Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

Магістр
(назва освітнього ступеня)

на тему: **Методи і засоби тестування ІТ-продукту, побудованого на мікросервісній архітектурі**

Виконав: студент(ка) 6 курсу, групи СІм-61
спеціальності 123 «Комп'ютерна інженерія»
(шифр і назва спеціальності)

		Свергун С.М.
	(підпис)	(прізвище та ініціали)
Керівник		Жаровський Р.О.
	(підпис)	(прізвище та ініціали)
Нормоконтроль		Тиш Є.В.
	(підпис)	(прізвище та ініціали)
Завідувач кафедри		Осухівська Г.М.
	(підпис)	(прізвище та ініціали)
Рецензент		
	(підпис)	(прізвище та ініціали)

Тернопіль
2022

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Осухівська Г.М.
(підпис) (прізвище та ініціали)
« » 2022 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня *магістр*
(назва освітнього ступеня)

за спеціальністю *123 «Комп'ютерна інженерія»*
(шифр і назва спеціальності)

студенту *Свергуну Сергію Михайловичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Методи і засоби тестування ІТ-продукту, побудованого на мікросервісній архітектурі*

Керівник роботи *Жаровський Руслан Олегович, к.т.н.*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*06*» *грудня* *2022* року № *4/7-986*
2. Термін подання студентом завершеної роботи *22.12.2022 р.*
3. Вихідні дані до роботи *Методика тестування ПЗ, мікросерівісна архітектура ПЗ, використання тестових дублерів*

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1. Огляд теоретичних основ тестування ІТ - продукту побудованого на мікросервісній архітектурі
2. Аналіз методів автоматизованого тестування ІТ - продукту, побудованого на мікросервісній архітектурі
3. Апробація методів проведення автоматизованого тестування ІТ - продукту побудованого на мікросервісній архітектурі
4. Охорона праці та безпека в надзвичайних ситуаціях
Висновки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Актуальність і мета дослідження.
2. Задачі дослідження, об'єкт і предмет, наукова новизна і практична цінність дослідження.
3. Особливості мікросервісної архітектури
4. Критерії і методи тестування
5. Процес розробки і тестування мікросервісів
6. Діаграма варіантів використання процесу тестування мікросервісу
7. Результати експериментального дослідження.
8. Висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека в надзвичайних ситуаціях			
Основи охорони праці			

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	*Аналіз сучасних методів і технологій автоматизованого тестування програмного забезпечення*	*14.11.2022-20.11.2022*	*виконано*
2	*Методи і сценарії автоматизованого тестування програмних продуктів побудованих на мікросервісній архітектурі*	*20.11.2022 – 27.11.2022*	*виконано*
3	*Апробація методів автоматизованого тестування програмних продуктів побудованих на мікросервісній архітектурі*	*27.11.2022 – 04.12.2022*	*виконано*
4	*Аналіз ефективності використання тестових дублерів при автоматизованому тестуванні програмних продуктів побудованих на мікросервісній архітектурі*	*04.12.2022– 08.12.2022*	*виконано*
5	*Охорона праці та безпека в надзвичайних ситуаціях*	*08.12.2022-12.12.2022*	*виконано*
6	*Оформлення пояснювальної записки і графічного матеріалу*	*12.12.2022-14.12.2022*	*виконано*
7	*Попередній захист кваліфікаційної роботи магістра*	*14.12.2022*	*виконано*
8	*Захист кваліфікаційної роботи магістра*	*22.12.2022*	

Студент _____ *Свергун С.М.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Жаровський Р.О.*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Методи і засоби тестування ІТ-продукту, побудованого на мікросервісній архітектурі // Кваліфікаційна робота // Свергун Сергій Михайлович // ТНТУ, комп'ютерна інженерія, група СІм-61 // Тернопіль, 2022 // с. – 85, рис. – 31, табл. – 3, аркушів А1 – 8, додат. – 2 , бібліогр. – 29.

Ключові слова: ІТ-продукт, мікросервісна архітектура, тестування, заглушки, тестовий дублер.

