МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Порівняльне дослідження тестових фреймворків JUnit 5 та TestNG для організації тестування Java-застосунків**

Виконала: студентка 4 курсу, групи 6.1210-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
В.С. Шевцова
(ініціали та прізвище)
Керівник к.ф.-м.н., доцент, доцент кафедри ЕІС та ПЗ
Г.П. Коломоєць
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дискус»
Р.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти _____перший (бакалавський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Програмне забезпечення систем_____
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Тетяна КРИТСЬКА
"_01_" __березня_____2024_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Шевцовій Ванесі Сергіївні
(прізвище, ім'я, по батькові)

1.Тема роботи: **Порівняльне дослідження тестових фреймворків JUnit 5 та TestNG для організації тестування Java-застосунків**

керівник роботи ___Коломоєць Геннадій Павлович_____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від ___26.12.2023 № 2215-с

2. Строк подання студентом кваліфікаційної роботи_____07.06.2024_____
3. Вихідні дані кваліфікаційної роботи
- комплект нормативних документів;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд існуючих рішень та аналогів;
- дослідження можливостей обраних фреймворків;
- перелік можливостей модульного тестування для порівняння;
- проектування та розробка модульних тестів та їх опис;
- тестування обраних застосунків та формування висновків і практичних рекомендацій.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_____слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата
		Завдання прийняв

7. Дата видачі завдання 01.03.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів бакалаврської роботи	Примітка
1	Аналіз предметної області	02.02 – 16.02.24	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	17.02 – 18.02.24	виконано
3	Огляд та збір літератури стосовно теми кваліфікаційної роботи	24.02 – 14.03.24	виконано
4	Аналіз існуючих засобів модульного тестування Java-застосунків	15.03 – 20.03.24	виконано
5	Огляд функціональності фреймворків JUnit 5 та TestNG, їх архітектури та життєвих циклів тестів	24.03 – 12.04.24	виконано
6	Огляд застосунків, за допомогою яких буде виконуватись порівняння	13.04 – 24.04.24	виконано
7	Проектування модульних тестів для обох застосунків	26.04 – 01.05.24	виконано
8	Розробка спроектованих тестів для обраних застосунків	02.05 – 16.05.24	виконано
9	Апробація отриманих результатів тестування на обох фреймворках	17.05 – 29.05.24	виконано
10	Оформлення звіту	30.05 – 12.06.24	виконано

Студент _____ Шевцова В.С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Коломоєць Г.П.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок – 82,

Рисунків – 36,

Таблиць – 3,

Лістинги – 16,

Джерел – 20.

Шевцова В.С. Порівняльне дослідження тестових фреймворків JUnit 5 та TestNG для організації тестування Java-застосунків: кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник Г.П. Коломоєць. Запоріжжя: ЗНУ, 2024. 82 с.

Метою роботи є порівняльний аналіз популярних тестових фреймворків для мови програмування Java — JUnit 5 та TestNG з метою визначення їх ефективності при тестуванні Java-застосунків.

У процесі дослідження тестових фреймворків був виконаний детальний аналіз основних характеристик та функціональних можливостей JUnit 5 та TestNG, включаючи підтримку анотацій, управління тестовими наборами, керування життєвим циклом тестів та інші аспекти. Для обраних Java-застосунків були спроектовані, розроблені та виконані тести, для яких порівнювалася ефективність окремих компонентів тестових фреймворків. За результатами роботи напрацьовані рекомендації щодо використання тестових фреймворків JUnit 5 та TestNG для різних сценаріїв.

Ключові слова: *JUnit 5, TestNG, ефективність, модульні тести, java-застосунки, порівняння, тестування, працездатність, коректність.*

ABSTRACT

Pages – 82,

Drawings – 36,

Tables – 3,

Listings – 16,

Sources – 20.

Shevtsova V.S. Comparative Study of JUnit 5 and TestNG Test Frameworks for Organizing Java Application Testing: bachelor's thesis in specialty 121 «Software engineering» / Science manager H. P. Kolomoets. Zaporizhzhia: ZNU, 2024. 82 p.

The purpose of the work is a comparative analysis of popular test frameworks for the Java programming language — JUnit 5 and TestNG in order to determine their effectiveness in testing Java applications.

In the process of researching test frameworks, a detailed analysis of the main features and functionality of JUnit 5 and TestNG was performed, including annotation support, test suite management, test life cycle management, and other aspects. For the selected Java applications, tests were designed, developed and performed, for which the effectiveness of individual components of the test frameworks was compared. Based on the results of the work, recommendations were developed for the use of JUnit 5 and TestNG test frameworks for various scenarios.

Keywords: *JUnit 5, TestNG, efficiency, unit tests, java applications, comparison, testing, performance, correctness.*

ЗМІСТ

ВСТУП 8

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 13

1.1 Тестування як технологія забезпечення якості застосунків 13

1.2 Поняття модульного тестування програмного забезпечення і можливості які воно надає 16

1.3 Сучасні фреймворки модульного тестування Java-застосунків 22

1.4 Постановка завдання 24

1.5 Висновки до розділу 1 25

2 ПРОЕКТУВАННЯ ТЕСТОВИХ СЦЕНАРІЇВ 26

2.1 Огляд функціональності фреймворків JUnit 5 та TestNG 26

2.1.1 Архітектура і життєві цикли тестів у JUnit 5 26

2.1.2 Архітектура і життєві цикли тестів у TestNG 28

2.1.3 Огляд характеристик та особливостей JUnit 5 та TestNG 29

2.2 Огляд застосунків, за допомогою яких буде виконуватись порівняння можливостей обраних фреймворків 35

2.2.1 Функціональність та структура застосунка OnlineTest 35

2.2.2 Сценарії використання застосунка OnlineTest 38

2.2.3 Функціональність та структура застосунка PortScanner 41

2.2.4 Сценарії використання застосунка PortScanner 44

2.3 Проектування модульних тестів для обраних застосунків 46

2.4 Висновки до розділу 2 49

3 РОЗРОБКА ТА ВИКОНАННЯ ТЕСТІВ ЗАСОБАМИР JUNIT 5 ТА TESTNG 50

3.1 Використання можливостей фреймворків при тестуванні обраних застосунків 50

3.1.1 Тестування застосунка OnlineTest 50

3.1.2 Тестування застосунка PortScanner 58
3.2 Аналіз результатів проведеного порівняльного дослідження 73
3.3 Висновки до розділу 3 77
ВИСНОВКИ 79
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 81

ВСТУП

Актуальність теми

Розробку сучасного програмного забезпечення (ПЗ), такого як мобільні застосунки, веб-застосунки, десктопні програми, ігри та інше не можливо уявити без належного тестування на кожному з етапів розробки. Тестування відіграє критичну роль у кожному життєвому циклі програмного забезпечення і має безліч важливих функцій, а саме таких як:

- Перевірка відповідності ПЗ поставленим вимогам і очікуваним діям та результатам роботи.
- Перевірка окремих функцій та модулів в ізоляції від інших об’єктів.
- Оцінка продуктивності розроблювальної системи.
- Визначення сумісності компонентів програмного забезпечення, безпеки даних.
- Перевірка програми на наявність нових помилок після внесення змін тощо.

Актуальність обраної теми вбачається в тому, що фреймворки, порівняння яких здійснюється, майже в однаковій мірі популярні по сьогодні, і до того ж за своєю функціональністю вони дуже схожі, тож тестувальники-початківці і розробники які хочуть самі тестувати свої застосунки на Java можуть замислюватись над тим, чи є між ними різниця, за яким критерієм обирати будь-який з них для роботи, і якщо відмінності є, то в чому саме? Порівняльне дослідження обраних тестових фреймворків може відповісти на ці питання.

Мета дослідження

Метою роботи є порівняльний аналіз популярних тестових фреймворків для мови програмування Java — JUnit 5 та TestNG з метою визначення їх ефективності при тестуванні Java-застосунків.

Завдання дослідження

Завданням дослідження є розгляд гнучкості, продуктивності досліджуваних тестових фреймворків та можливості їх інтеграції з іншими інструментами. Також необхідно було визначити архітектурні особливості кожного фреймворку, їх легкість або складність в освоєнні, та підтримку тестів різноманітних типів.

Об’єкт дослідження

Об’єктом дослідження є тестові фреймворки JUnit 5 та TestNG для організації тестування Java-застосунків.

Предмет дослідження

Зіставлення можливостей, які надають обрані фреймворки, збір їх сильних та слабких сторін, порівняння характеристик, архітектурних рішень та життєвих циклів.

Методи дослідження

В роботі використані наступні методи дослідження: аналіз літературних джерел, порівняння функціоналу, аналіз особливостей фреймворків JUnit 5 та TestNG при організації тестування Java-застосунків.

Апробація одержаних результатів

Результати дослідження були представлені на XVII науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених Запорізького національного університету «Молода наука-2024» [1].

Глосарій

Тестувальник (QA-інженер, інженер забезпечення якості) — це спеціаліст, який виконує тестування програмного забезпечення та його

компонентів. Мета його роботи полягає у пошуку помилок, недоліків та невідповідності ПЗ вимогам і запобігання виходу неякісного програмного продукту на ринок.

Тестовий фреймворк — платформа або структура рішень та інструментів, що спрощує та оптимізує процес тестування програм.

JUnit 5 — один з найвідоміших та частовживаних фреймворків для тестування програм на Java, версія якого має ряд нововведень та покращень у порівнянні з попередніми версіями JUnit.

TestNG — фреймворк для тестування Java-застосунків, не менш популярний ніж JUnit, але відомий за більш розширенні можливості у порівнянні з ним.

Автоматизоване тестування — використання автоматизації при виконанні тестів з метою зменшення ручної роботи та підвищення ефективності та швидкості тестування.

Параметризований тест — це тест, який виконується декілька разів підряд, приймаючи і використовуючи параметри для свого виконання у вигляді різних наборів вхідних даних.

Модульне тестування — тестування окремих, найменших модулів програми в ізоляції від інших модулів з метою перевірки їх роботи на коректність.

Інтеграційне тестування — тестування, при якому перевіряється взаємодія між різними компонентами програми для забезпечення їх правильної роботи у комбінації, тобто разом. Його виконання передбачено одразу після модульного тестування, а вже після виконують верифікацію та валідацію програмного продукту.

Валідація (англ. Software Validation) — перевірка фактичного (кінцевого) програмного забезпечення на відповідність встановленим вимогам до нього, очікуванням та вимогам клієнта, системи або платформи, для якої розроблено програму. Для такої перевірки необхідно виконувати програму.

Верифікація — відбувається до виконання валідації ПЗ, вона націлена на

перевірку того, як програма спроектована, чи були дотримані при розробці такі аспекти як зазначений дизайн, специфікації вимог, відповідність програми своєму опису та призначенню. При верифікації виконання (запуск) програми не виконується.

Функціональне тестування — перевірка функціоналу програми висунутим вимогам та специфікаціям до нього.

Нефункціональне тестування — оцінка продуктивності, надійності, безпеки, сумісності, користувацького інтерфейсу, локалізації.

Регресійне тестування — процес пошуку нових помилок після внесення змін у код ПЗ або після виправлення попередніх багів та дефектів у ньому.

Анотації — це мітки, які визначають, як тести будуть оброблені, у якому порядку, яку кількість разів тощо. Їх вказують перед тестами, до яких безпосередньо потрібно застосувати мітку.

Фікстури — методи або функції, з допомогою яких можна підготувати наявне середовище до запуску тестів, а також очистити пам’ять після їх виконання. Дозволяє ініціалізувати об’єкти один раз для кожного тесту з метою запобігання дублювання коду, фіксувати певний стан об’єктів, необхідний для якогось тестового сценарію тощо.

Припущення — умова або обмеження, яке являє собою передбачення про середовище, умови використання програми або компонента, наявність певних ресурсів або обмеження на функціональність і передбачає перевірку отриманих результатів на відповідність очікуваним.

Твердження — умова або очікування щодо функціоналу програми, яку можна перевірити шляхом виконання тестових випробувань. Може бути виражено в формі "Якщо X то Y", "Після A має статися B", "Система повинна X".

Тестовий набір — набір тестів що відносяться до певного модуля, або які тестують одну конкретну функціональність або просто ті які поєднані в одну категорію. Передумови та постумови у тестах з набору можуть бути пов’язані між собою.

Тестовий сценарій (англ. Test Case) — це детальний опис послідовності дій, які потрібно виконати для перевірки функціональності програмного забезпечення. Він визначає вхідні умови, передумови, кроки виконання тесту, очікувані результати, можливі вихідні результати та постумови тесту. Сценарії пишуться на основі тестових вимог.

Позитивне тестування — процес перевірки програмного забезпечення з метою підтвердження його правильної роботи в умовах, коли тестувальник вводить правильні вхідні дані, виконує логічні та послідовні кроки, і очікується, що програма відповідатиме згідно очікувань.

Негативне тестування — процес пошуку помилок, недоліків та вразливостей у системі, що тестується, з допомогою некоректних сценаріїв та вхідних даних для перевірки системи на предмет обробки помилок під час роботи у неналежних для неї умовах.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Тестування як технологія забезпечення якості застосунків

Очевидно, що програмне забезпечення тісно переплітається з усіма сферами нашого життя. Це забезпечення програмами комп'ютерів, мобільних пристроїв, інтернету, соціальних мереж, веб-сайтів та інших цифрових платформ, якими ми користуємось щодня. Розробка програмного забезпечення стає у нагоді, коли потрібно впровадити програму для медичної діагностики, медичного обладнання, пристроїв моніторингу, банківської системи, навігації, планування маршрутів, автопілотів, впровадження навчальних платформ, у тому числі інтерактивних, автоматизація виробничих процесів, ПЗ для проектування та моделювання, розробка відеоігор, програм та ігор з підтримкою віртуальної реальності, електрона комерція тощо. Все це можливо завдяки розробці і впровадженню у перелічені сфери програмного забезпечення, тож необхідність забезпечення якості цих застосунків стає все більш очевидною. Тестування програм допомагає забезпечити надійність, функціональність та безпеку програмного забезпечення, без якого ми вже не можемо уявити своє життя сьогодні.

Сама якість програмних продуктів не обмежується лише відсутністю помилок або відмінною роботою окремих функцій в них. Якість включає в себе також відповідність вимогам користувача, ефективність, зручність використання, швидкість реакції та безпеку даних. Тож, тестування не лише спрямоване на виявлення дефектів, а й на забезпечення того, що програмне забезпечення відповідає очікуванням користувачів і функціонує належним чином за будь-яких умов (якщо це можливо забезпечити та реалізувати).

Тож за визначенням тестування передбачає перевірку програмного забезпечення на відповідність вимогам та очікуванням (клієнта і користувачів), а також на виявлення помилок і дефектів. Це інструмент, з допомогою якого QA-фахівець (або сам розробник, якщо він працює один або в команду не

залучено тестувальника) може забезпечити якість та надійність програмного продукту перед його випуском у виробництво або в експлуатацію.

Тестування визначається своїми принципами, такими як [2]:

1. Тестування відображає наявність дефектів: Підтвердити або довести відсутність проблем воно не може. Тож головна задача полягає у розробці тест-кейсів, які будуть знаходити якомога більше помилок та дефектів в програмі. Невдала спроба знайти помилки в програмі також не є аргументом відсутності помилок. Натомість гарне покриття коду тестами (в ідеалі на 100%, але не завжди це можливо, тому чим ближче відсоток покриття до 100%, тим краще) якраз забезпечує низьку вірогідність того, що в програмі залишились дефекти та баги.

2. Повне або вичерпне тестування не є можливим: Як було зазначено в першому пункті, розробити такі тести, які б покривали всі-всі можливі комбінації користувацьких дій, станів системи та даних на вхід далеко не завжди є можливим. Якщо тестувальник хоче вивести рівень покриття коду тестами на якомога вищій рівень, він має звернути увагу на ризики, які стосуються безпосередньо програми, що тестується і на зіставлення їх пріоритетів (ймовірності виникнення, впливовості на систему та користувача).

3. Завчасне тестування: Перевірка розроблювальної програми на предмет помилок, низької якості та невідповідності вимогам повинна виконуватись з перших же етапів життєвого циклу розробки ПЗ, тобто якомога раніше, інакше з плином розробки пропорційно буде зростати кількість і важкість виявлення помилок, на це буде витрачатись більше часу, що може зірвати вже визначені терміни проекту і коштувати компанії грошей. Також, чим пізніше розпочнеться процес тестування застосунка, тим вище вірогідність, що проект доведеться розпочинати з самого початку, бо кількість необхідних виправлень і безпосередньо часу на коригування помилок буде більше, ніж якщо почати з нуля.

4. Кластеризація або накопичення дефектів: Коли мова йде про скупчення помилок в блоках програми, зазвичай на практиці здійснюється закон Парето:

80% відсотків помилок індексуються у 20% модулів програми, тестування якої здійснюється. Тож, якщо більшість помилок міститься в конкретному модулі, концентрувати свою увагу і зусилля потрібно саме на цьому модулі в першу чергу, а вже потім обмежувати свій час та старання на тестування інших модулів розроблювальної системи.

5. Парадокс пестициду: Фахівці з тестування та забезпечення якості програмних продуктів нерідко стикаються з тим, що розроблені раніше тести з плином життєвих циклів розробки ПЗ втрачають свою силу та актуальність. Тобто тести, які раніше добре покривали код програми, у наступний раз вже не будуть забезпечувати таке гарне покриття коду тестами, ці тести пропускатимуть нові способи та комбінації взаємодії із програмою і ніяк не зможуть допомогти відстежити нові помилки та баги. І якщо з цим нічого не робити, тести виявлятимуть ще менше помилок у подальшому. Тож деякі тести після відправлення програми на доопрацювання часто вже не несуть користі, і старі тестові набори підлягають видаленню або коригуванню та оцінці з точки зору нової версії програми з метою внесення нових і більш доречних тест-кейсів.

6. Тестування є залежним від контексту: Тестувальник перш ніж приступити до тестування, має розуміти ступінь ризику на випадок коли якусь помилку буде проігноровано або не помічено і наскільки руйнівними будуть наслідки. Наприклад, хибні обчислення на медичних приладах призведуть до ризиків для здоров’я людини через неможливість зробити повірку або калібрування приладу. Недостатньо ретельна перевірка системи, що призначена для великої кількості користувачів може неочікувано дати збій при реальній експлуатації тощо. Тож при тестуванні застосунків потрібно звертати увагу на ризики та масштаб проекту і ставитись до процесу розробки тестів з відповідною ретельністю та суворістю (можливо навіть із залученням додаткових експертів у проект).

7. Омана відсутності помилок: Незнаходження помилок у розроблювальній програмі при виконанні тестів не є підтвердженням або доказом відсутності

дефектів у програми. Для впевненості у тому, що програмний продукт готовий до релізу, важливо, наприклад, виконувати не тільки позитивні тестові сценарії, а й негативні. Це є ще одним важливим принципом тестування. Також у пошуку помилок може стати у нагоді принцип позначення очікуваного результату виконання тесту. Виконання тестів різного рівня та типу допомагає також підвищити шанс на знаходження більшої кількості проблем у програмному забезпеченні.

8. Зміни при виконанні тестування недопустимі: Немає сенсу коригувати програму саме у ході тестування, бо тестові сценарії розроблюються для конкретної версії застосунка, яку передбачається або затвердити і продовжити процес розробки далі (у випадку відсутності помилок або наявності незначної кількості дефектів які швидко можна усунути), або повернення даної версії програми на доопрацювання розробникам із звітом, який позначає модулі та ситуації, в яких було виявлено проблеми із зазначенням отриманого та очікуваного результату.

