runDateTime = runDateTime.minusMinutes(1)
        runDate = new SimpleDateFormat(dateFormat).format(now)
        runTime = new SimpleDateFormat(timeFormat).format(now)
        suiteId = Helper.randomId()
        pages = to ApplicationPages
    }
    protected getMethodName() {
        StackTraceUtils.sanitize(new Throwable()).stackTrace[2].methodName
    }
    protected List<Map<String, Object>> getTestData() {
        getTestData(getMethodName())
    }
    protected List<Map<String, Object>> getTestData(String methodName) {
        FeatureInfo spec = specificationContext.currentSpec.features.find {
            FeatureInfo info ->
                info.dataProviders.any {
                    it.dataProviderMethod.name == methodName
                }
        }
        specData.get(spec.name) as List<Map<String, Object>>
    }
    Map<String, Object> getSetupSpecData() {
        return specData.get('setupSpec') as Map<String, Object>
    }
    void loadSpecData() {
        SpecInfo currentSpec = specificationContext.currentSpec
        specData = TestDataUtil.loadSpecData(currentSpec)
    }
    def cleanupSpec() {
        browser.quit()
    }
    void manageWindowSize(int w, int h, int x = 0, int y = 0) {
        driver.manage().window().size = new Dimension(w, h)
        driver.manage().window().position = new Point(x, y)
    }
}
```

util.extension.listener.RunListener

```
package util.extension.listener
import groovy.time.TimeCategory
import groovy.time.TimeDuration
import io.qameta.allure.Allure
import org.spockframework.runtime.AbstractRunListener
import org.spockframework.runtime.model.ErrorInfo
import org.spockframework.runtime.model.FeatureInfo
import org.spockframework.runtime.model.IterationInfo
import org.spockframework.runtime.model.SpecInfo
import util.Helper
import util.extension.recorder.OutputCapture
import util.extension.recorder.VideoRecorder
class RunListener extends AbstractRunListener {
    VideoRecorder videoRecorder
    OutputCapture outputCapture
    boolean testFailed = false
    void beforeSpec(SpecInfo spec) {
        Helper.log "[BEFORE SPEC]: ${spec.name}"
        videoRecorder = new VideoRecorder()
        outputCapture = new OutputCapture()
        outputCapture.captureOutput()
    }
    void beforeFeature(FeatureInfo feature) {
        Helper.log "[BEFORE FEATURE]: ${feature.name}", 2
    }
    Date timeStart
    void beforeIteration(IterationInfo iteration) {
        testFailed = false
        timeStart = new Date()
        videoRecorder.start(iteration.name)
        outputCapture.resetOutput()
        Helper.log "[BEFORE ITERATION]: ${iteration.name}", 4
        iteration.dataValues?.each {
            (it as Map)?.each {
                Helper.log String.valueOf(it), 8
            }
        }
    }
    void afterIteration(IterationInfo iteration) {
        def timeStop = new Date()
        TimeDuration duration = TimeCategory.minus(timeStop, timeStart)
        if (!testFailed) {
            Helper.log "[TEST ITERATION PASSED]: ${iteration.name} | Duration: $duration",
4
            videoRecorder.kill()
        }
    }
    void error(ErrorInfo error) {
        testFailed = true
        Helper.log "[LAST MESSAGE RECEIVED FROM TEST ITERATION]", 4
        error.getException().printStackTrace()
        try {
            def file = videoRecorder.stop()
            if (file?.exists()) {
                Allure.addAttachment(file.name, "video/avi", new FileInputStream(file),
"mov")
            }
            videoRecorder.kill()
        } catch (Exception e) {
            Helper.log "Error attaching video file at ${error.method.feature?.name}:
```