У кваліфікаційній роботі магістра досліджено методи і засоби тестування ІТ-продукту, побудованого на мікросервісній архітектурі. Для розглянутих методів визначено основні концепції та етапи проведення тестування.

Проведено огляд теоретичних основ тестування хмарного програмного продукту побудованого на мікросервісній архітектурі. При такій організації розробки програмних продуктів доцільно використовувати автоматизоване тестування, де відповіді на запити досліджуваного мікросервісу посилатимуть тестові дублери, що імітують зв'язок із підсистемами, що взаємодіють між собою.

Відповідно в другому розділі розглянуті методи автоматизованого тестування і їх можливості при тестуванні ІТ-продукту, побудованого на мікросервісній архітектурі. В результаті вивчення методологій розробки програмних продуктів виявлено, що методологія BDD дозволяє зробити процес тестування зручнішим, дешевим і корисним, підвищуючи якість готового ІТ - рішення.

Складено сценарій проведення автоматизованого тестування хмарного програмного продукту, збудованого на мікросервісній архітектурі з використанням тестових дублерів. В рамках апробації виділених методів, сценарію та інструментального засобу для проведення автоматизованого тестування було складено набір тест-кейсів для тестування змодельованого хмарного програмного продукту, побудованого на мікросервісній архітектурі. Описано хід та результат тестування.

ABSTRACT

Methods and means of testing an IT product built on a microservice architecture // Master thesis // Sverhun Serhii Mykhailovych // TNTU, computer engineering, group СІм-61 // Ternopil, 2022 // p. – 85, fig. – 31, tab. - 3, sheets A1 - 8, add. – 2, bibliography. - 29.

Keywords: IT product, microservice architecture, testing, stubs, test backup.

The master's thesis researched the methods and means of testing an IT product built on a microservice architecture. The main concepts and stages of testing are defined for the considered methods.

An overview of the theoretical foundations of testing a cloud software product built on a microservice architecture was conducted. With such an organization of the development of software products, it is advisable to use automated testing, where answers to the requests of the microservice under investigation will be sent by test duplicates simulating communication with subsystems that interact with each other.

Accordingly, in the second section, the methods of automated testing and their possibilities in testing an IT product built on a microservice architecture are considered. As a result of the study of software product development methodologies, it was found that the BDD methodology makes the testing process more convenient, cheap and useful, increasing the quality of the finished IT solution.

A script for automated testing of a cloud software product built on a microservice architecture with the use of test backups has been compiled. As part of the approbation of the selected methods, scenario and tool for automated testing, a set of test cases was compiled for testing a simulated cloud software product built on a microservice architecture. The course and result of testing are described.

ЗМІСТ

ПЕРЕЛІК ОСНОВНИХ УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ І СКОРОЧЕНЬ

BDD англ. Behavior Driven Development розробка з урахуванням поведінки

QA англ. Quality Assurance забезпечення якості

STLC англ. Software Testing Life Cycle життєвий цикл тестування програмного забезпечення

ІТ інформаційні технології

ОС операційна система

ПЗ програмне забезпечення

ВСТУП

Актуальність теми. Останнім часом особливої актуальності у сфері інформаційних технологій набувають питання якості програмного продукту, що пов'язано з великою конкуренцією серед компаній-розробників ІТ-продуктів.

Основним етапом життєвого циклу програмного продукту, є виявлення вимог, параметрів та критеріїв, які забезпечуватимуть функціонал і якість програмного продукту, яка необхідна замовнику. Одним з методів управління якістю у сфері інформаційних технологій є тестування програмного продукту.

Виявлення та усунення помилок під час супроводу програмного продукту спричиняє додаткові витрати в середньому в 200 разів більше, ніж на стадії тестування, а у разі пізнього виявлення дефектів підсумковий бюджет проекту може збільшитися до 40%.

З метою визначення критерію відповідності готового програмного продукту заявленим характеристикам необхідно протестувати розроблений ІТ-продукт різними методами. Це дозволить виявити помилки, усунення яких підвищить якість продукту, що розробляється.

Традиційні методи ручного тестування не можуть забезпечити необхідну якість сучасних програмних систем. До основних недоліків ручного тестування можна віднести:

- людський фактор,
- трудомісткість повторного тестування програми після внесення змін,
- неможливість тестування навантаження.