1.2 Поняття модульного тестування програмного забезпечення і можливості які воно надає

Модульне тестування за своїм визначенням є процесом перевірки окремих компонентів або модулів програмного забезпечення з ціллю впевнитись, що вони працюють правильно і відповідають своїм специфікаціям [3]. Основний принцип модульного тестування полягає в тому, що програма розбивається на маленькі частини, які називаються модулями (або юнітами), і для кожного модуля створюються тести, які перевіряють його роботу в ізоляції від інших модулів, щоб переконатися, що його функціонал працює коректно незалежно від інших частин програми.

Виявлення помилок та дефектів в окремих компонентах програми ще до того, як вони вплинуть на інші частини програми, або на систему в цілому,

дозволяє розробникам виправити проблеми на ранніх етапах розробки, коли їх виправлення не являє собою складної та великої за обсягом задачі.

За модуль або блок вважають найменшу тестовану одиницю коду, а саме — функцію або метод, процедуру, модуль, об’єкт класу або сам клас. З цих блоків складається сама програма, яку передбачається реалізувати та випустити у експлуатацію. Якщо навіть без взаємодії з іншими блоками, тобто самі по собі модулі, що тестуються, не працюють, то продовження розробки програми і перехід на її новий життєвий цикл не є доречним та можливим. Бо чим далі буде просуватись розробка функціоналу, тим знову ж таки буде важче потім виправляти помилки та навіть просто з їх пошуком можуть виникнути проблеми, тому раціональніше на кожному етапі програмування застосунків шукати усі можливі дефекти, невиконання очікуваних дій, проблеми, виклики виключень, тайм-аутів тощо.

До того ж, модульні тести допомагають підтримувати високу якість коду шляхом перевірки правильності та працездатності окремих функцій і методів, і безпосередньо найменших частин системи, розробка якої виконується. Це сприяє покращенню читабельності, масштабованості та ефективності коду, оскільки вимагається ретельне розуміння та перевірка кожного окремого модуля перед включенням його до загальної системи.

За мету підтримки надійності та якості програмного забезпечення береться зменшення витрат на виправлення помилок у пізніших етапах розробки, а саме часу і коштів, а також підтримка продукту на протязі усього його життєвого циклу. Модульне тестування впливає на розробку ПЗ таким чином, що вона стає передбачуваною (у плані ризиків та виникнення інших неочікуваних факторів, що безпосередньо впливають на неї) та контрольованою. До того ж на рівні модульного тестування відбувається мінімізація регресії програмного коду.

Модульне тестування можуть робити як розробники, так і тестувальники, але розробники не завжди зацікавлені у такому форматі робіт і можуть мати на меті зосередитись лише на програмуванні, а тестування перекласти на

інженерів з контролю якості. Якщо тестувальник залучений у проект, то раціональніше долучити його до розробки юніт-тестів, і поки він виконуватиме свій план тестування, розробник може зосередитись на створенні та програмуванні інших компонентів майбутньої програми.

Зі сторони розробників бажаним є написання коду в такому вигляді, щоб він у подальшому легко підтримувався і був читабельним, поділеним на логічні одиниці (а не так що один метод містить у собі 2-3 різні за логікою функціонали, які можна було б зробити на декілька методів), щоб кожен блок програми мав чітко визначену задачу, виконання або невиконання якої можна перевірити тестами, окремо від інших модулів або методів.

Юніт-тестування зазвичай виконується в першу чергу. Воно вважається першим рівнем тестування, яке необхідно виконати перш ніж приступати до будь-яких інших, таких як інтеграційне тестування, що перевіряє взаємодію модулів або блоків та їх роботу в тандемі, системного тестування, що зіставляє роботу системи із заявленими функціональними та нефункціональними вимогами і приймальне тестування, що виконується безпосередньо перед тим, як клієнт побачить готову програму або завершену частину від неї. Ієрархію рівнів тестування можна побачити на рисунку 1 [4].

Це не єдиний можливий варіант ієрархії рівнів тестування. Їх може бути більше, або менше. На кожному проекті план тестування затверджується індивідуально. Іноді інтеграційне тестування поєднують із системним разом. Але модульне тестування завжди застосовується і передбачається за планом тестування програмного продукту.

Ці рівні не дарма представлені саме у форматі піраміди. Модульне тестування завжди знаходиться на першій сходинці піраміди, бо кількість розроблювальних тестових сценаріїв при ньому більша, ніж передбачається створювати на наступних рівнях, але час, необхідний на розробку кожного тесту навпаки мінімальний, бо тестовані елементи відносно невеликі. Це базовий рубіж, після якого можна із впевненістю переходити до наступних кроків розробки. Юніт-тестування виконується швидко. Пропорційну

залежність кількості тестів, швидкості проходження етапу тестування та вартість часу та ресурсів, необхідних для тестування наведено на рисунку 2 [5].

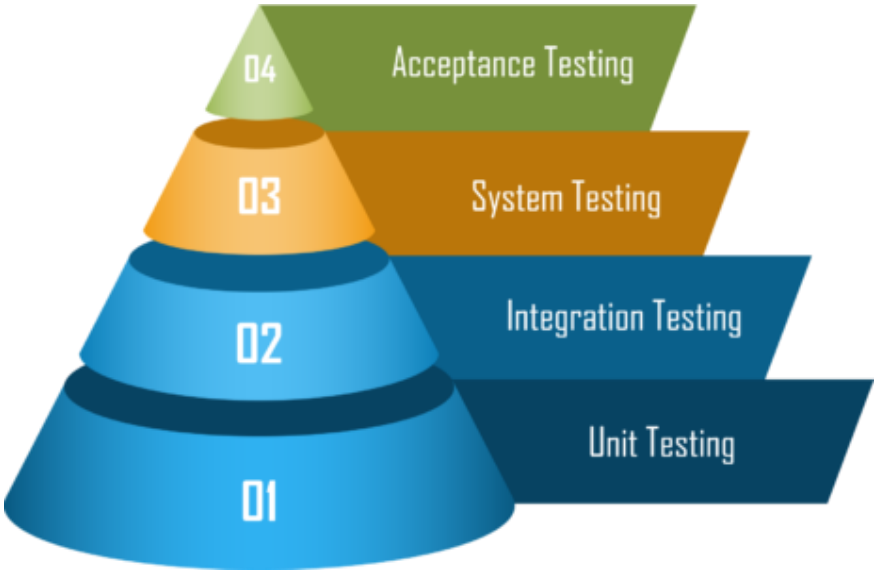

Рисунок *1* — *Ієрархія рівнів тестування*

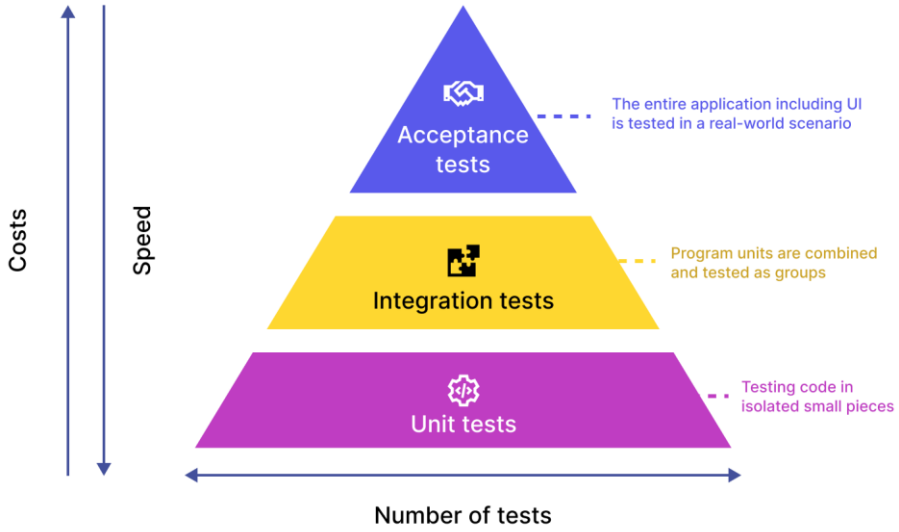

Рисунок *2* — *Кількість тестів, часу та витрат потрібних для кожного з рівнів тестування*

Зазвичай перед тим, як розпочати написання юніт-тестів, розробники створюють тестові фікстури. Це такі методи, що являють собою контекст для виконання тесту або тестів. Тобто в них можна задати початкові стани для

тестового блоку, створити моки об’єктів, щоб запобігти видозмін реальних даних тощо.

Якісний модульний тест перевіряє атомарні (найменші, неподільні) одиниці коду. Такі тести виконуються за лічені мілісекунди. Це зручно і раціонально, бо великий проект на якийсь момент може налічувати тисячі модульних тестів. Також, як вже було зазначено раніше, бажано в юніт-тестах оперувати не реальними об’єктами та даними, а створювати їх моки (підробки), які забезпечують максимальну ізольованість тестового елементу від інших об’єктів. Модульні тести ідеально підходять для автоматизованого тестування, де немає потреби зайвий раз запускати застосунок і перевіряти його вручну.

Модульне тестування надає можливість виконувати такі техніки, як тестування білої скриньки і тестування чорної скриньки. При першому варіанті передбачається знання і видимість внутрішньої структури, логіки та реалізації модуля, що тестується. Така техніка забезпечує повне покриття коду тестами і допомагає досліджувати усі розгалуження та гілки коду програми. Частіше за все використовується на початку розробки, коли модулі маленькі і прості в розумінні.

У той час як тестування чорної скриньки не дозволяє знати, як реалізовано програмний модуль, і залучати до перевірки внутрішню структури системи, також не можна, тож усе, що залишається на перевірку модульними тестами при такому методі — це дослідження зовнішньої поведінки програми. Метод тестування чорної скриньки застосовується вже на більш пізніших етапах проекту, коли модулі інтегровані один з одним і вже сформували складну систему.

Позитивність тест-кейсів — одна з класифікацій тестування. Юніт тести також мають в собі ознаку позитивності, тобто вони можуть бути позитивні або негативні:

- Позитивні тестові сценарії мають на меті перевірити систему на правильність роботи при коректних, очікуваних вхідних даних. Тобто метою є

проведення мануального або ручного тестування в належних і передбачених умовах, як і очікується від користувача.

- Негативне тестування застосовується для перевірки реакції програми на неналежну експлуатацію та на хибні вхідні данні. Від системи, яку тестують, очікується обробка помилок, виклик виключення тощо. Тож негативні тести не мають падати (fail), як і позитивні тести, і якщо негативний тест не пройдено, то програма не готова до введення неправильних типів даних або недопустимих значень, а значить програму треба відправити на доопрацювання.

Модульне тестування в основному використовує твердження (assert) та припущення (assumption). Твердження являють собою умови або вирази, які перевіряють правильність поведінки коду в тестових сценаріях. При запуску модульних тестів вони автоматично перевіряють, чи виконуються ці умови і виводять повідомлення про помилки, якщо якась з них не виконується (наприклад, умова на рівність або нерівність значень та типів).

Припущення у свою чергу використовуються для встановлення початкових умов перед виконанням тестових сценаріїв. Якщо умова припущення не виконується — тест буде пропущений (наприклад, якщо об’єкт для тестування пустий, рівний певному значенню або до нього немає доступу) [6].

Ще одна можливість, яку надає модульне тестування — це можливість створювати параметризовані тести. Такі тести запускаються багато разів підряд, використовуючи при цьому різні набори заданих вхідних даних. Іншими словами, для його створення потрібно написати сам тест і до нього навести різні набори параметрів, тоді тест буде запущено стільки разів, скільки параметрів задано, і при кожному новому виклику буде використано новий параметр зі списку. При цьому в тесті вказується очікуваний результат, і вже з ним кожен раз порівнюються результати виклику тесту і передачі йому тих чи інших даних. Використання таких тестів значно заощаджує час і дозволяє знаходити недоліки в програмі набагато швидше, ніж при мануальному тестуванні [7].

1.3 Сучасні фреймворки модульного тестування Java-застосунків

Одним із чинників стабільної популярності та переваги мови програмування Java над іншими є те, що вона застосовується всеохоплююче. Її застосовують для розробки enterprise-проектів (це великі та складні програмні проекти, які розробляються для великих підприємств чи організацій з використанням технологій та підходів, спрямованих на задоволення бізнес-потреб), підтримки та оновлення мобільних застосунків, що були написані на ній. Також вона може бути використана для написання серверних частин веб-застосунків.

Мова програмування Java має визначне значення у автоматизованому тестуванні, тому що цей розвиток почався з неї, тому природньо те, що для цієї мови розроблено ледь не найбільшу кількість тестових бібліотек і фреймворків.

Стосовно саме модульного тестування, Java нараховує чимало інструментів та засобів. Більшість з них регулярно оновлюються, прогресують та вдосконалюються, тож кожен із них заслуговує на увагу і може вважатись дещо унікальним.

Найвідомішим фреймворком для модульного тестування Java-застосунків є JUnit [8]. Він представляє собою простий та зручний інструмент для написання та виконання модульних тестів. JUnit дозволяє визначати тести, виконувати їх автоматично, проводити перевірку очікуваних результатів, робити припущення, твердження та інше.

Версія JUnit Jupiter (JUnit 5) є новою версією JUnit, яка пропонує багато нововведень та покращень порівняно з JUnit 4, включаючи підтримку лямбда-виразів, динамічних тестів, розширень, створення тестових наборів, використання простих, мета та складених анотацій, застосування припущень тощо.

TestNG [9] є також розповсюдженим засобом модульного тестування. Він є нащадком вищезазначеного JUnit і є його більш потужною версією, бо

пропонує дещо більш розширені можливості, як от більш гнучке та ефективне управління потоком виконання програми, що ним тестується, кращу структурованість тестів і кращі точки валідації. Вдало поєднується у роботі з Selenium (інструментом автоматизованого тестування веб-застосунків).

Mockito — це бібліотека для створення та використання мок-об'єктів у модульних тестах [10]. Вона дозволяє створювати підробки об'єктів (моки) для тестування класів та інтерфейсів, замінюючи реальні об'єкти. Поведінка моків може бути спеціально налаштована, що дозволяє розробнику створювати сценарії та умови, за яких будуть виконуватись тести. Ця бібліотека дозволяє відстежувати, які саме були викликані методи для використовуваних мок-об’єктів, що є дуже доречним при необхідності перевірити залежності методів застосунка. Mockito може бути використаний разом із JUnit та TestNG.

PowerMock — API розширення, яке доповнює Mockito, дозволяючи мокувати статичні методи, конструктори класів, приватні та фінальні класи та методи, на відміну від Mockito, який не надає подібних можливостей [11].

Hamcrest — це бібліотека, в якій використовуються відповідники (matchers) [12]. Вона також дозволяє писати тести, але відповідники це єдине, що вона впроваджує для виконання тестів, тому зазвичай вона використовується у парі з повнофункціональними фреймворками, як, наприклад, JUnit. Відповідник являє собою певний вираз, який тестує деяке співпадіння, наприклад — співпадіння двох текстових рядків. Надаються такі відповідники як equalTo(), assertThat(), not(), anyOf(), is() та інші. При застосуванні анотацій з JUnit, наприклад — @Test, написаний тест буде виконано с допомогою класу org.junit.runner.

AssertJ — це бібліотека асертів (тверджень), яка надає API для написання розбірливих та описових тестових тверджень [13]. Асерти цієї бібліотеки надаються у великому обсязі, що робить AssertJ дійсно потужним засобом тестування. Вона дозволяє перевіряти рівність або нерівність об’єктів, хеш-коду, розмірів колекцій, надає вичерпні і чіткі повідомлення про помилки, коли вони виникають, впроваджує можливість встановлення порядку виконання

методів, також перевіряє такі типи даних як рядки, числа, колекції, масиви, об'єкти тощо, і використовує при цьому безпеку типів в Java, тим самим дозволяючи виявляти помилки ще на етапі компіляції тестів. З цією бібліотекою запроваджується можливість створювати власні твердження.

Spock — це фреймворк-альтернатива відомих JUnit та TestNG, що використовує мову специфікацій для програм Java та Groovy. Його особливостями можна вважати декларативний синтаксис, який перетворює зовнішній вигляд тестів у опис умов та дій, які передбачається розглянути та виконати в межах тесту [14].

1.4 Постановка завдання

Мета даної кваліфікаційної роботи — порівняльний аналіз популярних тестових фреймворків для мови програмування Java — JUnit 5 та TestNG з метою визначення їх ефективності при тестуванні Java-застосунків. Реалізація такого завдання передбачає наступні кроки:

1. Аналіз предметної області: Перед проведенням порівняльного дослідження слід розглянути можливості, техніки, принципи та підходи, що разом складають з себе модульне тестування.
2. Огляд можливостей обох фреймворків: Важливо дослідити відмінності між функціоналом обраних фреймворків, щоб побачити, що між ними є спільного та що їх відрізняє один від одного.
3. Визначення характеристик для порівняння: Визначити показники, за якими будуть порівнюватися фреймворки JUnit 5 та TestNG, і за якими комусь із них буде надаватись перевага у підсумку порівняння.
4. Огляд архітектури, життєвих циклів та засобів організації тестування обома феймворками: зіставлення складу архітектури, вигляду життєвих циклів тестів та аналіз можливостей для організації тестів у вигляді наборів тестів, груп тестів, оцінка підтримки параметризованих тестів, роботи зі звітами.

5. Вибір застосунків: Обрання і огляд застосунків, на яких буде продемонстровано виконання тестів і збір результатів.
6. Написання тестових сценаріїв: Розробка тестових сценаріїв для обраних застосунків з метою порівняти можливості модульного тестування, які надають JUnit 5 та TestNG.
7. Виконання тестів та збір результатів: Написання спроектованих тестів та їх запуск. Збір результатів, оцінка і висновки щодо переваг і недоліків кожного з фреймворків.

Результатом роботи є зібрані сильні та слабкі сторони обох фреймворків, розроблені тести для демонстрації відмінностей між ними та напрацьовані рекомендації щодо використання фреймворків.

1.5 Висновки до розділу 1

1. Виконано огляд поняття тестування та його принципів.
2. Дослідження поняття модульного тестування, його особливостей, можливостей та переваг, які воно надає.
3. Оглянуто інші фреймворки і бібліотеки модульного тестування Java-застосунків.
4. Визначено завдання та етапи дослідження.

2 ПРОЕКТУВАННЯ ТЕСТОВИХ СЦЕНАРІЇВ

2.1 Огляд функціональності фреймворків JUnit 5 та TestNG

2.1.1 Архітектура і життєві цикли тестів у JUnit 5

Фреймворк JUnit 5 можна розбити на 3 модулі, що разом і складають його архітектуру, а саме:

- JUnit Platform: це основа для запуску тестів у JUnit 5. Вона забезпечує інфраструктуру для виконання тестів та підтримує різноманітні середовища виконання, такі як JVM, Android та навіть JavaScript (за допомогою проекту JUnit Pioneer [15]). Також виконує визначення API TestEngine, що необхідно для середовища тестування, що працює на JUnit Platform. Також надає засіб для запуску платформи з командного рядка і Suite Engine — механізм, що дозволяє запускати тестові набори. Підтримується такими середовищами розробки як Eclipse, Visual Studio Code, IntelliJ IDEA та NetBeans, а також такими інструментами управління та збірки проектів як Gradle, Maven та Ant.
- JUnit Jupiter: це підпроект JUnit 5, який надає підтримку для написання та виконання тестів на основі нового програмного інтерфейсу (API) і також дозволяє створювати розширення (наприклад, можна використовувати розширення для створення власних ассертів або налаштовувати середовище перед кожним тестом). Він надає TestEngine, що включає в себе усі основні анотації тестування, мета-анотації, параметризацію і дозволяє виконувати тести.
- JUnit Vintage: це модуль, який забезпечує підтримку для виконання тестів, написаних на ранніх версіях JUnit (а саме — JUnit 3 і JUnit 4) на платформі JUnit 5. Для цього використовується TestEngine. Тож цей модуль забезпечує зворотню сумісність у JUnit 5 і дозволяє поступово впроваджувати у проект новий функціонал, який надається фреймворком. Також цей модуль не викликає конфліктів з іншими частинами платформи JUnit 5.