```
${e.getMessage()}", 4
            e.printStackTrace()
        }
        Allure.addAttachment("${error.method?.feature?.name}_system_out", "text/html",
outputCapture.toStringAsNewLines(), "txt")
    }
    void afterFeature(FeatureInfo feature) {
        Helper.log "[AFTER FEATURE]: ${feature.name}", 2
    }
    void afterSpec(SpecInfo spec) {
        Helper.log "[AFTER SPEC]: ${spec.name}"
        videoRecorder.kill()
        outputCapture.releaseOutput()
    }
}
              }
    }
    static String cutEnd(String data, int maxLength) {
        if (data.length() > maxLength) {
            return data.substring(0, maxLength) + "..."
        } else {
            return data
        }
    }
    static String cutBegin(String data, int maxLength) {
        if (data.length() > maxLength) {
            return "..." + data.substring(data.length() - maxLength, data.length())
        } else { return data
        }
    }
    static String getParametersAsString(Object[] parameters, int maxLength) {
        if (parameters == null || parameters.length == 0) {
            return "()"
        }
        StringBuilder builder = new StringBuilder()
        builder.append("(")
        for (int i = 0; i < parameters.length; i++) {
            builder.append(arrayToString(parameters[i]))
            if (i < parameters.length - 1) {
                builder.append(", ")
            }
        }
        return cutEnd(builder.toString(), maxLength) + ")"
    }
    static Object arrayToString(Object obj) {
        if (obj != null && obj.getClass().isArray()) {
            int len = Array.getLength(obj)
            String[] strings = new String[len]
            for (int i = 0; i < len; i++) {
                strings[i] = String.valueOf(Array.get(obj, i))
            }
            return Arrays.toString(strings)
        } else {            return obj
        }
    }
}
```

util.TestDataReader

```
package util
interface TestDataReader {
}
```

util.TestDataUtil

```
package util
import com.google.gson.Gson
import com.google.gson.reflect.TypeToken
import org.spockframework.runtime.model.SpecInfo
import java.nio.file.Path
import java.nio.file.Paths
class TestDataUtil implements TestDataReader {
    private static final String UNICODE_CHARSET = 'UTF-8'
    private static final String DATA_FOLDER = 'testData'
    private static final Gson gson = new Gson()
    static Map<String, Object> loadSpecData(SpecInfo currentSpec) {
        Map<String, Object> data = [:]
        String fileName = currentSpec.filename.replace('groovy', 'json')
        String testPkgName = currentSpec.package.split('\\.').first()
        String suitePkgName = currentSpec.package.split('\\.').last()
        if (getDataFile(testPkgName, suitePkgName, fileName)?.exists()) {
            data = getJsonMappedData(testPkgName, suitePkgName, fileName)
        }
        return data
    }
    static <T> Map<String, T> getJsonMappedData(
            String pkgName, String suiteName, String fileName) {
        String testDataJson =
                getDataFile(pkgName, suiteName, fileName).getText(UNICODE_CHARSET)
        return gson.fromJson(testDataJson, new TypeToken<Map<String, T>>() {}.type)
    }
    static File getDataFile(String testPkgName, String suiteName, String fileName) {
        String dataResPath =
                Paths.get(DATA_FOLDER, testPkgName, suiteName, fileName).toString()
        URL dataResURL = TestDataUtil.getClassLoader().getResource(dataResPath)
        return dataResURL ? new File(dataResURL.toURI()) : null
    }
    final static String TEST_FILES_RESOURCE_DIR = "files"

    static File getResourceFile(String pathInResourceDir) {
        return new File(TestDataUtil.classLoader.getResource(
                Paths.get(TEST_FILES_RESOURCE_DIR, pathInResourceDir).toString()).toURI())
    }
    static Map<String, Object> loadSpecData(SpecInfo currentSpec, List<String>
excludedPkgParts) {
        String fileName = currentSpec.filename.replace('groovy', 'json')
        List<String> allPkgParts = currentSpec.package.split('\\.').toList()
        Path dataFilePath = Paths.get(*(allPkgParts - excludedPkgParts)).resolve(fileName)
        File dataFile = getDataFile(dataFilePath)
        return dataFile?.exists() ? getJsonMappedData(dataFile) : [:]
    }
    static File getDataFile(Path path) {
        String dataResPath = Paths.get(DATA_FOLDER).resolve(path).toString()
        URL dataResURL = TestDataUtil.getClassLoader().getResource(dataResPath)
        return dataResURL ? new File(dataResURL.toURI()) : null
    }

    static <T> Map<String, T> getJsonMappedData(File file) {
        String testDataJson = file.getText(UNICODE_CHARSET)
        return gson.fromJson(testDataJson, new TypeToken<Map<String, T>>() {}.type)
    }
}
```