В зв’язку з чим все частіше використовують автоматизоване тестування. Головною перевагою якого є зменшення часу на тестування програмних модулів, оскільки опрацювати величезний обсяг даних вручну набагато важче, ніж один раз скласти сценарій виконання тестування. З кожним днем з'являється все більше методів, інструментів та технологій автоматизованого тестування, спрямованих на покращення якості та зменшення витрат на етап тестування ІТ-продуктів.

Крім того, створений тестовий сценарій доцільно застосовувати далі для організації такого ж виду тестування, що у майбутньому значно заощадить час. Важливим фактором можна відзначити те, що при автоматизованому тестуванні можливо швидко змоделювати навантаження на всю систему, що забезпечить різносторонню перевірку коректності функціонування системи і, отже, гарантує високу якість програмного продукту.

У міру того, як нові тенденції в різних областях з'являються з кожним днем, потрібна нова архітектура для різних додатків.

Користувачам потрібен інтерактивний, багатий та динамічний досвід роботи програм на різних платформах. Цим вимогам задовольняють програми, що мають високу доступність, масштабованість та простоту виконання на хмарній платформі.

Більшість організацій хочуть оновлювати свої програми часто, кілька разів на день. Монолітні програми мають обмеження для підтримки таких вимог.

Мікросервіс є новою архітектурою, в якій великі і складні програмні продукти складаються з одного або декількох невеликих сервісів. Вони можуть працювати незалежно один від одного. Такі мікросервіси слабо пов'язані один з одним, кожен з них відповідає за ефективне виконання лише одного завдання. Мікросервіси дуже корисні для програм у хмарних обчисленнях. Використання мікросервісів у хмарних обчисленнях дозволяє підвищити популярність хмари. Використання мікросервісів пропонує більше варіантів та можливостей для незалежного розвитку сервісу. У 2022 році воно покликане прискорити процес розробки мобільних та веб-додатків.

Оскільки організації створюють дедалі більше мікросервісів, дедалі частіше стикаються з проблемами інтеграції. Наслідки можуть бути різними – від часу на розробку нового функціоналу, до повністю неробочої програми. У крайніх випадках помилки узгодженості API – інтерфейсів виявляються лише під час експлуатації програми, і ведуть до довгої та дорогої його доопрацювання та нового циклу тестування.

Метою кваліфікаційної роботи є створення ефективного алгоритму проведення автоматизованого тестування програмного продукту, побудованого на

мікросервісній архітектурі, для підвищення якості програмних додатків та оперативності доставки оновлень замовнику.

Задачі кваліфікаційної роботи:

1. Розглянути теоретичні аспекти існуючих засобів та методів автоматизованого тестування IT - продукту, побудованого на мікросервісній архітектурі;

2. На основі розглянутої теорії проаналізувати методики проведення автоматизованого тестування хмарного IT -продукту;

3. Розробити алгоритм проведення автоматизованого тестування хмарного IT -продукту, побудованого на мікросервісній архітектурі, що дозволяє підвищити якість та швидкість доставки оновлень замовнику.

Відповідно до цілей та завдань дисертаційної роботи визначено її об'єкт та предмет.

Об’єкт дослідження: методи проведення автоматизованого тестування програмного продукту, побудованого на мікросервісній архітектурі.

Предмет дослідження: розробка сценарію автоматизованого тестування програмного продукту, збудованого на мікросервісній архітектурі.

Методи дослідження: Метод моделювання, який використовується для побудови моделі IT-продукту та аналізу його властивостей на основі побудованої моделі. Метод абстрагування, який дозволяє виключити з розгляду при тестуванні незначні властивості об'єкта та приділити увагу найбільш значущим характеристикам об'єкта. Метод візуалізації даних, що використовується для побудови графіків і схем, що дозволяє наочно представляти отримані результати дослідження.

Наукова новизна дослідження полягає у розробці методу автоматизованого тестування IT-продукту, побудованого на мікросервісній архітектурі, із застосуванням інтелектуального тестового дублера.