Більш детально архітектуру цього фреймворка можна побачити на рисунку 3 [16]:

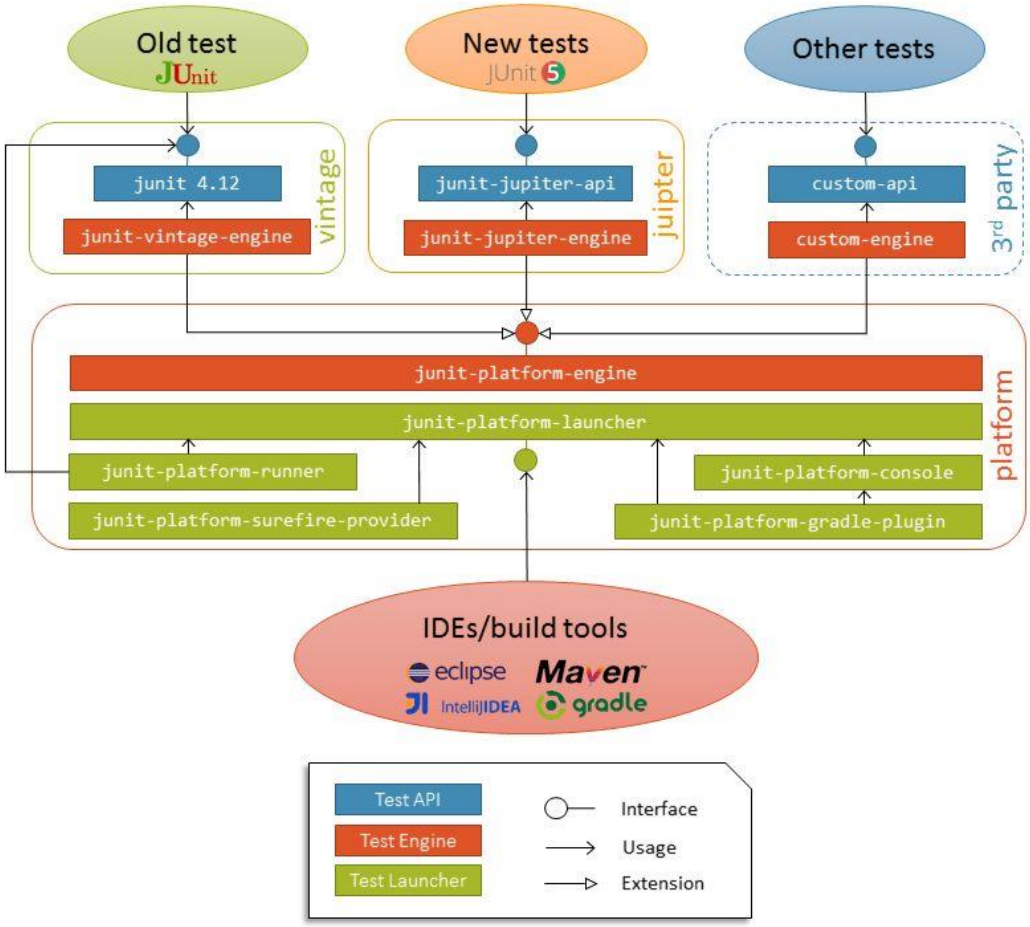

Рисунок *3 — Детальна архітектура JUnit 5*

Життєвий цикл тестів у JUnit 5 організовується за допомогою анотацій і поділяється він на 3 етапи:

1. Налаштування (або Setup): у цій фазі йде підготовка інфраструктури тестів за допомогою анотацій @BeforeAll (на рівні класу) і @BeforeEach (на рівні методів). Цей етап включає в себе установку початкових умов, налаштування середовища та підготовку об'єктів, необхідних для виконання тестів.

2. Виконання тесту (або Test Execution): у цій фазі тест вже безпосередньо виконується використовуючи припущення, твердження.

3. Очищення середовища тестування (або Cleanup): тут йде очищення налаштувань інфраструктури, яка була створена на початку і для цього використовується анотація @AfterAll на рівні класів і анотація @AfterEach на рівні методів.

Фази життєвого циклу тестів у JUnit 5 представлено на рисунку 4 [17]:

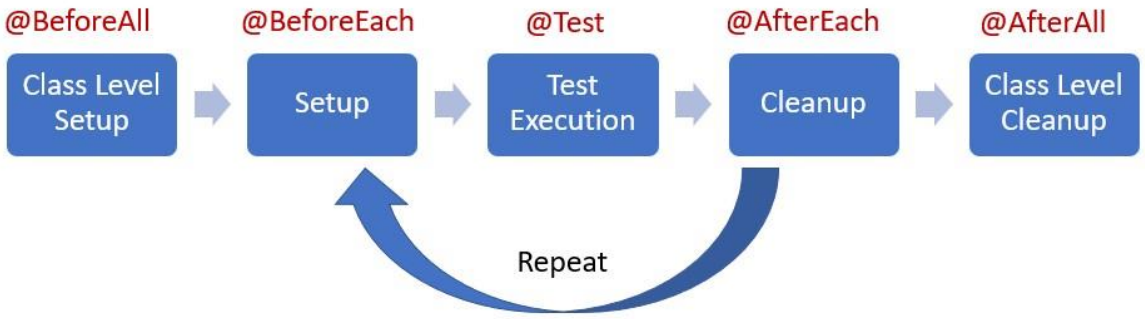

Рисунок *4 — Фази життєвого циклу тестів у JUnit 5*

2.1.2 Архітектура і життєві цикли тестів у TestNG

“Нащадок” JUnit — TestNG складається лише з 1 модуля, що не є перевагою чи недоліком, але використання одного модуля є дещо зручнішим, ніж звертання до декількох різних.

Детальне відображення архітектури TestNG можна побачити на рисунку 5 [18].

На представленій діаграмі архітектури також частково відображено керування життєвим циклом тестів у TestNG: перед кожним тестом (з відповідною анотацією @Test) виконуються умови, описані у @BeforeMethod, потім виконується безпосередньо сценарій тест-кейсу і після кожного тесту виконуються умови, що записані після анотації @AfterMethod. Але це керування життєвим циклом тестів на рівні методів. Для керування життєвим циклом на рівні класів використовуються анотації @BeforeClass (умови буде виконано перед найпершим тест-кейсом у класі) та @AfterClass (умови

виконуються після того як будуть виконані усі тести класу). Повний життєвий цикл тестів у TestNG представлено на рисунку 6 [19].

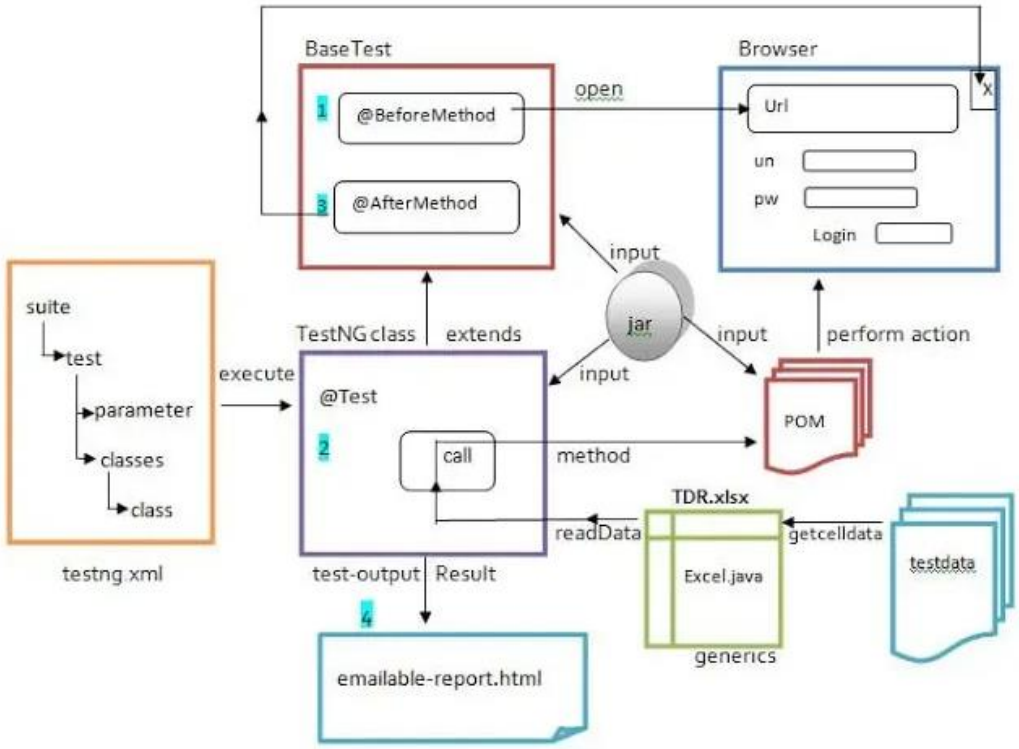

Рисунок *5 — Детальна архітектура TestNG*

```
<BeforeSuite>
    <BeforeTest>
        <BeforeClass>
            <BeforeGroups>
                <BeforeMethod>
                    <Test>
                <AfterMethod>
            </AfterGroups>
        </AfterClass>
    </AfterTest>
</AfterSuite>
```

Рисунок *6 — Повний життєвий цикл тестів у TestNG*

2.1.3 Огляд характеристик та особливостей JUnit 5 та TestNG

Почати порівняння обраних тестових фреймворків доцільно з огляду їх документацій. Вони надають змістовну інформацію стосовно того, що вміє кожен із них, як правильно скористатись можливостями, які вони надають і

також в деякій мірі відповідають на можливі запитання, які можуть виникати при написанні тестів, наприклад, якщо якийсь компонент фреймворку не вдається застосувати.

Так як фреймворк JUnit з'явився раніше за TestNG, доцільніше почати огляд характеристик з нього. Основу його тестів складають твердження та припущення. Вони обидва є static-методами класів org.junit.jupiter.Assertions і org.junit.jupiter.Assumptions відповідно.

Методи-твердження використовуються для порівняння очікуваного результату з фактичним, який надходить в дійсності. Твердження робляться з метою перевірити рівність або нерівність операндів, виклик або навпаки відсутність виклику виключень (для одного об'єкта або для елементів колекції), перевірка значення стану об'єкта (якщо стан зберігається в змінній логічного типу — boolean), перевірка виконання певної умови до спливу часу, вказаного у тайм-ауті тощо.

Припущення у JUnit використовуються на випадок, коли виконання тесту має залежати від певної умови, і якщо ця умова не виконується — виконання тесту має бути перервано. В основному використовується перевірка припущення на хибність чи істинність (методи assumeFalse() і assumeTrue() відповідно), а також метод assumingThat(), який на відміну від інших методів-припущень не перериває тест, якщо умова не виконується. Даний метод просто нічого не робитиме, а якщо умова виконується і при цьому виникає виключення — виникне звичайне падіння тесту.

Для організації тестування у JUnit 5 також можуть використовуватись конфігураційні файли. Там можна конфігурувати життєві цикли екземплярів тестів.

JUnit 5 підтримує впровадження слухачів подій через Listeners API, створення параметризованих тестів, тестових наборів, тестів на винятки, тестів на перевищення тайм-ауту і забезпечує можливість виконувати тести паралельно. До того ж, цей фреймворк надає можливість групувати тести за допомогою тегів, після чого за цими тегами їх можна знайти і виконати окремо

від тестів з іншими тегами і тестів без тегів. Порядок виконання тестів можна встановити вручну — за допомогою анотацій @TestMethodOrder() і @Order.

Також є можливість створювати усередині тестового класу ще один клас — внутрішній, за допомогою анотації @Nested. Методи внутрішніх тестових класів також можуть бути помічені як @Nested, що призведе до передачі передумов зовнішніх тестів на всі рівні дерева вкладень внутрішніх тестів нижче класу, у якому ці передумови було визначено. Це дає можливість виконувати тести незалежно один від одного, наприклад, запускаючи лише внутрішні тести, не запускаючи зовнішні, оскільки код налаштування із зовнішніх тестів завжди виконується.

Ще однією можливістю, яку надає JUnit 5 є створення повторювальних тестів. Такі тести стають у нагоді тоді, коли система працює нестабільно і виконання однієї і тієї ж дії декілька разів підряд призводить до різних результатів. Іноді достатньо один раз відстежити зміну отримуваного результату (із заданих, наприклад, 10 спроб виконати якусь дію), щоб перервати тест і зробити необхідні висновки та дії для корегування несправності в системі.

У свою чергу підставою для розробки фреймворку TestNG були незручності і недоліки, які були на той час у фреймворка JUnit, а саме: відсутність можливості виконувати тести паралельно (тобто у різних потоках одночасно), неможливість робити тести залежними один від одного (що не є недоліком саме для JUnit, бо цей фреймворк не підтримує залежні тести принципово, тобто це закладено в його філософію і це правило не порушується і по сьогодні), відсутність можливості створення параметризації об’єктів тощо. Але обидва ці фреймворки розвиваються, і оновлені версії кожного з них досі виходять регулярно, тож на даний момент, JUnit 5 не має тих недоліків, через які була потреба створювати його більш потужний аналог, але навіть попри це, TestNG все ще має деякі переваги і особливості порівняно з ним, а саме:

- Більша кількість анотацій, спрямованих на керування життєвим циклом тестів, а також більше різноманіття анотацій, що групують тести. TestNG надає

анотації для позначення методу, який має виконатись один раз перед або після запуска поточного набору тестів, а саме @BeforeSuite і @AfterSuite відповідно.

- Підтримка груп тестів. Можливість створювати кілька тестів в одній групі за допомогою атрибута "group" в анотації @Test, після чого застосовуються анотації @BeforeGroups і @AfterGroups, якими будуть позначатись методи, які треба виконати лише один раз до або після усіх тестів, що належать до вказаної групи. До того ж групи можуть включати у себе інші групи. Групи можуть бути на рівні класів і на рівні методів, що робить групування тестів достатньо гнучким, враховуючи те що один проект може потребувати наявність тисячі тестів і більше.
- Наявність атрибутів. Це не є перевагою, а скоріше особливістю, бо в JUnit 5 атрибути не використовуються, тоді як в TestNG атрибутами можна задавати пріоритети тестів (у якому порядку вони мають виконуватись), можна позначати доступність тесту (а саме встановити її в хибне значення) тощо.
- Слухачі підтримуються анотацією @Listeners, тоді як у JUnit 5 слухачі підтримуються за допомогою Listeners API.
- Наявність вбудованих і детальних HTML-звітів і логів виконання тестів, тоді як у JUnit 5 є необхідність користуватись сторонніми плагінами для отримання звітів.

В іншому, функціонал цих фреймворків не сильно відрізняється, обидва вміють робити тести, в які можна передавати набір параметрів, обидва підтримують паралельне виконання тестів, розробку тестів на перебільшення тайм-аутів тощо.

Щодо відмінностей в анотаціях, то найзручніше їх представити у вигляді таблиці. Подивитись відмінності можна в таблиці 1:

Таблиця *1 — Порівняння анотацій у JUnit 5 та TestNG*

JUNIT 5	TESTNG
@BeforeAll	@BeforeClass
@AfterAll	@AfterClass
@BeforeEach	@BeforeMethod
@AfterEach	@AfterMethod
@Disabled, @DisabledOnOs, @EnabledOnOs, @DisabledIfSystemProperty, @EnabledIfSystemProperty, @DisabledIfEnvironmentVariable, @EnabledIfEnvironmentVariable, @DisabledIf, @EnabledIf	@Ignore, @Test (enabled = false)
@TestMethodOrder(Alphanumeric.class), @Order	@Test (priority = 1)
@TestFactory (для написання динамічних тестів)	@Factory (динамічне створення екземплярів тестових класів)
@ParameterizedTest	@Parameters, @DataProvider(name = "credentials")
@DisplayName("X")	@Test(description = "X")
@RepeatedTest	@Test(invocationCount = 10)
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)	@Test(timeOut = 10000)
@Tag("X") (де X — назва тегу)	–
@Nested	–
@Execution(SAME_THREAD), @Execution(CONCURRENT)	–
@TestInstance(Lifecycle.PER_CLASS)	–

JUNIT 5	TESTNG
@TestTemplate (для багаторазового виконання тестів з різними наборами даних)	–
@ExtendWith, @RegisterExtension	–
@TempDir (для збереження тестів і їх очищення у тимчасових сховищах)	–
@TestClassOrder (для @Nested класів)	–
@DisplayNameGeneration (для автоматичного корегування імен, зазначених у @DisplayName)	–
–	@Test(groups = { "X", "Y"}) (де X та Y — назви груп)
–	@Test(threadPoolSize = 2, invocationCount = 4)
–	@DataProvider(parallel = true)
–	@Inject
–	@BeforeSuite
–	@AfterSuite
–	@Listeners
–	@BeforeGroups
–	@AfterGroups
–	@BeforeTest
–	@AfterTest

2.2 Огляд застосунків, за допомогою яких буде виконуватись порівняння можливостей обраних фреймворків

2.2.1 Функціональність та структура застосунка OnlineTest

Для порівняння можливостей JUnit 5 та TestNG було обрано простий застосунок з графічним інтерфейсом користувача, який забезпечує перевірку знань аспектів мови програмування Java шляхом опитування. База питань складається з 10 запитань, кожне з яких має лише 1 правильну відповідь серед запропонованих чотирьох варіантів відповідей. Після того, як користувач натискає кнопку Result (яка виникає на тлі застосунка після переходу на останнє, 10-те запитання), з'являється вікно з результатом, яке інформує, скільки правильних відповідей з десяти надав користувач. Для навігації по запитанням наявні кнопки Next та BookmarkN.

До того ж, застосунок надає можливість створювати закладинки (Bookmark) з питаннями, на випадок, коли користувач поки не знає правильної відповіді та хоче пропустити запитання, залишивши його на потім. Таких закладинок можна створити до 10 екземплярів включно, відповідно до кількості запитань у самому тесті. Після створення закладинки, до неї можна повернутись лише 1 раз, після чого вона стає недоступною для відкриття.

Інтерфейс застосунка можна побачити на рисунку 7:

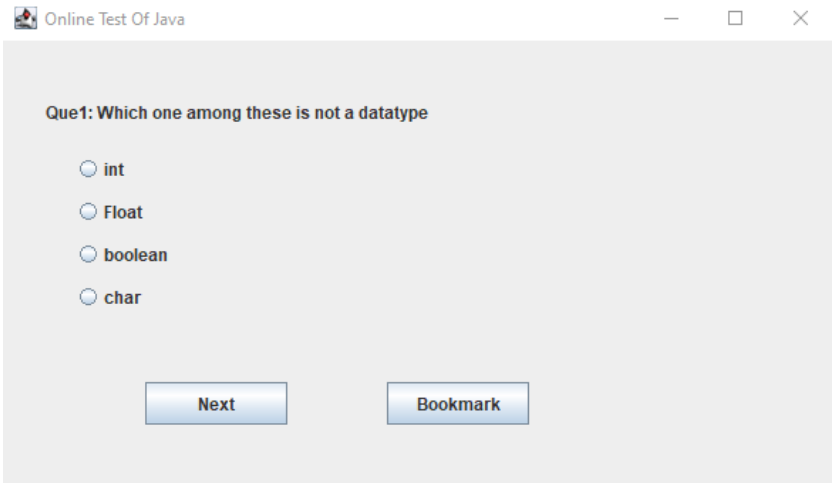

Рисунок *7 — Користувацький інтерфейс застосунка OnlineTest*

Створені закладинки видно на рисунку 8:

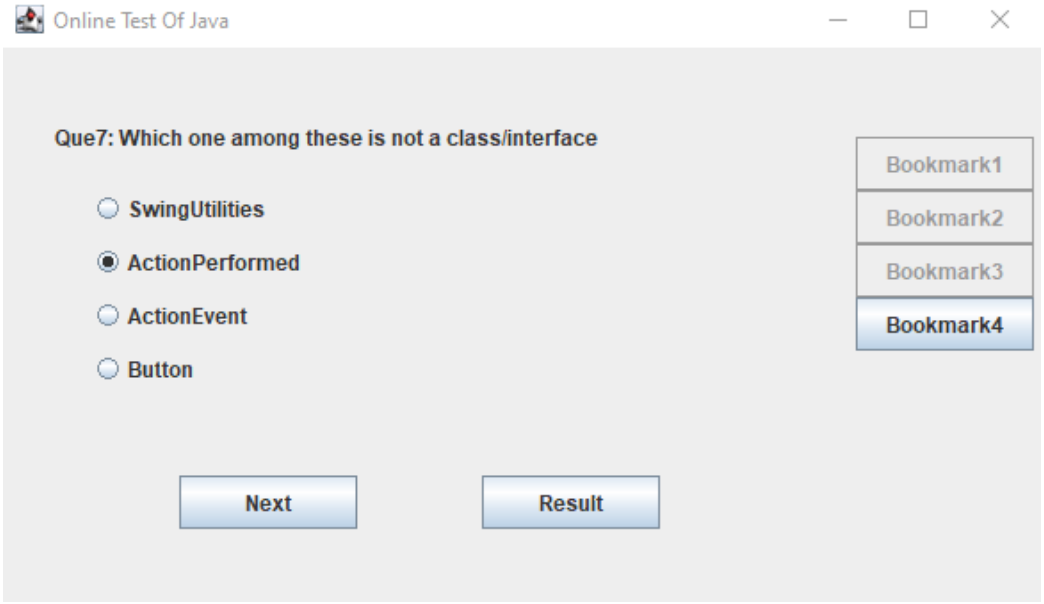

Рисунок *8* — *Створені та задіяні закладинки*

Поява модального вікна з результатом проходження тесту відображена на рисунку 9:

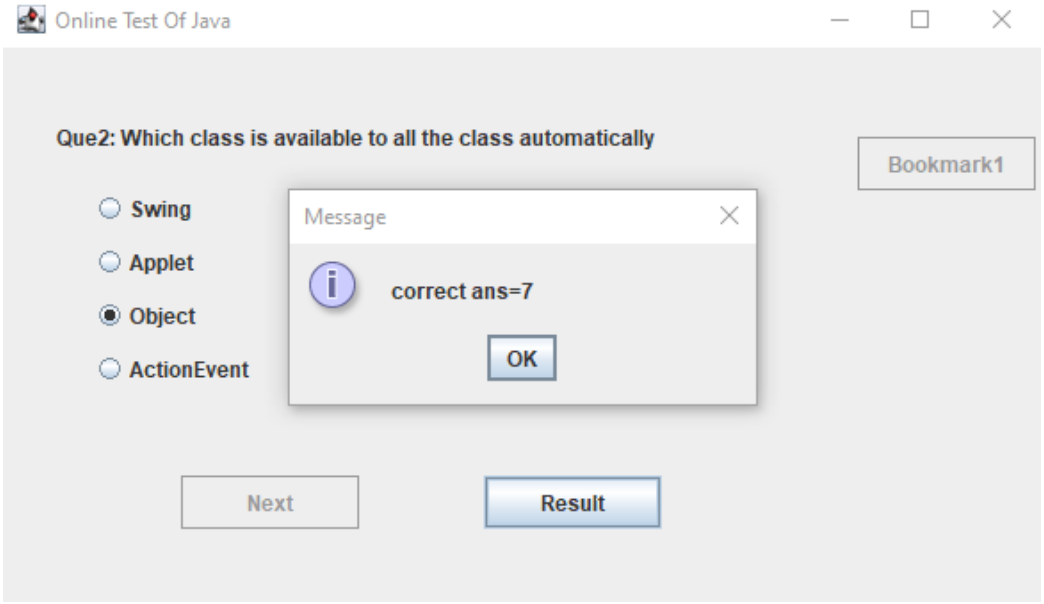

Рисунок *9* — *Результат проходження тесту*

Структура даного застосунка доволі проста. Вона складається з:

1. Одного пакету ua.edu.znu.onlinetest, що містить клас OnlineTest, який наслідує клас JFrame, який у свою чергу реалізує графічний інтерфейс за допомогою бібліотеки Swing;

2. Атрибутів головного класу, таких як мітки, радіо-кнопки та кнопки, що використовуються для відображення питань та вибору відповідей, а також деякі додаткові поля для ведення обліку поточного стану тестування;

3. Клас OnlineTest містить методи для налаштування інтерфейсу, обробки подій та виконання основної логіки програми. Зокрема, метод set() встановлює питання та варіанти відповідей, метод check() перевіряє відповідність обраної відповіді правильній відповіді, а метод actionPerformed() обробляє події натискання кнопок. Наприклад, при натисканні кнопки Next йде перевірка правильності обраної відповіді й нарахування балів у разі, якщо обрана опція з представлених варіантів збігається з правильною;

4. Головний клас містить стандартний метод main(), який створює новий екземпляр класу OnlineTest.

Структура ж проекту також доволі проста. Її можна побачити на рисунку 10.

В проекті наявна папка src, де зберігається вихідний код застосунка, який складається лише з одного файлу OnlineTest.java. У папку target компілюється вихідний код, і в даному випадку туди зберігається клас OnlineTest.class після виконання компіляції. У папці test будуть зберігатись розроблені тести для цього застосунка і вигляді тестових класів та наборів (Test Suit). Файл OnlineTest.iml містить інформацію про структуру та конфігурацію проекту, яку використовує IntelliJ IDEA для правильного відображення та керування проектом, а файл pom.xml містить інформацію про проект і деталі конфігурації, використані Maven для створення проекту OnlineTest.

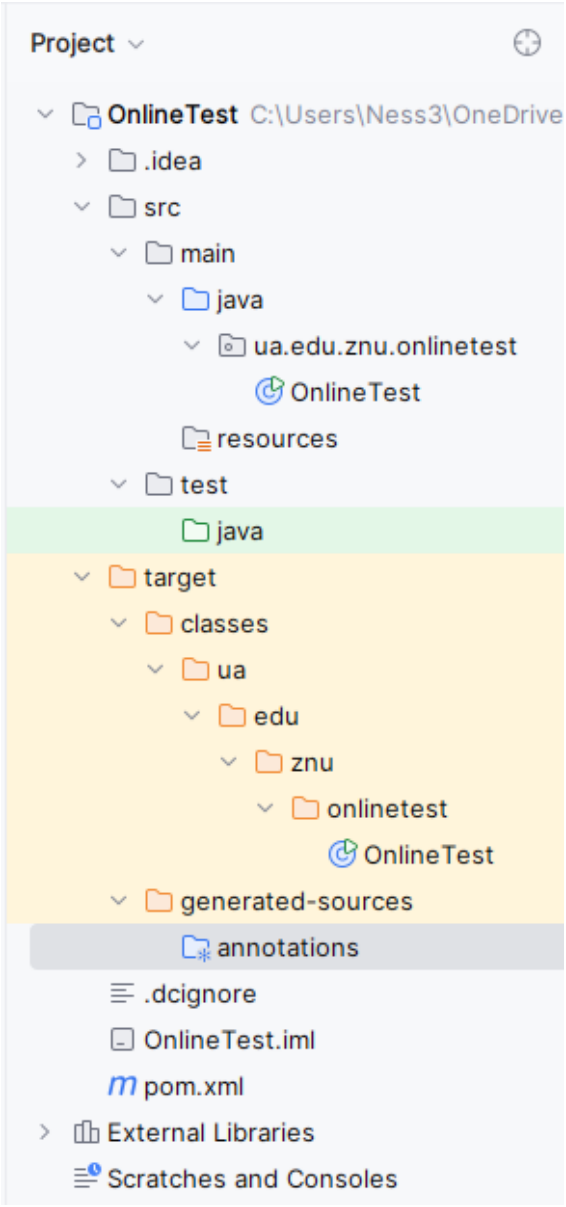

Рисунок *10 — Структура проекту OnlineTest*

2.2.2 Сценарії використання застосунка OnlineTest

Можливості взаємодії користувача із застосунком OnlineTest можна побачити на діаграмі прецедентів (або діаграмі використання), представленій на рисунку 11.

Щодо сценаріїв використання, то вони випливають із діаграми прецедентів. На рисунку 12 представлено діаграму активності для користувача, який безпосередньо почав проходити тест і стоїть перед вибором: відповісти на питання чи пропустити його.

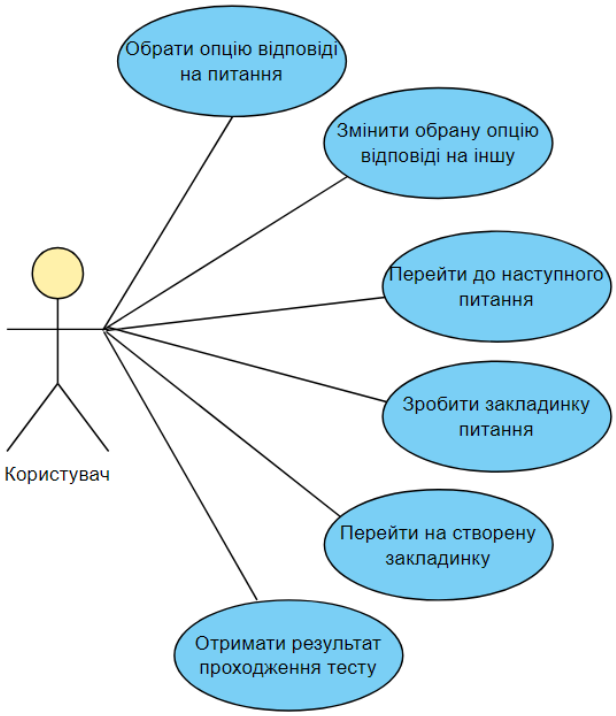

Рисунок *11 — UseCase діаграма застосунка OnlineTest*

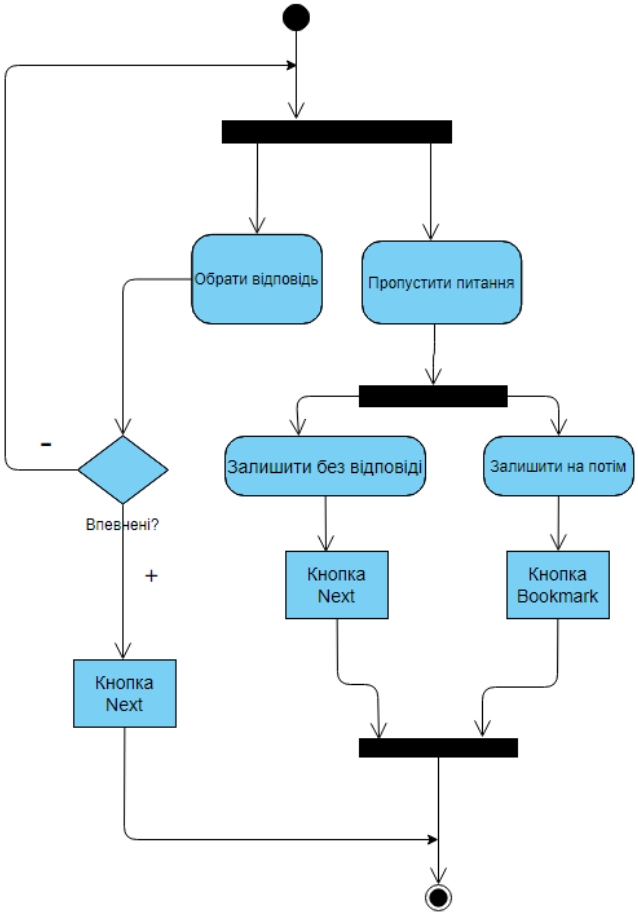

Рисунок *12 — Діаграма активності при проходженні тесту*

Випадок, коли користувач хоче скористатись закладинкою при проходженні тесту відображено на рисунку 13:

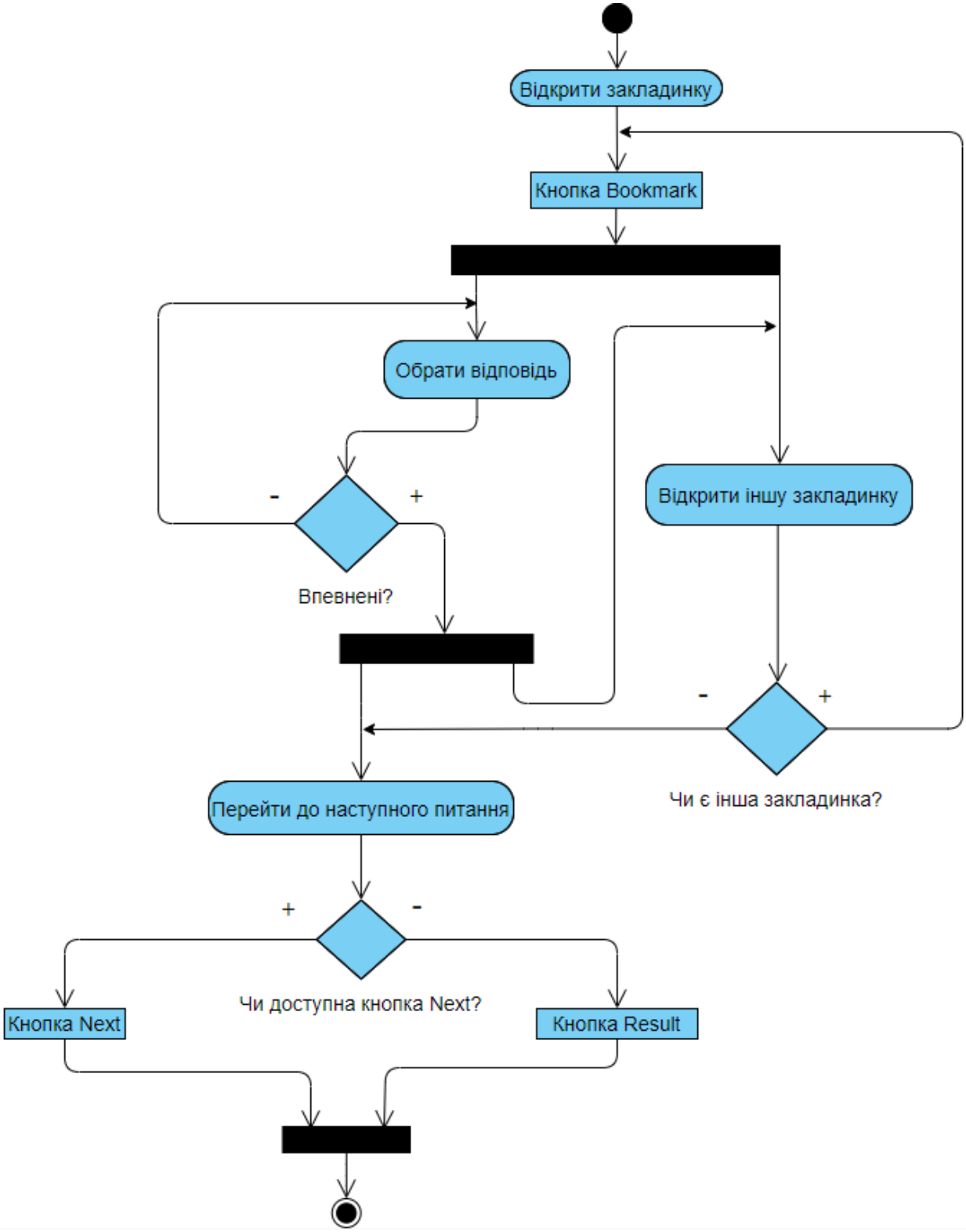

Рисунок *13 — Діаграма активності при відкритті закладинок*

Тож при експлуатації застосунка користувач може відповідати на питання, залишати їх на потім з допомогою закладинок, пропускати питання, перемикатись з питань на закладинки і також переходити з однієї закладинки на іншу і отримувати результат проходження тесту після досягнення 10-го питання. Також важливою приміткою з приводу функціоналу даного застосунка є те, що при відкритті однієї із закладинок і перемикання з неї на іншу закладинку або питання повернутись знову до раніше відкритої закладинки вже буде не можливо.

2.2.3 Функціональність та структура застосунка PortScanner

Для порівняння решти можливостей обраних фреймворків було прийнято рішення провести порівняння також на другому застосунку, який має інший функціонал і дозволяє продемонструвати ще більше відмінностей між JUnit 5 та TestNG.

Java-застосунок PortScanner виконує сканування портів на заданому хості з метою пошуку відкритих портів. Користувачу залишається зазначити ім’я хоста і діапазон портів для пошуку, в межах від 0 до 65535 включно. Цей застосунок також має графічний інтерфейс користувача і містить у собі поля для введення даних, кнопки для очищення даних та для запуску сканування і текстове поле для отримання номерів відкритих портів.

Користувацький інтерфейс цього застосунка можна побачити на рисунку 14, приклад введених даних та результату сканування наведено на рисунку 15, а результат очищення полів кнопкою Reset надано на рисунку 16.

Структура цього застосунка складається з наступних компонентів:

1. Один пакет ua.edu.znu.portscanner, що містить клас PortScanner, який наслідує клас графічного інтерфейсу користувача JFrame. Атрибути головного класу представлені у вигляді полів для вводу, текстового поля, мітки і кнопок.
2. Конструктор, що створює інтерфейс застосунка і методи, що додають на полотно графічного інтерфейсу визначені в якості атрибутів класу PortScanner елементи і задають їм властивості.

3. Внутрішній клас PortScanListener, що реалізує інтерфейс ActionListener (слухача подій) і відповідає за обробку подій при натисканні кнопки “Scan”. Він має атрибути для введених даних у вигляді полів для вводу і атрибут логічного типу для зупинки потоку сканування.

4. Методи у PortScanListener, що перевіряють, чи всі поля заповнені, і якщо так, запускають процес сканування портів у новому потоці на заданому діапазоні портів.

5. Внутрішній клас PortResetListener, що реалізує інтерфейс ActionListener і відповідає за обробку подій при натисканні кнопки “Reset”.

6. Методу класу PortResetListener, який скидає введені дані та зупиняє процес сканування.