Теоретична значущість полягає у виявленні нового способу проведення автоматизованого тестування хмарного IT-продукту, побудованого на мікросервісній архітектурі, з використанням тестового дублера. Отримані висновки

можуть бути використані для подальшого дослідження факторів, що впливають на якість ІТ-продукту.

Практичне значення результатів кваліфікаційної роботи полягає у способах застосування тестового дублера, а також способах його інтеграції у стратегію автоматизованого тестування ІТ-продукту.

Публікації. Результати дослідження апробовано на X науково-технічній конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі, системи та технології» (7-8 грудня 2022 року) у вигляді тез конференцій.

1. Свергун С., Жаровський Р. Тестування програмного забезпечення побудованого на мікросервісній архітектурі. Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі системи та технології» (7-8 грудня 2022 року). Тернопіль: ТНТУ. 2022. С. 92.

2. Свергун С., Жаровський Р. Тестування програмного продукту, побудованого на мікросервісній архітектурі на основі BDD. Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі системи та технології» (7-8 грудня 2022 року). Тернопіль: ТНТУ. 2022. С. 93.

Структура роботи. До складу кваліфікаційної роботи магістра входить розрахунково-пояснювальна записка та графічний матеріал. Розрахунково-пояснювальна записка містить вступ, 4 розділи, загальні висновки, список використаної літератури і додатки. Обсяг роботи: розрахунково-пояснювальна записка – 85 арк. формату А4, графічна частина – 8 аркушів формату А1.

РОЗДІЛ 1
ОГЛЯД ТЕОРЕТИЧНИХ ОСНОВ ТЕСТУВАННЯ ІТ - ПРОДУКТУ ПОБУДОВАНОГО НА МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

1.1. Характеристики та моделі хмарних обчислень

Хмарні обчислення слід розглядати як засіб для використання зручного та широкодоступного мережевого доступу до загального сховища обчислювальних ресурсів. Такими ресурсами можуть бути мережі, сервери, додатки та послуги, які надаються на вимогу замовника та можуть бути використані з невеликими зусиллями щодо управління або взаємодії з постачальником послуг хмарного сховища.

Виділяють п'ять важливих характеристик та три моделі обслуговування хмарних моделей.

До основних важливих характеристик хмарних моделей відносяться:

− самообслуговування на вимогу. Клієнт самостійно конфігурує обчислювальні можливості (наприклад, серверний час та мережеве сховище), без взаємодії з постачальником послуг;

− широкий доступ до мережі. Мережа може використовуватися через стандартні механізми на гетерогенних платформах тонких або товстих клієнтів (наприклад, мобільними телефонами, планшетами, ноутбуками та робочими станціями);

− об'єднання ресурсів. Використовуючи багатоклієнтську модель, можна поєднати обчислювальні ресурси провайдера. Різні фізичні та віртуальні ресурси можуть динамічно призначатися та перерозподіляються залежно від вимог споживачів.

Незважаючи на те, що клієнт, як правило, не може контролювати і знати точне розташування використовуваних ресурсів, він може вказати місце на більш високому рівні абстракції (наприклад, країна, область або конкретний центр

обробки даних). Як приклади ресурсів можна розглядати зберігання, обробку, пам'ять і пропускну здатність мережі;

− еластичність. Ця характеристика передбачає гнучке налаштування хмарних потужностей, можливість їх скоротити у разі потреби. Відповідно до попиту, таке скорочення може статися автоматично для більш швидкого масштабування системи. Для споживача хмарні обчислювальні ресурси не обмежені та можуть бути надані будь-якої миті;

− облік споживання. Автоматичний контроль та оптимізація використовуваних ресурсів у хмарних системах відбувається на певному рівні абстракції, який відповідає типу послуги (таких як зберігання, обробка, пропускна здатність та активні облікові записи користувачів). Використання таких послуг можна легко відстежити та проконтролювати. З метою забезпечення прозорості для постачальника та споживача складаються звіти щодо даних послуг.

NIST виділяє три сервісні моделі хмарних систем:

− інфраструктура як сервіс (IaaS),

− платформа як сервіс (PaaS),

− програмне забезпечення як сервіс (SaaS).