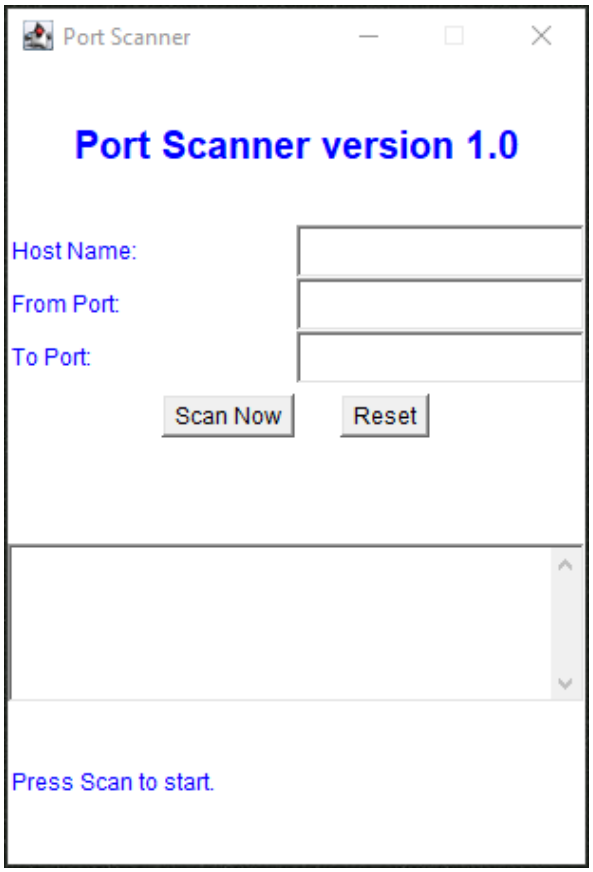

Рисунок *14 — Користувацький інтерфейс застосунка PortScanner*

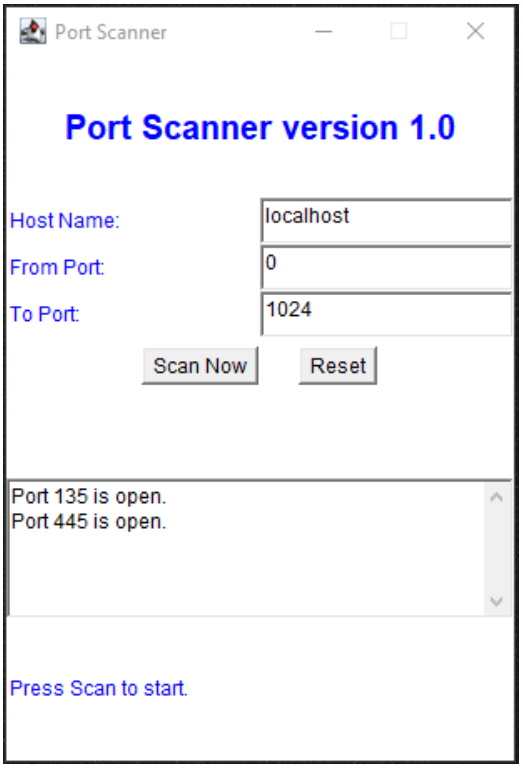

Рисунок *15* — *Результат сканування портів заданого діапазону*

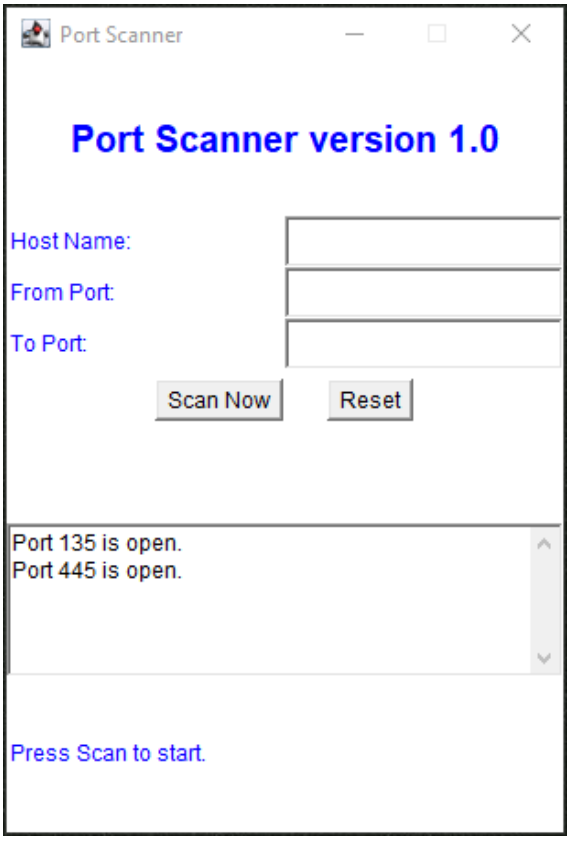

Рисунок *16* — *Результат очищення полів кнопкою Reset*

Структура проекту PortScanner налічує пакет ua.edu.znu.portscanner, головний клас Main, який створює екземпляр класу PortScanner і тим самим запускає застосунок, клас PortScanner, користувацький клас ThreadClosedException, який наслідує клас Exception і створює виключення на випадок закритого потоку виконання сканування, а також package-info.java – спеціальний файл, який зберігає специфічну інформацію про пакет (документування, анотації та коментарі). Структуру проекту можна побачити на рисунку 17:

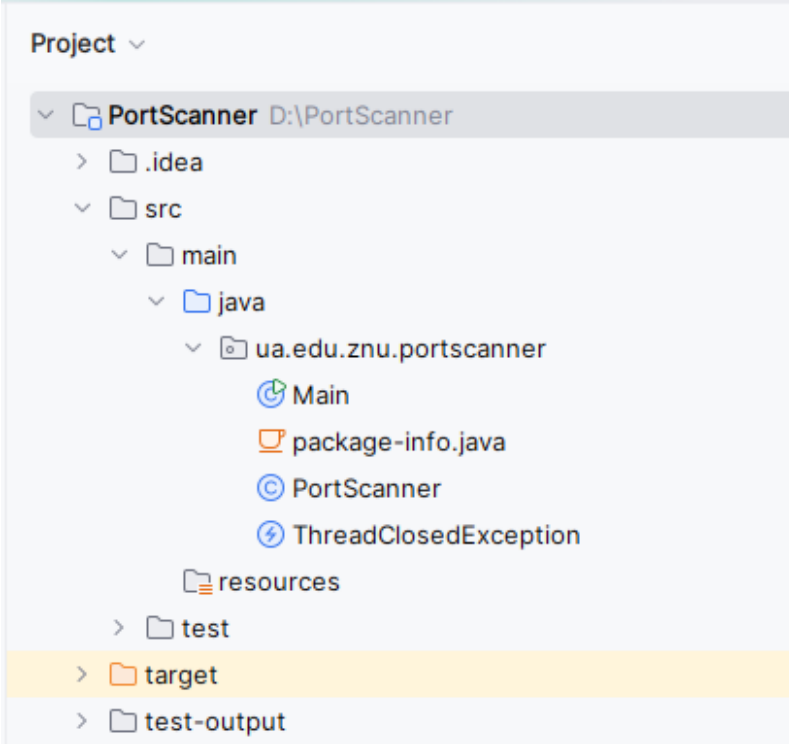

Рисунок *17 — Структура проекту PortScanner*

2.2.4 Сценарії використання застосунка PortScanner

Діаграму прецедентів для цього Java-застосунка можна переглянути на рисунку 18, а сценарій використання цього застосунка згідно діаграми прецедентів було виконано і представлено на рисунку 19.

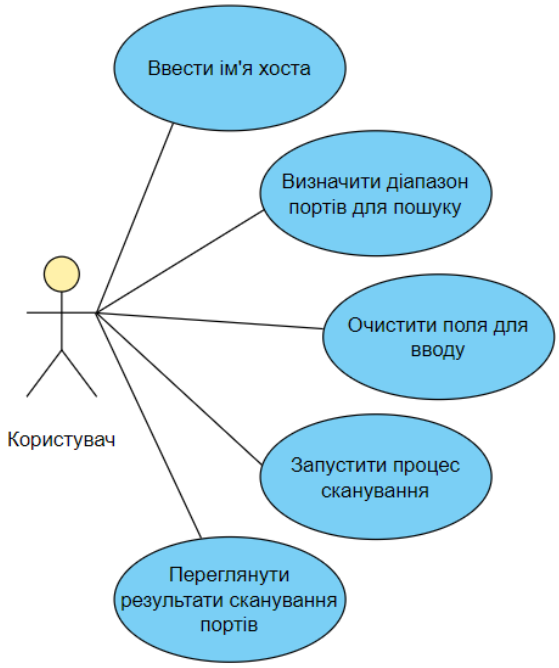

Рисунок *18* — *UseCase діаграма застосунка PortScanner*

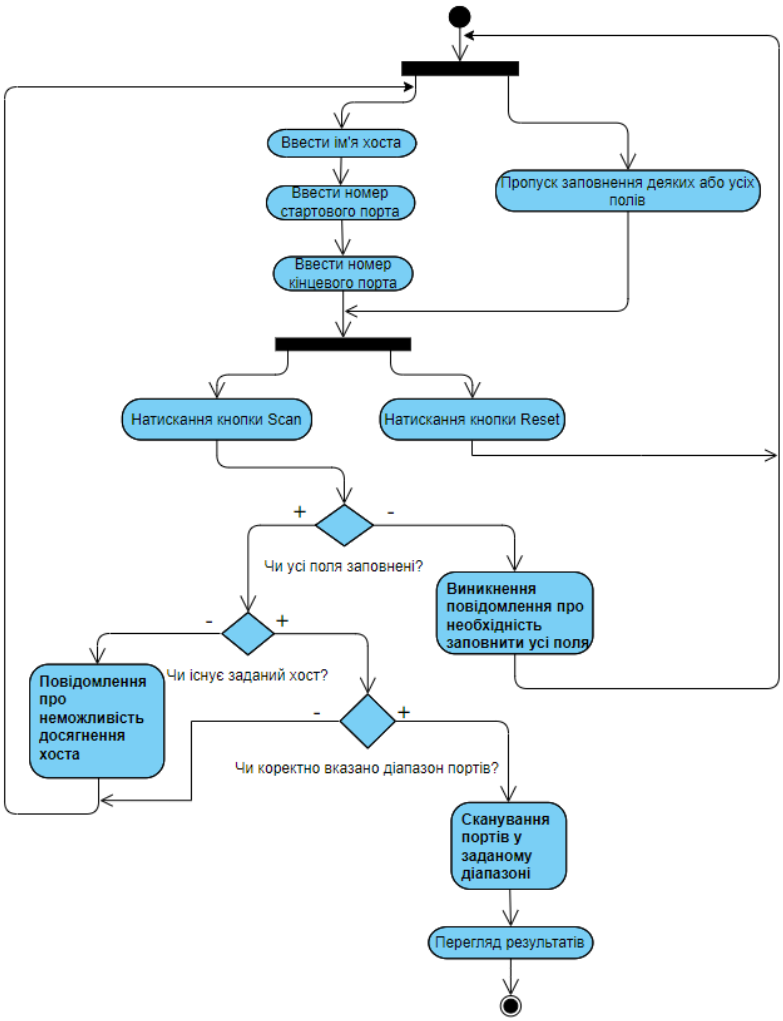

Рисунок *19* — *Сценарій використання застосунка PortScanner*

У такий спосіб користувач може взаємодіяти із цим застосунком.

2.3 Проектування модульних тестів для обраних застосунків

Найбільш показовим порівняння двох фреймворків модульного тестування буде у випадку, коли їх функціонал буде перевірено на практиці. Для цього має сенс спроектувати тест-план [20] для обраних застосунків і безпосередньо написати тестові сценарії для них, враховуючи такі можливості модульного тестування як:

- Позитивні тести.
- Негативні тести.
- Тести на виключення.
- Тести на перебільшення тайм-ауту.
- Параметризовані тести.
- Залежні тести.
- Тести за умовою.
- Генерація звітів.
- Групування тестів.
- Паралельне виконання тестів.

Проектування модульних тестів для застосунка OnlineTest було виконано в таблиці 2, а таблицю зі спроектованими тестами для застосунка PortScanner можна переглянути в таблиці 3.

Тож наступним кроком буде розробка спроектованих тестів на обох фреймворках – JUnit 5 та TestNG. Далі матиме сенс порівняти їх роботу, зручність використання, можливості для групування тестів, змістовність звітів тощо.

Таблиця *2 — Тест-план для застосунка OnlineTest*

№ п/п	Назва тестового класу та тестового методу	Сигнатура методу, що тестується	Перелік вхідних даних (з вказівкою типу)	Очікуваний результат (з вказівкою типу)	Джерело очікуваного результату
1. Чи може користувач натиснути кнопку Next, якщо не вибрано жодної відповіді	**Method:** testCannotProceedWithoutAnswering()	void actionPerformed(ActionEvent e)	onlineTest.jb[1].setSelected(true) (boolean)	initialQuestion == onlineTest.current (boolean)	onlineTest.current
2. Чи збільшується кількість правильних відповідей, якщо відповідь правильна	**Method:** testCorrectAnswerIncreasesCount()	void actionPerformed(ActionEvent e)	onlineTest.jb[1].setSelected(true) (boolean)	initialCount + 1 == onlineTest.count (boolean)	onlineTest.count
3. Чи доступна конпка Result наприкінці тесту	**Method:** testNextButtonAndResultButton()	void doClick()	onlineTest.jb[0].setSelected(true)	Result" == onlineTest.b2.getText() (boolean)	onlineTest.b2.getText()

Таблиця *3 — Тест-план для застосунка PortScanner*

№ п/п	Назва тестового класу та тестового методу	Сигнатура методу, що тестується	Перелік вхідних даних (з вказівкою типу)	Очікуваний результат (з вказівкою типу)	Джерело очікуваного результату
1. Чи викидається виключення якщо хост недійсний	**Method:** unknownHostException	void performPortScan()	hostGetter = "a3" (String), fromPortGetter = 0 (int), toPortGetter = 1024 (int)	UnknownHostException.class (Exception), ThreadClosedException.class (Exception)	portScanListener.performPortScan
2. Робота метода не перевищує 100 мс	**Method:** performanceTest	void performPortScan()	hostGetter = "localhost" (String), fromPortGetter = 1 (int), toPortGetter = 1024 (int)	No Exception	onlineTest.count
3. Параметризований тест	**Method:** testPerformPortScan	void performPortScan()	{hostName (String), fromPort (int), toPort (int), expectedResult (boolean)}	–	–
4. actionPerformed очищує значення полів для введення	**Method:** actionPerformedShouldResetFieldsToEmptyValues	void actionPerformed (ActionEvent e)	hostName = "localhost" (String), fromPort = 0 (String), toPort = 1024 (String)	hostName = "" (String), fromPort = "" (String), toPort = "" (String)	portScanner.hostName, portScanner.fromPort, portScanner.toPort
5. Текстові поля зчитуються правильно	**Method:** readsTextFieldsCorrectly	void getText()	hostName = "localhost" (String), fromPort = 0 (String), toPort = 1024 (String)	hostName = "localhost" (String), fromPort = "0" (String), toPort = "1024" (String)	portScanner.hostName, portScanner.fromPort, portScanner.toPort
6. Чи доступна кнопка Scan	**Method:** scanButtonEnabledAfterCall	void setEnabled(boolean b)	hostName = "localhost" (String), fromPort = 0 (String), toPort = 1024 (String)	scan.isEnabled() == true (boolean)	portScanner.scan
7. Чи дійсно знаходяться відкриті порти	**Method:** testPortScannerLogOutput	void performPortScan()	hostName = "localhost" (String), fromPort = 0 (String), toPort = 1024 (String)	logOutput.contains("Port 135 is open.") (String), logOutput.contains("Port 445 is open.") (String)	portScanner.logOutput

2.4 Висновки до розділу 2

1. Було зроблено огляд архітектури і життєвих циклів тестів обраних фреймворків.
2. Виконано порівняння основних можливостей тестових фреймворків JUnit 5 та TestNG і їх анотацій.
3. Зроблено огляд застосунків, на яких передбачається виконувати тестування з метою порівняння можливостей JUnit 5 та TestNG на практиці, а саме – огляд задач, які вони виконують, варіантів та сценаріїв використання.
4. Спроектовано тести для кожного із застосунків, які будуть розроблені на обох фреймворках.

3 РОЗРОБКА ТА ВИКОНАННЯ ТЕСТІВ ЗАСОБАМИР JUNIT 5 ТА TESTNG

3.1 Використання можливостей фреймворків при тестуванні обраних застосунків

3.1.1 Тестування застосунка OnlineTest

Спроектовані у попередньому розділі тести для застосунка OnlineTest спрямовані на перевірку роботи методів у належних умовах (позитивне тестування) та при некоректних діях (негативне тестування). Самі тестові методи були розміщені у двох класах, один з яких призначений для тестів, написаних на JUnit 5, а інший – для тестів на TestNG відповідно.

Перший спроектований і розроблений тест перевіряє можливість натиснення кнопки Next за умови, що жодної відповіді не було обрано. Це є прикладом негативного тестування, бо передбачається, що користувач завжди повинен залишати відповідь, перш ніж переходити до наступного питання, тож потрібно відтворити таку дію у тесті і перевірити застосунок на реакцію на цю дію або її перешкоджанню. Написаний на JUnit 5 тест можна переглянути на лістингу 1.

Лістинг *1 — Негативний тест на JUnit 5*

```
@Tag("Negative_test")
@DisplayName("Чи може користувач натиснути кнопку Next,
якщо не обрано жодної відповіді")
@Test
public void testCannotProceedWithoutAnswering() {
    boolean noAnswerSelected = true;
    for (int i = 0; i < 4; i++) {
        if (onlineTest.jb[i].isSelected()) {
            noAnswerSelected = false;
```

```
                break;
            }
        }
        assertTrue(noAnswerSelected, "No answers should be
selected");
 int initialQuestion = onlineTest.current;
        ActionEvent nextButtonEvent = new
ActionEvent(onlineTest.b1, ActionEvent.ACTION_PERFORMED,
"Next");
        onlineTest.actionPerformed(nextButtonEvent);
        assertEquals(initialQuestion, onlineTest.current);}
```

Як видно із лістингу 1, спочатку нам потрібно впевнитись, що жодна відповідь не була обрана, після чого робиться перше твердження на істинність значення прапорця noAnswerSelected. Якщо твердження не вірне, відповідь усе ж було обрано і тест впаде, надавши зазначене повідомлення у консоль. Якщо ж твердження вірне, тест виконується далі, зберігаючи поточний номер питання й імітуючи натискання кнопки Next, після чого робиться вже друге твердження на рівність очікуваного номера питання (0) і фактичного. Запустивши цей тест, можна побачити нове повідомлення і успішне проходження тесту, це можна побачити на рисунках 20 і 21:

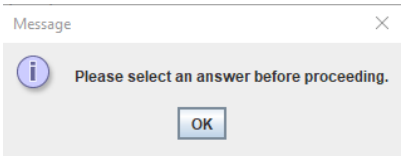

Рисунок *20 — Повідомлення про необхідність спершу обрати відповідь*

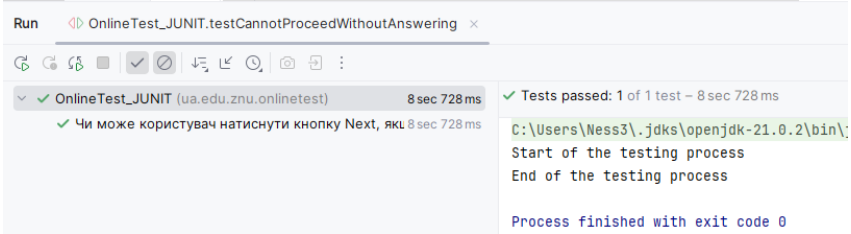

Рисунок *21 — Успішне проходження тесту*

Тіло аналогічного до цього тестового методу, розробленого на TestNG повністю співпадає з лістингом, зазначеним вище, але є різниця у тому, які анотації застосовуються при використанні фреймворка TestNG, це можна побачити в лістингу 2.

Лістинг *2 — Негативний тест на TestNG*

```
@Test(description = "Чи може користувач натиснути кнопку
Next, якщо не вибрано жодної відповіді",
        groups = "Negative_test")
public void testCannotProceedWithoutAnswering() {
    boolean noAnswerSelected = true;
    for (int i = 0; i < 4; i++) {
        if (onlineTest.jb[i].isSelected()) {
            noAnswerSelected = false;
            break;} }
    assertTrue(noAnswerSelected, "No answers should be
selected");

    int initialQuestion = onlineTest.current;
    ActionEvent nextButtonEvent = new
ActionEvent(onlineTest.b1, ActionEvent.ACTION_PERFORMED,
"Next");
    onlineTest.actionPerformed(nextButtonEvent);

    assertEquals(initialQuestion, onlineTest.current);}
```

Як видно з лістингу 2, алгоритм дій в обох тестах дійсно вийшов ідентичним, а різниця вже вбачається у анотаціях перед тестовими методами. У JUnit 5 є можливість задати текст, який буде виведено при спробі виконати тест за допомогою анотації @DisplayName. Також, тест помічено тегом

"Negative_test" анотацією @Tag. У TestNG відсутні аналогічні анотації, але є можливість задати опис тесту за допомогою атрибута description анотації @Test, а також наявний атрибут groups, який додає поточний тест до вказаної групи.

Атрибут description дозволяє додати у автоматично згенеровані фреймворком TestNG звіти вказаний текст, а за допомогою груп є можливість згрупувати тести і задати їм передумови та післяумови виконання за допомогою анотацій @BeforeGroups і @AfterGroups відповідно. Можливості додавати тести у групи і задавати групам передумови та післяумови виконання, генерувати вбудовані звіти і додатково зазначати текст, який має там відображатись для кожного тесту у JUnit 5 відсутні.

Наступний тест, який передбачалось розробити, має перевіряти збільшення лічильника балів у разі правильної відповіді на запитання. Реалізацію такого тесту на JUnit 5 наведено на лістингу 3.

Лістинг *3 — Тест з перевіркою накопичення балів на JUnit 5*

```
@Tag("Positive_test")
@DisplayName("Чи збільшується кількість правильних
відповідей, якщо відповідь правильна")
@Test
public void testCorrectAnswerIncreasesCount() {
  // Локальна змінна для рахунку правильних відповідей
    int initialCount = onlineTest.count;

    // Обрання відповіді
    onlineTest.jb[1].setSelected(true);
    // Імітація натискання кнопки "Next"
    ActionEvent nextButtonEvent = new
ActionEvent(onlineTest.b1, ActionEvent.ACTION_PERFORMED,
"Next");
    onlineTest.actionPerformed(nextButtonEvent);
```

```
    // Перевірка: чи збільшився рахунок правильних
відповідей
    assertEquals(initialCount + 1, onlineTest.count);}
```

Даний тест імітує вибір опції з правильною відповіддю і натиск кнопки Next, після чого, у разі якщо накопичення балів пройшло успішно, тест має пройти. Інакше він впаде, що буде говорити про те, що кількість балів не збільшилась після відповіді. Правильні відповіді зазначені у застосунку заздалегідь, тож підібрати правильну опцію для тесту не складно і це не впливатиме на ймовірність провалу тесту. Реалізація такого тесту на TestNG нічим не відрізняється від наданого у лістингу 3 окрім вже зазначених та оглянутих анотацій перед тестами.

Даний тест є прикладом позитивного тестування, бо в ньому відтворюються дії, які є очікуваними від користувача, а саме обрання опцій з відповідями і перехід на інші питання.

І третій тест, який передбачалось розробити для даного застосунка, був такий, що перевіряє доступність кнопки Result після досягнення останнього питання. Його реалізація на JUnit 5 надана у лістингу 4.

Лістинг *4 — Тест з перевіркою кнопки Result на JUnit 5*

```
@Tag("Positive_test")
@DisplayName("Чи доступна конпка Result наприкінці тесту")
@Test
public void testNextButtonAndResultButton() {
    for (int i = 0; i < 10; i++) {
        onlineTest.jb[0].setSelected(true);
        onlineTest.b1.doClick();}
    assertEquals("Result", onlineTest.b2.getText(), "Result
button should be enabled on the last question");}
```

Реалізація такого тесту на TestNG ідентична наведеній на лістингу 4. Цей тест успішно пройшов, що говорить про те, що кнопка для отримання результату дійсно доступна наприкінці тесту.

Акцентуючи увагу на відмінностях при використанні фреймворків JUnit 5 та TestNG, варто почати з тегів і груп. Відмічені тегом тести у JUnit 5 можна запустити з допомогою конфігурацій запуску, як представлено на рисунку 22 (приклад для середовища розробки IntelliJ Idea):

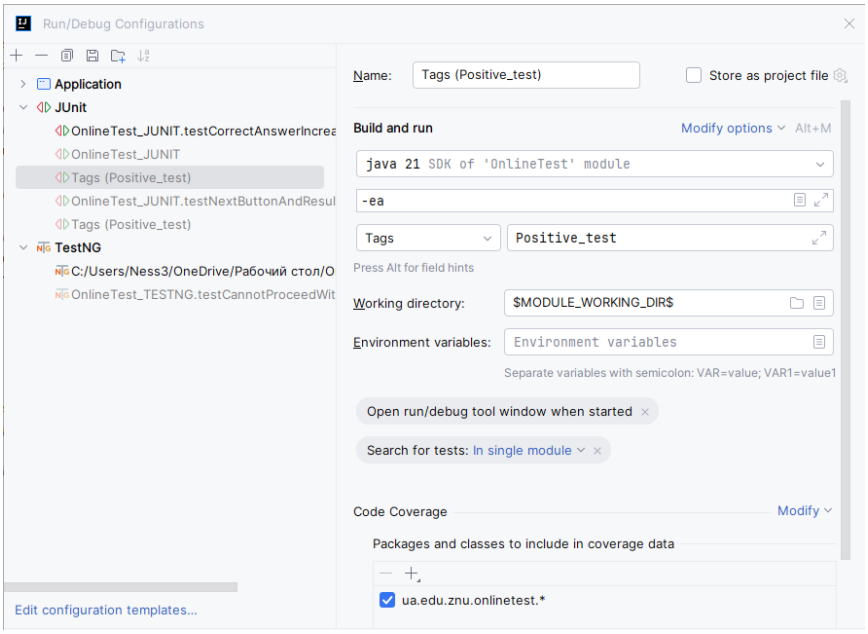

Рисунок *22 — Налаштування запуску тестів з тегом Positive_test*

Результат виконання тестів з вказаним тегом можна побачити на рисунку 23:

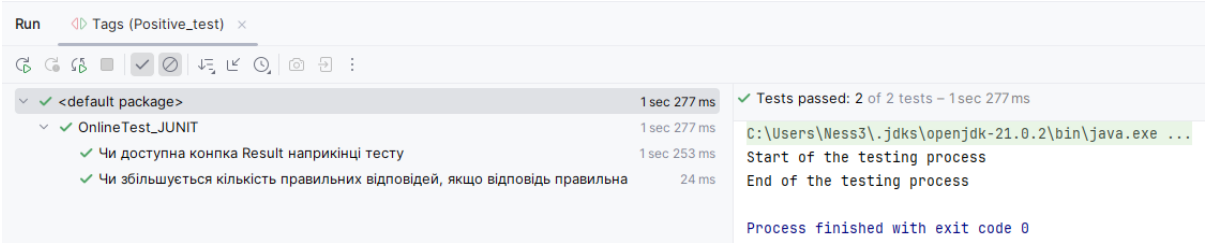

Рисунок *23 — Результат виконання тестів з тегом Positive_test*

Подібним чином згрупувати тести можна і в TestNG, але за допомогою груп, як це було зазначено раніше. Виконати тести, що належать до вказаних груп, можна або за допомогою конфігураційного фалу TestNG.xml, безпосередньо запустивши його, або вказавши у файлі з конфігураціями проекту Maven, а саме pom.xml, перелік груп і зазначений раніше конфігураційний файл TestNG.xml. Використовуваний у застосунку файл TestNG.xml наведено на лістингу 5.

Лістинг 5 — *Конфігураційний файл TestNG.xml*

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Positive test Suite">
    <test name="Positive test">
        <groups>
            <run>
                <include name="Positive_test"/>
            </run>
        </groups>
        <classes>
            <class
name="ua.edu.znu.onlinetest.OnlineTest_TESTNG"/>
        </classes>
    </test>
</suite>
```

Результат запуску групи тестів Positive_test через конфігураційний файл можна побачити на рисунку 24.

Серед вже зазначених переваг фреймворка TestNG є наявність генерації вбудованих звітів. Для того, щоб згенерувати такий звіт, достатньо виконати наступну команду командного рядка: mvn clean test -D"groups=Positive_test". Буде знайдено і виконано тестові методи, які включені у групу Positive_test, і

визначені у класі або класах, що прописані у TestNG.xml. Переглянути створені звіти можна на рисунках 25 та 26.

Рисунок *24 — Результат запуску конфігураційного файла TestNG.xml*

Positive test

Tests passed/Failed/Skipped:	2/0/0
Started on:	Tue Jun 11 02:41:30 EEST 2024
Total time:	1 seconds (1132 ms)
Included groups:	Positive_test
Excluded groups:	

(Hover the method name to see the test class name)

PASSED TESTS				
Test method	**Attribute(s)**	**Exception**	**Time (seconds)**	**Instance**
testNextButtonAndResultButton Test class: ua.edu.znu.onlinetest.OnlineTest_TESTNG Test method: Чи доступна кнопка Result наприкінці тесту		0	ua.edu.znu.onlinetest.OnlineTest_TESTNG@7807ac2c	
testCorrectAnswerIncreasesCount Test class: ua.edu.znu.onlinetest.OnlineTest_TESTNG Test method: Позитивний тест: чи збільшується кількість правильних відповідей, якщо відповідь правильна		0	ua.edu.znu.onlinetest.OnlineTest_TESTNG@7807ac2c	

Рисунок *25 — Результат генерації звіту*

Test	# Passed	# Skipped	# Retried	# Failed	Time (ms)	Included Groups	Excluded Groups
Positive test Suite							
Positive test	2	0	0	0	1 132	Positive_test	

Class	Method	Start	Time (ms)
Positive test Suite			
Positive test — passed			
ua.edu.znu.onlinetest.OnlineTest_TESTNG	testCorrectAnswerIncreasesCount	1718062890835	11
	testNextButtonAndResultButton	1718062890854	718

Positive test

ua.edu.znu.onlinetest.OnlineTest_TESTNG#testCorrectAnswerIncreasesCount

back to summary

ua.edu.znu.onlinetest.OnlineTest_TESTNG#testNextButtonAndResultButton

back to summary

Рисунок *26 — Згенерований emailable-report*

Саме звіт emailable-report стає у нагоді тоді, коли є необхідність надіслати звіт з пройденими, пропущеними або провалениими тестами проектному менеджеру або іншим спеціалістам, залученим у проект. Для створення звітів з результатами тестування на JUnit 5 необхідно підключати стороні плагіни, що є мінусом, на відміну від TestNG, який має вбудовані звіти.

3.1.2 Тестування застосунка PortScanner

Так як функціонал попереднього застосунка не зміг надати більший простір для демонстрації інших відмінностей обраних фреймворків, було прийнято рішення провести тестування ще одного застосунка для демонстрації інших можливостей JUnit 5 та TestNG.

Одним із спроектованих та розроблених для цього застосунка тестів був тест на виклик виключення у випадку, коли користувач вказує невірне ім’я хоста. Реалізація цього метода на JUnit 5 представлена на лістингу 6.

Лістинг *6 — Тест на виключення на JUnit 5*

```
@Nested
class HostException {
    @Tag("Exception_test")
    @DisplayName("Чи викидається виключення якщо хост 
недійсний")
    @Test
    void unknownHostException() {
        PortScanner.PortScanListener portScanListener = 
portScanner.new PortScanListener();
        portScanListener.hostGetter = "a3";
        portScanListener.fromPortGetter = 0;
        portScanListener.toPortGetter = 1024;
        assertThrows(UnknownHostException.class, 
portScanListener::performPortScan); }}
```

Відповідний до цього тест, але реалізований на TestNG наведено у лістингу 7.

Лістинг *7 — Тест на виключення на TestNG*

```
@Test(priority = 2, description = "Чи викидається
виключення якщо хост недійсний",
        expectedExceptions = {UnknownHostException.class,
ThreadClosedException.class},
groups = "Exception_test")
public void unknownHostException() throws Exception{
    PortScanner.PortScanListener portScanListener =
portScanner.new PortScanListener();
    portScanListener.hostGetter = "a3";
    portScanListener.fromPortGetter = 0;
    portScanListener.toPortGetter = 1024;
    portScanListener.performPortScan();}
```

З наведених лістингів можна побачити, що способи перехоплення виключень різні. JUnit 5 використовує для цього твердження із зазначенням виключення у тестовому методі, а TestNG зазначає батьківські класи виключень, які можуть виникнути, у анотації @Test без застосування тверджень.

Також, із лістингу 6 можна помітити, що тестовий метод помічено анотацією @Nested. Ця анотація дозволяє впроваджувати внутрішні класи до основного тестового класу. Це дозволяє додатково групувати тести і відокремлювати їх від інших, з метою розділити тестові методи за призначенням або іншою ознакою і застосувати до них інші методи-фікстури з передумовами та післяумовами виконання на рівні методів і класів. У свою чергу TestNG не підтримує можливість впровадження внутрішніх класів до тестового класу, тож потрібно звертатись до можливості створювання групи

тестів або створювати різні тестові класи і здійснювати їх виклик за допомогою конфігураційного файлу (результат такого рішення наведено на рисунку 28). Результат розбиття основного тестового класу на підкласи наведено на рисунку 27:

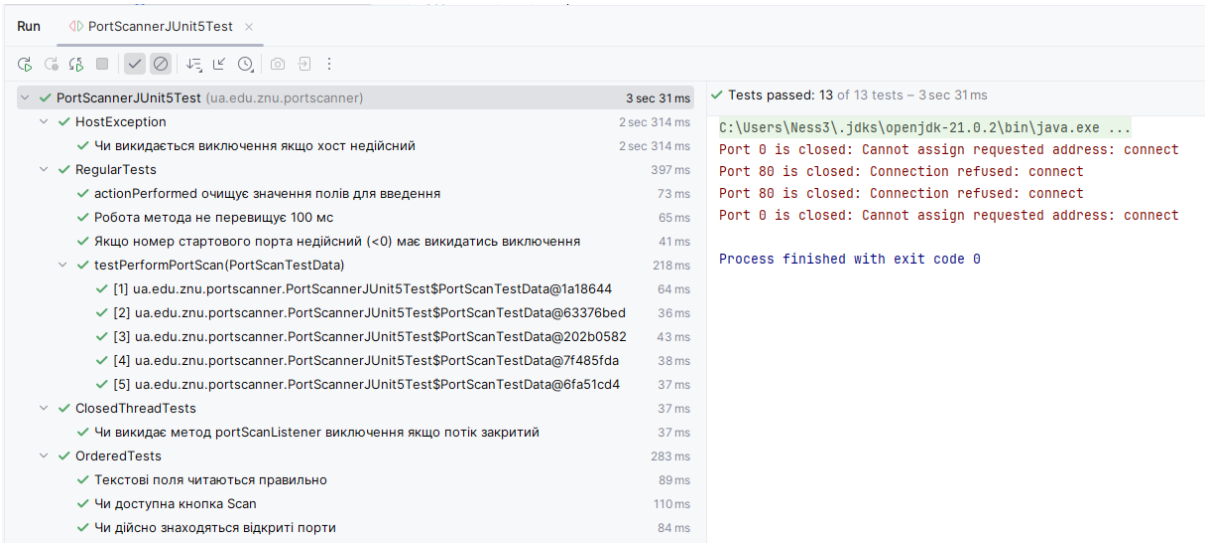

Рисунок *27 — Результат виконання основного тестового класу JUnit 5*

Рисунок *28 — Результат розбиття тестів на класи у TestNG*

Наступний вид тесту, який передбачалось розробити на обох фреймворках був тест на перебільшення тайм-ауту. Такий тест відстежує, чи виконується якийсь метод більше заданого для роботи часу, і якщо так, тест переривається і падає. Це дозволяє контролювати час, у продовж якого виконується функціонал застосунка, що тестується, і у разі необхідності оптимізувати методи, час виконання яких не відповідає вимогам до нього. Реалізація такого методу на JUnit 5 наведено на лістингу 8.

Лістинг 8 — *Тест на тайм-аут на JUnit 5*

```
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
@Tag("Timeout_test")
@Test
@DisplayName("Робота метода не перевищує 100 мс")
void performanceTest() {
    PortScanner.PortScanListener portScanListener =
portScanner.new PortScanListener();
    portScanner.hostName.setText("localhost");
    portScanner.fromPort.setText("1");
    portScanner.toPort.setText("1024");
    assertDoesNotThrow(portScanListener::performPortScan);}
```

Реалізація аналогічного тесту на TestNG представлена на лістингу 9.

Лістинг 9 — *Тест на тайм-аут на TestNG*

```
@Test(priority = 3, description = "Робота метода не
перевищує 1000 мс", timeOut = 1000, groups =
"Timeout_test")
public void performanceTest(){
    PortScanner.PortScanListener portScanListener =
portScanner.new PortScanListener();
    portScanListener.hostGetter = "localhost";
    portScanListener.fromPortGetter = 1;
    portScanListener.toPortGetter = 1024;
    try {
        portScanListener.performPortScan();
    } catch (Exception e) {
        Assert.fail("Expected no exception to be thrown,
but got: " + e);}}
```

Так як у TestNG відсутні твердження для заперечення факту виникнення виключення (як у випадку з JUnit 5 це assertDoesNotThrow), реалізація перехоплення виключення відрізняється від реалізації тесту на JUnit 5, і через це займає більше часу на виконання, тож зазначення тайм-ауту в 100 мс призведе тільки до падіння тесту. Відсутність необхідного виду твердження у TestNG погано впливає на процес тестування, вимушуючись підлаштовуватись під обмеження або звертатись до можливостей JUnit 5, комбінуючи використання двох фреймворків в одному тестовому методі.

Наступний вид тесту, який був спроектований для порівняння можливостей обох фреймворків був параметризований тест. Такий вид тесту дозволяє передавати у тестовий метод різні набори параметрів, тоді як виконати тест необхідно усього один раз. Такий тест за один виклик виконається стільки разів, скільки наборів даних було до нього передано.

Розроблені тестові методи і на JUnit 5, і на TestNG мають ідентичну будову, тож має сенс сконцентруватись на відмінностях при передачі параметрів до тестових методів. Спосіб передачі параметрів до тестового методу на JUnit 5 наведено у лістингу 10.

Лістинг *10 — Спосіб передачі параметрів до тестового методу на JUnit 5*