Кожна сервісна модель побудована з наступних ресурсів:

− апаратні засоби – сервери, мережеві пристрої, сховища даних;

− віртуалізація – комплекс програмно-апаратних ресурсів які забезпечують виконання різноманітних програмних процесів на певному фізичному ресурсі [25];

− платформа – програмне забезпечення, необхідне для запуску програми (Apach, PHP, .NET, JRE);

− додаток – програма, створена для вирішення бізнес-завдань.

Конфігурація ІТ-продукту до застосування хмарних послуг представлена на рис. 1.1.

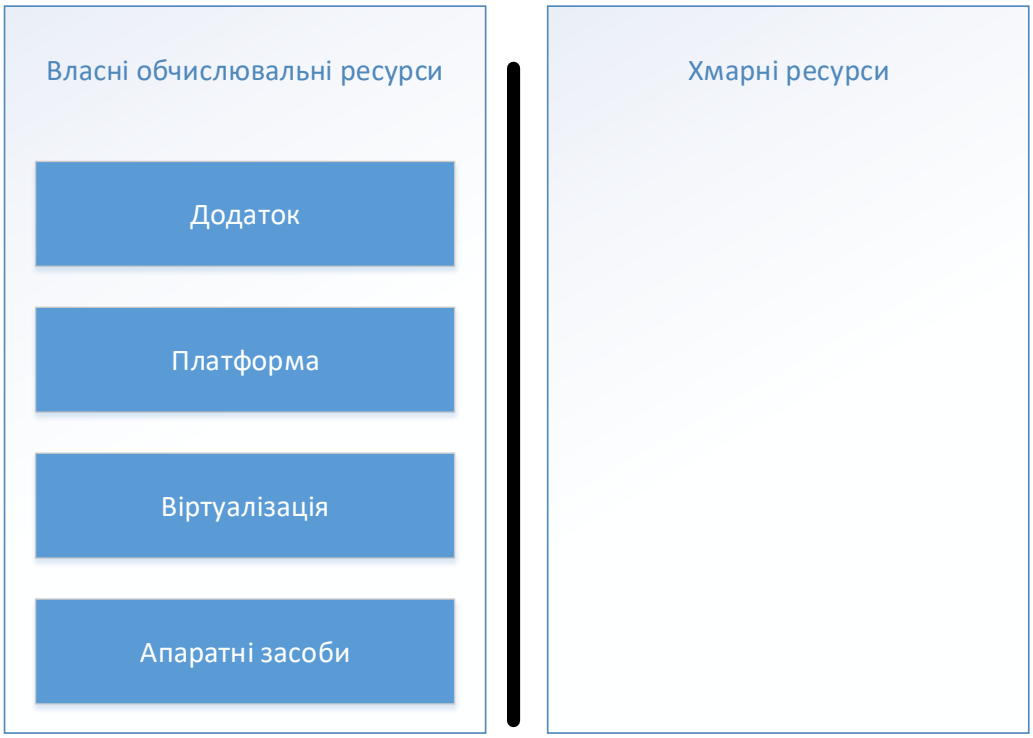

Рис.1.1. Конфігурація рішення до впровадження хмарних послуг

Відповідно до рис. 1.1 всі складові IT-продукту є власними обчислювальними ресурсами [27].

Розглянемо конфігурацію IT-продукту у разі використання хмарних сервісів:

– інфраструктура як сервіс (IaaS): покупець отримує доступ до обчислювальної потужності, сховища даних, мережевих ресурсів, він не управляє хмарною інфраструктурою, але може вирішувати, які операційні системи запускати на виділеній потужності, обсяг дискового простору та додатки, які будуть встановлені на операційну систему. Замовник також може впливати на конфігурацію мережевих компонентів, таких як брандмауер. Віртуалізація також стає частиною хмари. Приклад: Amazon пропонує оплату за фактом використання більш ніж 160 хмарних сервісів. Є можливість оплачувати окремі необхідні послуги, причому оплачувати лише за час їх фактичного використання;

– платформа як сервіс (PaaS): можливе використання програмних мов та інструментів, що підтримуються постачальником PaaS. Замовник вже не має контролю над інфраструктурою, в якій працює програма, але може її конфігурувати під бізнес-завдання. Програма залишається під повним контролем замовника.

Приклад: веб-хостинг є відомим сервісом PaaS, який дозволяє розгортати сайти на основі різних технологій, і використовувати бази даних, які не потрібно встановлювати самостійно;

– програмне забезпечення як сервіс (SaaS): замовник може використовувати технології від постачальника послуг, що працюють у хмарній інфраструктурі. Дані програми доступні з браузера для всіх, будь-де, будь-коли, на будь-якому пристрої. Все, що може змінити замовник – локалізацію та візуальне відображення продукту. Приклад: поштовий сервіс – типовий представник SaaS. Користувачі не знають, як і де зберігаються листи, але сервіс з усім вмістом їм завжди доступний.

Конфігурація на базі IaaS представлена на рис. 1.2. Сервісний рівень IaaS передбачає, що віртуалізація та апаратні засоби розташовуються у хмарному просторі [28]. Компанії можуть на вимогу як збільшити, так і зменшити потужність, що орендується, уможливлюючи обробку максимальних навантажень без капітальних витрат.

Рис. 1.2. Конфігурація рішення із сервісним рівнем IaaS

Конфігурація рішення лише на рівні PaaS представлена рис. 1.3.

Рис. 1.3. Конфігурація рішення із сервісним рівнем PaaS

Відповідно до рисунка 1.3 у власності замовника залишається лише додаток, а платформа, віртуалізація та апаратні засоби стають частиною хмарного простору.

Конфігурація рішення із сервісним рівнем SaaS представлена рис. 1.4.

Рис.1.4. Конфігурація рішення з сервісним рівнем SaaS

Відповідно до рисунка 1.4, при використанні сервісного рівня SaaS всі ресурси знаходяться в хмарному просторі, вони відкриті і доступні в будь-який час і на будь-якому пристрої.

Таким чином, розглянувши всі три моделі використання хмарного простору, можна сформулювати висновок про те, що за будь-яких вимог замовника та можливостей IT-компанії можна вибрати зручну конфігурацію використання хмарного простору. Зручний та швидкий доступ до ресурсів дозволить з мінімальними зусиллями контролю чи погодження з постачальниками послуг розробити та протестувати IT-продукт.

1.2. Особливості мікросервісної архітектури

У міру того, як нові тенденції в різних областях з'являються з кожним днем, потрібна нова архітектура для різних додатків. IT-компаніям потрібен підхід до розробки, що відповідає таким вимогам: крос-платформні додатки, здатні працювати швидко та якісно на різних пристроях; часті оновлення; швидке виведення рішення ринку – мінімальний час від виникнення ідеї до першої робочої версії; оптимальний вибір технологій, що найбільш підходить для вирішення конкретних проблем; широке перевикористання функціональності.

Мікросервісна архітектура - модель архітектури програмного забезпечення, яка орієнтована на взаємодію невеликих, слабко пов'язаних між собою компонент-мікросервісів. Такі модулі легко змінюються, можуть працювати незалежно один від одного, кожен з них відповідає за ефективне виконання лише однієї бізнес-функції.

Вирізняють такі особливості мікросервісів:

− невеликий розмір, орієнтованість на те, щоб якісно виконувати лише одну роботу;

− межі мікросервісів (модулів) визначаються бізнес-кордонами, які визначають область коду для конкретних функцій та виконуваних операцій;

− ізольованість: мікросервіс можна розгорнути у хмарному просторі, наприклад - Platform asa Service (PAAS). При іншому розгляді може бути розглянутий як процес своєї власної операційної системи;

− гнучкість розгортання: дозволяє не просто вибрати найпростіший інструмент для всієї системи, а індивідуально для кожного модуля визначити оптимальну технологію розгортання, не обмежуючись універсальним алгоритмом для всієї програми. У ситуації, коли якомусь модулю проектованої системи необхідна вища продуктивність, доцільно виділити інший набір технологій, зручніший задля досягнення необхідного рівня продуктивності. Іноді виправданим є рішення про зміну способу зберігання даних для різних підсистем;