```
static Stream<PortScannerJUnit5Test.PortScanTestData>
portScanTestData() {
    return Stream.of(
       new
PortScannerJUnit5Test.PortScanTestData("localhost", 80, 80,
false),  // Порт закритий
       new
PortScannerJUnit5Test.PortScanTestData("localhost", 135,
135, false), // Порт відкритий
       new
```

```
PortScannerJUnit5Test.PortScanTestData("localhost", -1,
65536, false), // неправильний номер порту
       new
PortScannerJUnit5Test.PortScanTestData("localhost", 0,
70000, false),
       new PortScannerJUnit5Test.PortScanTestData("", 80,
80, false) // порожній хост
    );}
@ParameterizedTest
@MethodSource("portScanTestData")
public void
testPerformPortScan(PortScannerJUnit5Test.PortScanTestData
testData) { }
```

Реалізація передачі параметрів до тестового методу на TestNG наведена на лістингу 11.

Лістинг *11 — Спосіб передачі параметрів до тестового методу на TestNG*

```
@DataProvider(name = "portScanTestData")
public Object[][] portScanTestData() {
return new Object[][] {
 { new PortScanTestData("localhost", 80, 80, false) },  //
Порт закритий
{ new PortScanTestData("localhost", 135, 135, false) }, //
Порт відкритий
{ new PortScanTestData("localhost", -1, 65536, false) }, //
неправильний номер порту
{ new PortScanTestData("localhost", 0, 70000, false) },
 { new PortScanTestData("", 80, 80, false) } // порожній
хост
    };}
```

```
@Test(priority = 1, description = "ParameterizedTest",
dataProvider = "portScanTestData")
public void testPerformPortScan(PortScanTestData testData)
{}
```

Як видно з представлених лістингів 10 і 11, JUnit 5 в якості джерела даних застосовує анотацію @MethodSource (та аналогічні до нього @ValueSource або @CsvSource), а самі дані передаються у вигляді потоку, масиву тощо, у той час як TestNG використовує анотацію @DataProvider з атрибутом name (що позначає постачальника даних), яка до того ж має атрибут паралельного виконання parallel (за замовчуванням встановлене у false) і повертає двовимірний масив об’єктів.

В лістингах тестових методів написаних на TestNG (а саме лістинги 7, 9 та 11) можна побачити атрибут priority анотації @Test. Цей атрибут визначає порядок виконання тестів. Аналогом засобу пріоритезації у JUnit 5 є анотація @Order. Приклад упорядкування тестів у JUnit 5 наведено на рисунку 29:

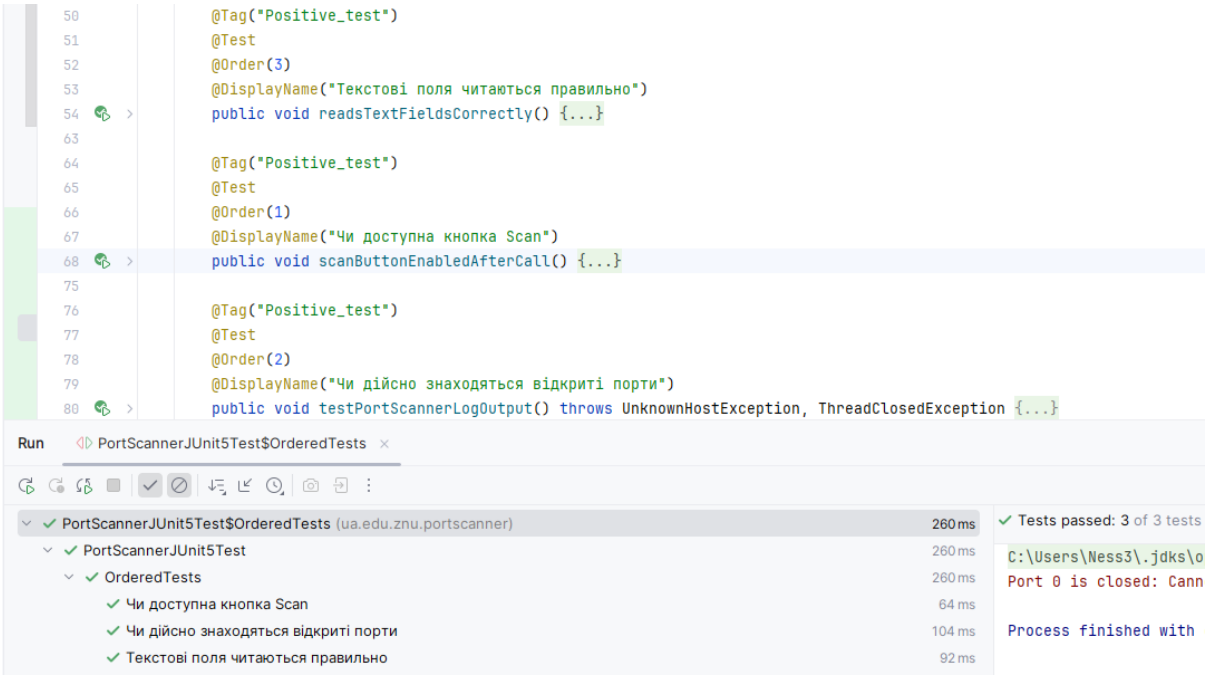

Рисунок *29 — Результат упорядкування тестів у JUnit 5*

Упорядковані тести у TestNG виконуються у заданому порядку після усіх неупорядкованих тестів. Результат роботи атрибута priority анотації @Test у TestNG наведено на рисунку 30:

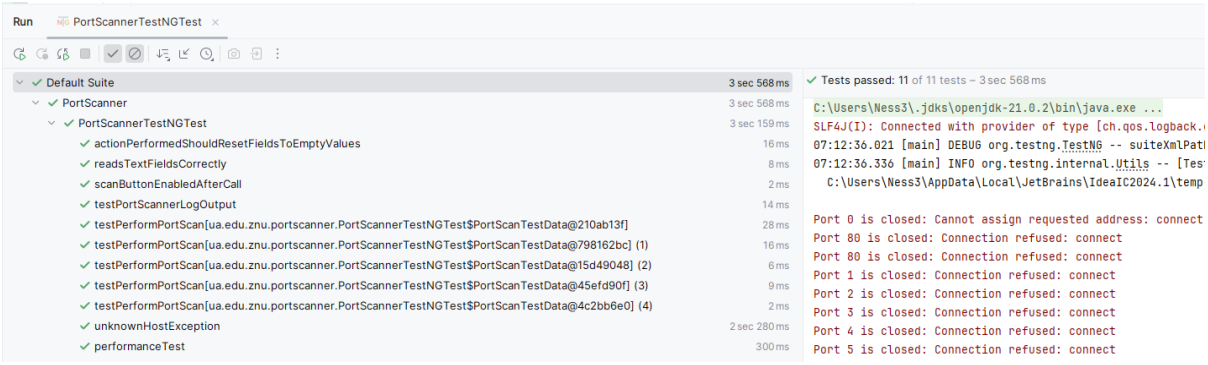

Рисунок *30 — Результат упорядкування тестів у TestNG*

Наступний вид тесту, реалізацію якого треба порівняти в обох фреймворках – це тести з умовою. Тіло тестового методу з умовою містить припущення, невиконання умови якого завадить тесту виконуватись далі. Фреймворк TestNG не підтримує використання припущень, тож таку можливість модульного тестування можна використати лише з фреймворком JUnit 5. Приклад такого тесту наведено на лістингу 12.

Лістинг *12 — Тест з умовою реалізований на JUnit 5*

```
@Test
@DisplayName("actionPerformed очищує значення полів для
введення")
public void actionPerformedShouldResetFieldsToEmptyValues()
{
    Assumptions.assumeTrue(portScanner.reset.isEnabled());
    portScanner.hostName.setText("localhost");
    portScanner.fromPort.setText("0");
    portScanner.toPort.setText("1024");

    portScanner.new
```

```
PortResetListener().actionPerformed(null);
    final PortScanner localPortScanner = portScanner;

    SwingUtilities.invokeLater(() -> {
        assertEquals("",
localPortScanner.hostName.getText());
        assertEquals("",
localPortScanner.fromPort.getText());
        assertEquals("",
localPortScanner.toPort.getText());
    });}
```

Визначена у припущенні вимога проста: кнопка Reset має бути доступна для натискання, інакше немає сенсу виконувати подальші кроки у тілі тестового метода. Наявність можливості застосовувати припущення у JUnit 5 є перевагою над можливостями, які надає TestNG.

Ще однією можливістю модульного тестування є створення залежних тестів. Ця практика не вважається виключно правильною та надійною, але по сьогодні трапляються ситуації, де залежні тести стають у нагоді й повністю виправдовують своє застосування на практиці. У JUnit 5 відсутня можливість робити такі тести, але це не є недоліком саме цього фреймворка, бо відсутність можливості робити тести залежними є для нього принциповою, це закладено в його філософію. А реалізацію залежності методів у TestNG можна переглянути на лістингу 13.

Лістинг *13 — Реалізація залежності методів у TestNG*

```
@Test(description = "Текстові поля читаються правильно",
groups = "Positive_test")
public void readsTextFieldsCorrectly() {}

@Test(description = "Чи доступна кнопка Scan",
```

```
dependsOnMethods = "readsTextFieldsCorrectly", groups =
"depended_test")
public void scanButtonEnabledAfterCall() {}

@Test(description = "Чи дійсно знаходяться відкриті порти",
dependsOnMethods = {"readsTextFieldsCorrectly",
"scanButtonEnabledAfterCall"},
        groups = "depended_test")
public void testPortScannerLogOutput() throws
UnknownHostException, ThreadClosedException {}
```

Якщо один тест залежить від іншого, то падіння першого тесту призведе до падіння усіх наступних тестів, які були залежні від нього, що дуже нагадує принцип “доміно”. Такий підхід у тестуванні не завжди виправданий, тому за будь-якої можливості його краще уникати.

Ще однією популярною можливістю, якою користуються при модульному тестуванні є паралельне виконання тестів. В звичайному режимі тести виконуються один за одним в одному потоці, і чим більша кількість розроблених тестів для проекту, тим помітніше, що час виконання тестових класів та методів стає довшим. Враховуючи, що модульні тести потрібно перезапускати часто (у разі зміни тестових методів або функціоналу застосунка), час, що витрачається на тестування в цілому буде невиправдано великим. Для вирішення цієї проблеми стає у нагоді розбиття виконання тестових методів між різними потоками. Чим більше потоків задіяно, тим швидше виконуватимуться тести, так як з однієї великої черги формуються декілька менших черг. Але кількість потоків для застосування не є необмеженою і залежить від кількості ядер процесору комп'ютера.

Паралельне виконання тестів, написаних на JUnit 5, здійснюється через налаштування у конфігураційному файлі junit-platform.properties. Такий файл

треба створювати вручну, і прописати там необхідні налаштування. Приклад таких налаштувань наведено на лістингу 14.

Лістинг *14 — Налаштування паралельного виконання тестів у JUnit 5*

```
junit.jupiter.execution.parallel.enabled=true
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default =
concurrent
junit.jupiter.execution.parallel.config.strategy=dynamic
junit.jupiter.execution.parallel.config.dynamic.factor=4
```

З приведеного лістинга 14 можна побачити, що паралельне виконання тестів увімкнено, виконання тестових методів виконуватиметься в одному потоці (execution.parallel.mode.default = same_thread), виконання тестових класів відбуватиметься в різних потоках, а також налаштування кількості потоків, які будуть задіяні при виконанні тестів. Ці налаштування можна змінювати під різні потреби. Наприклад, можна і тестові методи, і тестові класи виконувати у різних потоках, або задати максимальну кількість потоків, яку можна задіяти при паралельному виконанні тестів (execution.parallel.config.dynamic.max-pool-size-factor).

Як вже було сказано раніше, в звичайному режимі тестові методи виконуються один за одним, в одному потоці. Приклад такого виконання можна побачити на рисунку 31:

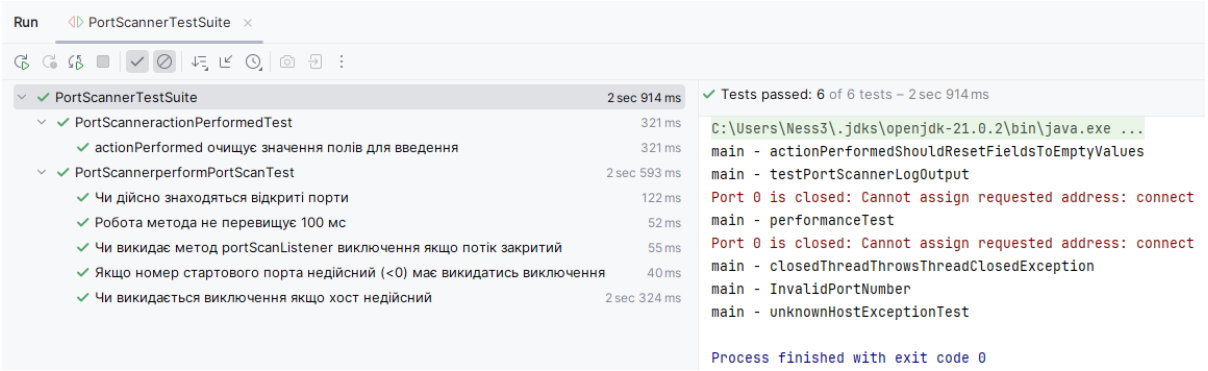

Рисунок *31 — Виконання тестового набору в звичайному режимі*

Як видно з рисунка 31, запущений тестовий набір (який містить у собі 2 тестові класи, а саме PortScanneractionPerformedTest і PortScannerperformPortScanTest) повністю виконався у одному потоці main. Якщо ж скористуватись налаштуваннями конфігураційного файлу junit-platform.properties, наведеними на лістингу 14, то як і було зазначено вище, запуск тестових класів буде здійснено у різних потоках, і кожен тестовий метод буде виконуватись у тому потоці, до якого належить його клас. Це можна побачити на рисунку 32:

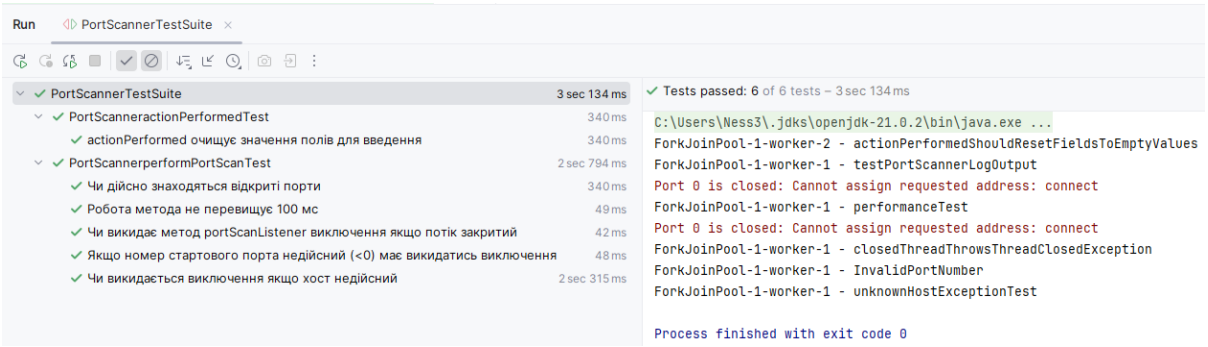

Рисунок *32 — Виконання тестового набору в паралельному режимі*

Використання конфігураційного файлу в JUnit 5 це не єдиний спосіб виконувати тести паралельно. Цей фреймворк також надає анотацію @Execution, якій можна встановити режим виконання SAME_THREAD або ж CONCURRENT, ця анотація вказується перед кожним тестовим методом.

Щодо паралельного виконання тестів у TestNG, там теж є можливість користуватись цим режимом або через конфігураційний файл, або через атрибут анотації @DataProvider, що призначена для параметризованих тестів. Наповнення конфігураційного файлу, який дозволяє виконувати тести паралельно можна побачити на лістингу 15.

Лістинг *15 — Налаштування паралельного виконання тестів у TestNG*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Test-method Suite" parallel="tests" thread-
count="3">
<test name="Test-method test" group-by-instances="true">
<classes>
<class
name="ua.edu.znu.portscanner.PortScannerParallelTestNG" />
</classes>
</test>
</suite>
```

З лістингу 15 можна побачити, що паралельне виконання налаштоване на рівні тестів (є також рівень методів та рівень класів), дозволена кількість потоків для використання дорівнює трьом і паралельне виконання буде здійснюватись для тестового класу PortScannerParallelTestNG. Якщо порівнювати налаштування конфігураційних файлів на обох фреймворках, то принциповою відмінністю є те, що у JUnit 5 увімкнення паралельного виконання тестів торкається усіх тестових класів і методів, написаних на JUnit 5, а у TestNG режим паралельного виконання тестів буде застосовано лише до тих класів, які зазначені у конфігураційному файлі, і до того ж треба запускати сам конфігураційний файл, а не самі класи. Результат запуску конфігураційного файлу, наведеного у лістингу 15 можна переглянути на рисунку 33:

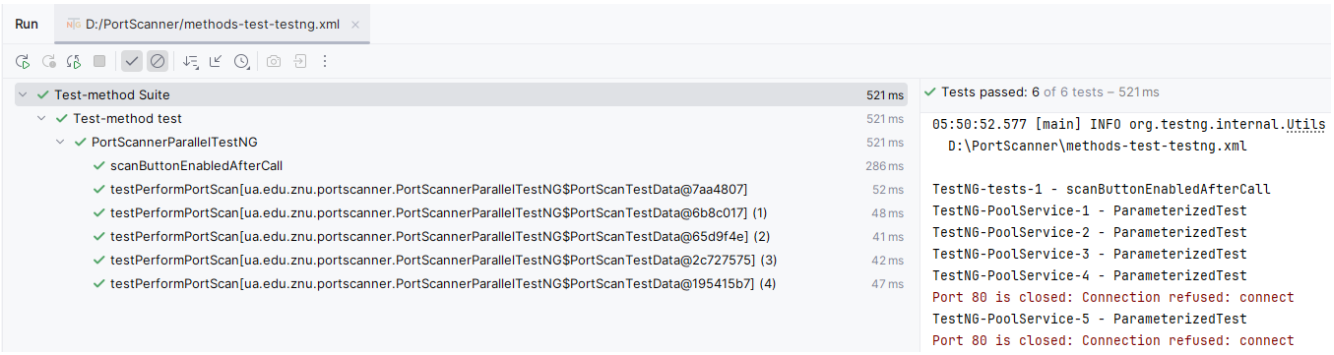

Рисунок *33 — Виконання конфігураційного файлу в TestNG*

Також у TestNG можна налаштовувати режим виконання тестів безпосередньо у анотаціях до тестових методів, а саме використовуючи атрибут singleThreaded = true анотації @Test і атрибут parallel анотації @DataProvider. Приклад виконання методів з використанням таких атрибутів можна переглянути на рисунку 34:

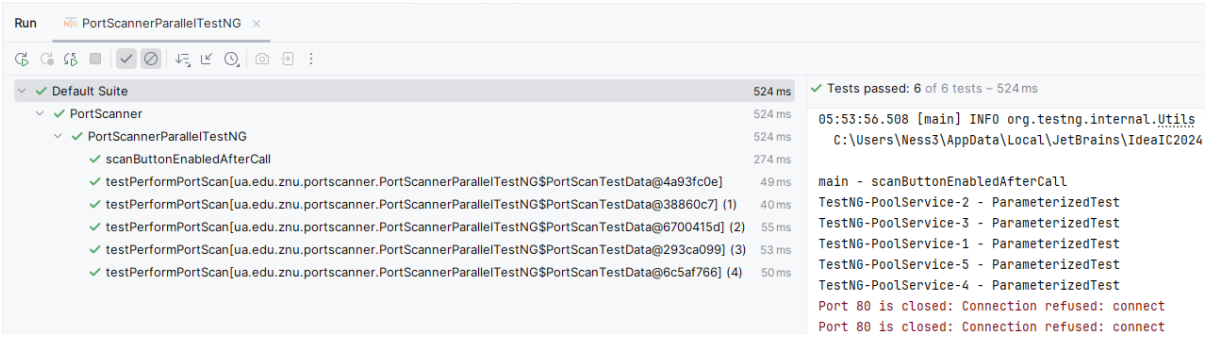

Рисунок *34 — Результат використання атрибутів керування режимом виконання у TestNG*

Фреймворк TestNG також надає можливість зазначати кількість потоків, які буде використано для повторювальних тестів. Приклад реалізації такого тесту на TestNG наведено у лістингу 16.