− стійкість: у ситуаціях, коли непередбачено відмовляє один із блоків підсистеми, але без наступних відмов, непрацюючий модуль слід ізолювати, але зберегти працездатність усієї іншої системи. Мікросервіси можуть бути написані різними мовами та можуть використовувати різні технології зберігання даних;

− масштабування: у разі застосування традиційних великих монолітних архітектур програму, що розробляється, доводиться розширювати всю відразу. Так відбувається тому, що один невеликий блок обмежений у продуктивності, але якщо через нього затримується відгук всього величезного монолітного додатку, то розширювати слід все як єдине ціле. У ситуації з невеликими модулями розширюють лише деякі, ті, які справді цього потребують. Такий підхід дозволяє запускати решту блоків системи на менш потужному обладнанні. При роботі з системами надання послуг на вимогу, наприклад, тих, які надає Amazon Web Services, часто використовують таке масштабування на вимогу для тих модулів програми, які цього потребують. Це дозволяє суттєво скоротити витрати;

− простота розгортання: оскільки мікросервіси побудовані навколо бізнес-функцій, їх можна розгорнути незалежно друг від друга через повністю автоматизований механізм розгортання. Можливо змінити лише один мікросервіс та провести розгортання ізольовано від усієї системи. Такий підхід дає змогу оновлювати мікросервіс швидше. Можна швидко ізолювати модель, що не працює, і повернутися на попередню версію розробки, якщо виникли якісь неполадки в

оновленій системі. Крім того, новий функціонал можна набагато швидше донести до замовника. Та особливість, що мікросервісна архітектура дозволяє усунути максимальну кількість неполадок і затримок при запуску програми в експлуатацію, і стала однією з основних причин, через які такі організації, як Amazon і Netflix, скористалися даною технологією.

Однією з важливих переваг, що з'являється при використанні мікросервісної архітектури, є можливість повторного використання функціональності. Розробка на мікросервісної архітектурі дозволяє по-різному використовувати будь-яку функціональну можливість у різних цілях [26].

При розробці та тестуванні сучасних IT-систем, питання ізольованості від зовнішніх сервісів завжди є актуальним. Зовнішній сервіс може бути недоступний на етапі розробки або його функціонал розробляється паралельно і на нього не можна покладатися. Особливо гостро це питання постає на етапі планування автоматизованого тестування, оскільки перевіряти необхідно як штатне поведінка своєї системи, а й виняткові випадки: наприклад, недоступність зовнішнього сервісу, чи випадок, коли зовнішній сервіс відповідає помилкою.

Навіть за мінімальної залежності від зовнішніх сервісів, мікросервісна архітектура передбачає поетапну розробку та тестування мікросервісів. У такому разі зовнішньою залежністю вже є взаємозв'язок із сусіднім мікросервісом, який може бути ще на стадії планування. У цій ситуації доцільно використати тестовий дублер.

Джерард Мезарос (Gerard Meszaros) запропонував термін «тестовий дублер» (TestDouble) – загальне ім'я для об'єктів, що використовуються для заміни справжніх мікросервісів IT-продукту з метою більш ефективного тестування. Тестовий дублер - загальний термін, для точніших реалізацій використовуються спеціальні імена. Виділяють 3 категорії тестових дублерів:

− фіктивний модуль (fake) - найпростіший, найпримітивніший тип тестового дублера, не виконує жодних функцій, а просто реалізує певний інтерфейс;

− заглушка (stubs). Складніше фіктивного модуля, являють собою функцію відображення вхідних параметрів у вихідних. Заглушки є мінімальними

реалізаціями інтерфейсів та базових класів. Методи, що повертають порожнечу, зазвичай не містять реалізації взагалі, тоді як методи, що повертають значення, зазвичай повертають жорстко визначені значення. Використовується для того, щоб тестувався сам метод класу, що тестується;

– імітація (mock) – використовується для імітації роботи деякого класу або модуля. Імітація містить складніші реалізації, зазвичай здійснюючи взаємодії між різними членами успадкованого нею типу.

Логічна схема програми, де потрібна авторизація користувача, з використанням фіктивного модуля представлена рис. 1.5.