Лістинг *16 — Реалізація повторювального тесту з паралельним виконанням у TestNG*

```
@Test(description = "actionPerformed очищує значення полів
для введення", threadPoolSize = 2, invocationCount = 4)
public void actionPerformedShouldResetFieldsToEmptyValues()
{
    PortScanner portScanner = getInstance();
    portScanner.hostName.setText("localhost");
    portScanner.fromPort.setText("0");
    portScanner.toPort.setText("1024");
    portScanner.new
PortResetListener().actionPerformed(null);
    final PortScanner localPortScanner = portScanner;
    SwingUtilities.invokeLater(() -> {
        Assert.assertEquals("",
localPortScanner.hostName.getText());
        Assert.assertEquals("",
localPortScanner.fromPort.getText());
        Assert.assertEquals("",
localPortScanner.toPort.getText());});}
```

Тест, наведений у лістингу 16, виконається 4 рази у 2 різних потоках, результат можна побачити на рисунку 35, а результат спроби виконати повторювальний тест паралельно у JUnit 5 можна побачити на рисунку 36.

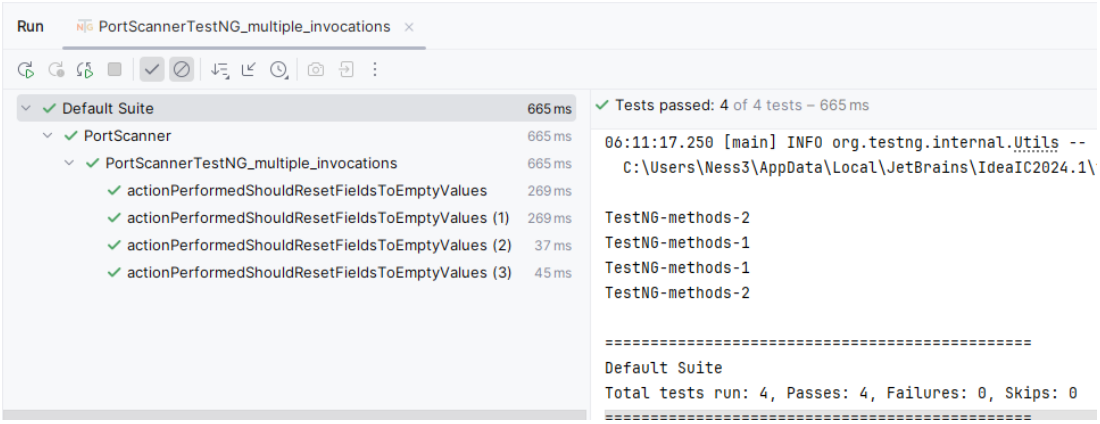

Рисунок *35* — *Результат виконання повторювального тесту в різних потоках на TestNG*

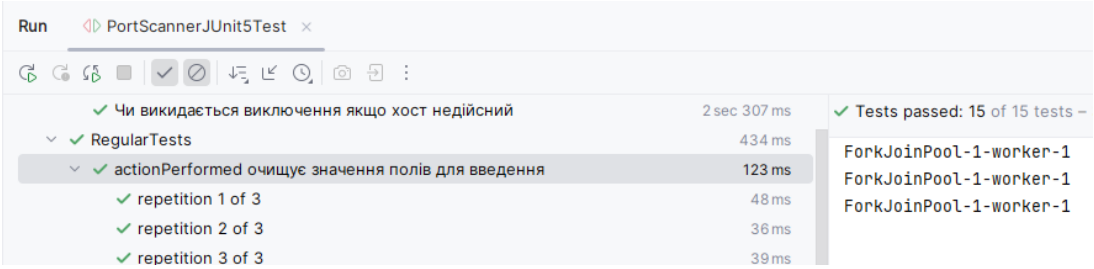

Рисунок *36* — *Спроба виконати повторювальний тест у паралельному режими на JUnit 5*

Як видно з рисунка 36, у JUnit 5 є можливість створювати повторювальні тести, але немає можливості виконувати такі тести паралельно.

3.2 Аналіз результатів проведеного порівняльного дослідження

Наприкінці порівняльного дослідження стали очевидними відмінності між фреймворками JUnit 5 та TestNG, а також були підкреслені їх слабкі та сильні сторони. Тож тепер є можливість чітко надати відповідь на таке питання як «В яких випадках той чи інший фреймворк буде ефективнішим».

Після огляду та застосування фреймворка JUnit 5 було виділено такі його переваги, як:

1. Архітектура: JUnit 5 має значну перевагу над архітектурою фреймворка TestNG через наявну підтримку тестування коду, який вже не підтримується та не оновлюється, але все ще активно експлуатується у проекті (іншими словами — legacy-код). Це стає можливим завдяки модулю JUnit Vintage, який створює зворотню сумісність старого і нового коду і запобігає виникненню конфліктів при використанні різних версій фреймворка JUnit.
2. Припущення: вони являють собою статичні методи для створення тестів з умовою, і вони є хорошим засобом запобігання роботи тестів, для виконання

яких потрібні специфічні умови середовища, при недотриманні яких запуск тестів буде марною витратою часу та ресурсів. У випадку, коли в проекті є потреба створювати залежні від операційної системи тести, припущення є незамінним засобом для того, щоб керувати процесом виконання таких тестів. Це також відноситься до залежності тестів від певних версій бібліотек, версії Java або версії самого фреймворка JUnit. Використання припущень значно зменшує кількість помилкових спрацьовувань тестів.

3. Анотація @TestInstance(): ця анотація максимально оптимізує процес тестування тоді, коли є необхідність оперувати «важким» екземпляром класу (тобто таким, на створення якого за раз витрачається багато часу та ресурсів, наприклад, великий файл або факт підключення до бази даних). Ця анотація змінює життєвий цикл тестів і дозволяє створити екземпляр класу один раз, після чого цей екземпляр буде використовуватись між тестовими методами без створення нових екземплярів для кожного метода окремо. У TestNG немає аналогічної анотації або атрибуту.

4. Більший набір тверджень: фреймворк JUnit 5 має у своїй колекції більше різноманіття тверджень, аналогів яких немає у TestNG, зокрема assertDoesNotThrow, assertAll, assertTimeout, assertIterableEquals тощо.

У той же час, порівняння такого засобу модульного тестування як TestNG з JUnit 5 допомогло відокремити його сильні сторони, зокрема:

1. Гнучкіше керування життєвим циклом тестів: можливість за рахунок великої різноманітності методів-фікстур, які працюють на різних рівнях (від рівня тестових методів до рівня тестових наборів), керувати життєвим циклом тестів може мати вирішальне значення при виборі фреймворка для тестування застосунка, де передбачена розробка великої кількості тестів (яка може сягати декількох тисяч і більше на один проект).

2. Підтримка груп тестів: можливість додавати тести до груп і включати одні групи в інші групи також є зручним засобом згрупувати тести і виконати в ізоляції від інших. До того ж надається можливість встановити передумови та післяумови для груп, а самі групи можуть бути створені як на рівні тестових

методів, так і на рівні тестових класів (тоді усі методи класу автоматично увійдуть у групу).

3. Наявність атрибутів: на відміну від JUnit 5, у TestNG є атрибути, які застосовуються безпосередньо в анотаціях. Це зменшує саму кількість застосованих до методів анотацій і також робить код компактнішим.

4. Паралельне виконання повторювальних тестів: у випадку коли повторювальний тест виконується довго або є трудомістким, є сенс розбити його на декілька потоків і виконувати паралельно, тим самим заощаджуючи час та ресурси при запуску саме таких тестів.

5. Наявність вбудованих звітів: у TestNG наявна генерація звітів після виконання тестового набору. Зокрема створюється html-файл «emailable-report», який можна відправити проектному менеджеру, розробникам та іншим учасникам команди за потреби. Створення таких звітів допомагає оперативно інформувати інших команду про стан та результат тестування застосунка на даному етапі.

6. Підтримка залежних тестів та груп: у ситуації, коли є необхідність прив’язати виконання тестових методів або груп тестів один до одного і змусити їх виконуватись у певному порядку, в якому падіння одного тесту має призводити до падіння усіх інших у черзі (тобто тести являють собою певні кроки, через які тестуються різні задіяні компоненти певного функціоналу) фреймворк TestNG готовий надати таку можливість.

7. Конфігураційні файли: використання файлів конфігурації дозволяє зручно і швидко, в одному місці налаштувати групування, параметризацію, конфігурацію паралельного виконання і фільтрацію тестів, що особливо стає у нагоді при вирішенні задач неперервної інтеграції.

Серед спільних можливостей обох фреймворків можна виділити підтримку тестів на тайм-аут, повторювальних, параметризованих, позитивних та негативних тестів, перехоплення виключень, використання тверджень, можливість виконувати тести паралельно.

Тож фреймворк JUnit 5 буде незамінним засобом забезпечення якості тестованих Java-застосунків у випадках, коли:

- Передбачається тестування проекту з так званою «важкою спадковістю», де є потреба працювати зі застарілим або просто не якісним кодом і старими тестами, бо на те, щоб оновити вже розроблені на попередній версії JUnit тести може піти багато часу та ресурсів, що виливається у збитки для компанії, тоді як використання JUnit 5 надає можливість старим тестам співіснувати з новими без конфліктів.
- Є необхідність прив’язувати тести до конкретної операційної системи, для якої або для яких розроблюється тестований застосунок, до версії Java або до версії бібліотек, що використовуються при розробці застосунка. Така необхідність виникає там, де є потреба тестувати застосунок за специфічних умов, без яких протестувати застосунок належним чином неможливо.
- Екземпляром для тестування виступає об’єкт, на постійне створення якого витрачається багато часу та ресурсів. Це може бути спроба підключення до бази даних, завантаження важких файлів, веб-сервіси, REST API (програмна архітектура), великі колекції даних тощо. JUnit 5 дозволяє створювати ці об'єкти лише один раз і використовувати їх між тестами, що значно оптимізує процес тестування.
- Передбачається розробка тестів для складних та масштабних проектів, де є потреба у складних логічних перевірках (наприклад групування декількох тверджень, перевіряти які потрібно в купі), перевірці коду на відсутність викидання виключень, рівність ітераторів тощо. Тобто проекти, де нестача варіативності тверджень може завадити добре покрити код тестами.

Перевагу при виборі фреймворку для тестування застосунка на Java слід надавати TestNG тоді, коли:

- Планується тестування масштабного проекту з великою кількістю тестів, де є потреба у гнучкому групуванні тестів з можливістю виконувати

групи ізольовано від інших і налаштовувати для них середовище виконання індивідуально до їх потреб.

- Потрібен детальний контроль над життєвим циклом тестів, а також підготовкою і очищенням тестового середовища, що є критично важливим для великих проектів, де передумови та післяумови виконання можуть відрізнятись в залежності від рівня, до якого застосовуються (наприклад при роботі з важкими об’єктами) і там, де є місце для використання різних конфігурацій, а також це важливо для більш якісної підтримки груп тестів.
- Є необхідність у створенні та поширенні звітів про результати тестування, а така необхідність часто запроваджується в IT-фірмах. Без можливості створити такі звіти буде важче та витратніше за часом інформувати залучених до тестування осіб.
- Передбачається розробка тестових сценаріїв, які мають виконуватись ланцюжком, де виконання одного тесту залежить від успішного виконання попереднього, тобто таких сценаріїв, для яких потрібно задіяти більше ніж одну функцію застосунка, і в зазначених сценаріях є сенс перевірити спільну роботу функцій, а не в ізоляції одна від одної.
- Йде робота з великою кількістю тестів, для яких раціонально і зручно було б визначити налаштування в одному місці аніж перед кожним тестовим методом окремо, у чому постає гостра потреба у разі, наприклад, запровадження неперервної інтеграції.

3.3 Висновки до розділу 3

1. Було розроблено спроектовані тести для обох застосунків із застосуванням фреймворків JUnit 5 та TestNG.
2. Виконано порівняння реалізації позитивних та негативних тестів, тестів на виключення та на перебільшення тайм-ауту, параметризованих

тестів, тестів за умовою, групування та паралельного виконання тестів, а також перевірено наявність вбудованих звітів у обох фреймворках.

3. Визначено як спільні можливості, що присутні і в JUnit 5 і в TestNG, так і ті, що наявні лише в тому чи іншому фреймворку і являють собою його переваги.
4. Було наведено рекомендації, де зазначено при яких потребах і в яких випадках застосування того чи іншого фреймворка буде ефективніше.

ВИСНОВКИ

1. При виконанні кваліфікаційної роботи було оглянуто принципи і засоби модульного тестування, а також було здійснено огляд аналогічних фреймворків модульного тестування Java-застосунків.
2. Перед тестуванням було оглянуто можливості фреймворків JUnit 5 та TestNG, впроваджені у них життєві цикли тестів а також зіставлено наявні у них анотації.
3. Було обрано застосунки, при тестуванні яких у подальшому виконувалось порівняльне дослідження можливостей JUnit 5 та TestNG.
4. Спроектовано і розроблено тести, за допомогою яких вдалось порівняти можливості обраних фреймворків модульного тестування Java-застосунків, а саме такі можливості як розробка позитивних, негативних та параметризованих тестів, тестів на виключення, перебільшення тайм-ауту та тестів за умовою, групування і паралельне виконання тестів, а також можливість генерувати звіти на обох фреймворках.
5. До спільних можливостей обраних фреймворків можна віднести підтримку тестів із тайм-аутами, повторювальних, параметризованих тестів, позитивних і негативних тестів, перехоплення виключень, використання тверджень, а також можливість паралельного виконання тестів.
6. Після проведення огляду та використання фреймворка JUnit 5 на практиці було встановлено його ефективність при необхідності роботи з legacy-кодом і потреби суміщати старі тести з тестами, написаними на новіших версіях JUnit, при потребі налаштувати специфічні умови середовища, в яких планується виконувати тести, при роботі з ресурсномісткими екземплярами для тестування, а також великий набір тверджень, який дозволяє створювати різноманітніші тестові сценарії на різні випадки та потреби з боку фахівця забезпечення якості програмного забезпечення.

7. Порівняння фреймворка TestNG з JUnit 5 допомогло висвітлити його ефективність при роботі з великою кількістю тестів, які є необхідність групувати, незамінність у можливості швидко і гнучко налаштовувати виконання тестів у разі вирішення задач неперервної інтеграції, можливість при потребі регулярно і швидко отримувати детальні звіти після виконання наборів тестів з результатами тестування, а також забезпечення можливості створювати залежні тести, що формують собою складні тестові сценарії, які неможливо реалізувати простими та ізольованими один від одного тестами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Шевцова В.С., Коломоєць Г.П. Порівняльне дослідження тестових фреймворків JUnit 5 та TestNG для організації тестування Java-застосунків. Збірник наукових праць студентів, аспірантів, докторантів і молодих вчених «Молода наука-2024» / Запорізький національний університет. – Запоріжжя : ЗНУ, 2024. – Т.5. – С. 211.
2. Принципи тестування. Qalight. URL: https://qalight.ua/baza- znaniy/printsipi-testuvannya/ (дата звернення: 31.03.2024).
3. Важливість тестування програмного забезпечення: типи, підготовка, інструменти. Itstep. URL: https://cloud.itstep.org/blog_3/importance-of-software-testing-types-preparation-tools#1 (дата звернення: 05.04.2024).
4. Let’s Talk about Test Levels. Medium. URL: https://numanhanduran.medium.com/lets-talk-about-test-levels-f93cb4aa1b98 (дата звернення: 05.04.2024).
5. What is Unit Testing? A Complete Guide. Katalon. URL: https://katalon.com/resources-center/blog/unit-testing (дата звернення: 05.04.2024).
6. Boni Garcia, Mastering Software Testing with JUnit 5. Packt Publishing, 2017. 177 с.
7. Параметризовані тести у Junit. JavaRush. URL: https://javarush.com/ua/quests/lectures/ua.questservlets.level03.lecture04 (дата звернення: 18.04.2024).
8. Catalin Tudose, JUnit in Action. Manning Publications Co; 3rd edition, 2020. 466 с.
9. Cedric Beust, Next Generation Java Testing: TestNG and Advanced Concepts. O'Reilly Media; 1st edition, 2007. 325 с.
10. Mockito framework site. Mockito. URL: https://site.mockito.org/ (дата звернення: 27.04.2024).

11. PowerMock. GitHub. URL: https://github.com/powermock/powermock (дата звернення: 27.04.2024).
12. Java Hamcrest. Hamcrest. URL: https://hamcrest.org/JavaHamcrest/ (дата звернення: 27.04.2024).
13. AssertJ - fluent assertions java library. AssertJ. URL: https://assertj.github.io/doc/#assertj-core-assertions-guide (дата звернення: 28.04.2024).
14. Spock the enterprise ready specification framework. Spockframework. URL: https://spockframework.org/ (дата звернення: 28.04.2024).
15. JUnit Pioneer: JUnit 5 extension pack, pushing the frontiers on Jupiter. junit-pioneer. URL: https://junit-pioneer.org/junit-pioneer/ (дата звернення: 06.05.2024).
16. JUnit 5 features that every Java Developer should know. Medium. URL: https://medium.com/@fullstacktips/junit-5-features-that-every-java- developer-should-know-by-now-792d13a40b80 (дата звернення: 08.05.2024).
17. JUnit 5 Test LifeCycle. HowToDoInJava. URL: https://howtodoinjava.com/junit5/junit-5-test-lifecycle/ (дата звернення: 08.05.2024).
18. Architecture of the TestNG framework. Scribd. URL: https://www.scribd.com/document/466343504/ARCHITECTURE-OF-THE-TestNG-FRAMEWORK (дата звернення: 10.05.2024).
19. TestNG Annotations Tutorial With Examples For Selenium Automation. lambdatest. URL: https://www.lambdatest.com/blog/complete-guide-on- testng-annotations-for-selenium-webdriver/ (дата звернення: 10.05.2024).
20. James Whittaker, Exploratory software testing. Artech House, 2004. С. 233.

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Шевцова Ванеса Сергіївна, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz20bd-212@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Порівняльне дослідження тестових фреймворків JUnit 5 та TestNG для організації тестування Java-застосунків**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 17.06.2024 _____ Шевцова Ванеса Сергіївна
(студент)

Дата 18.06.2024 _____ Коломоєць Геннадій Павлович
(науковий керівник)