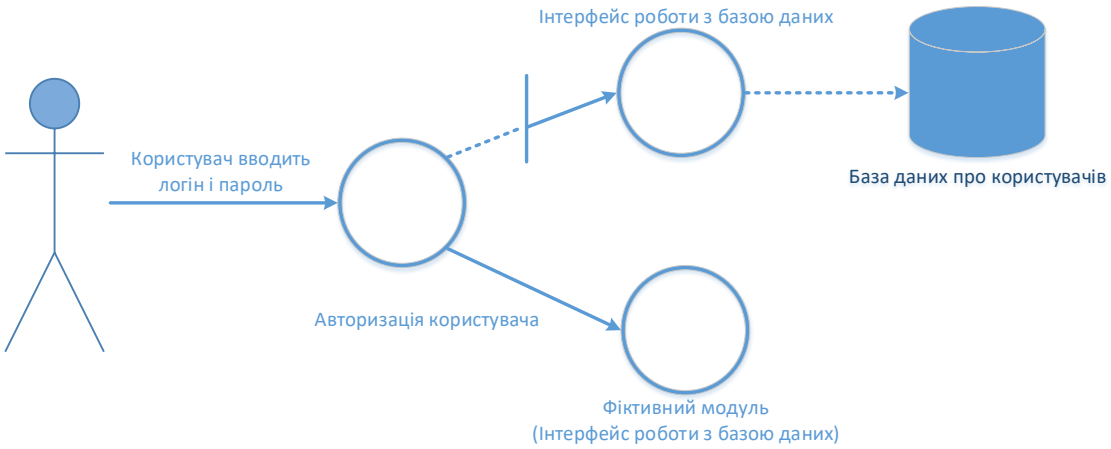

Рис. 1.5. Логічна схема роботи програми з використанням фіктивного модуля

Відповідно до рисунка 1.5 фіктивні модулі не містять реалізації і в основному використовуються, коли необхідні тільки значення параметрів.

Порожні значення можуть вважатися фіктивними модулями, але фіктивні модулі як такі є похідними від інтерфейсів та базових класів, не обтяженими реалізацією.

Логічна схема роботи програми з отримання середніх оцінок студентів із використанням заглушки представлена рис. 1.6.

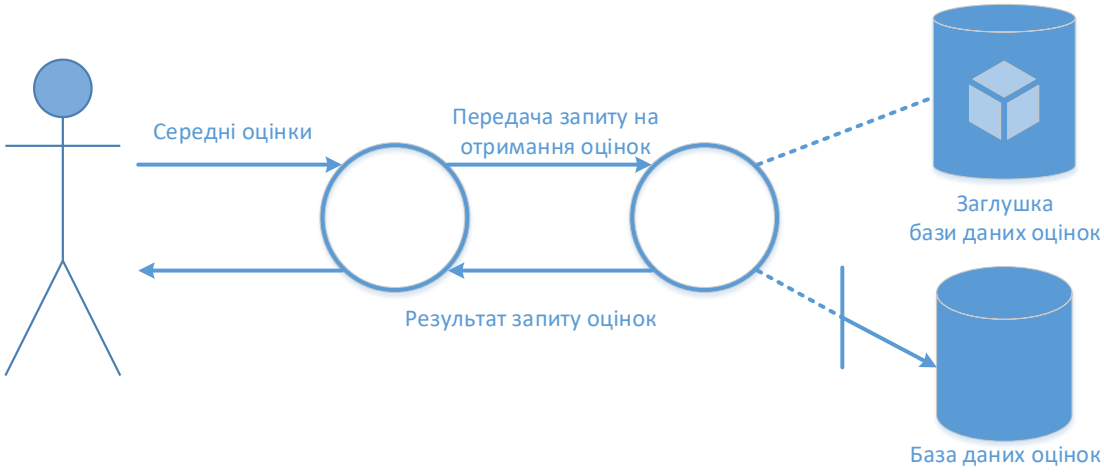

Рис. 1.6. Логічна схема роботи програми з використанням заглушки

Відповідно до рисунка 1.6 основне завдання заглушки - повернення до тестованого модуль деякого детермінованого значення.

Логічна схема роботи програми, яка надсилає повідомлення, з використанням імітації представлена на рис. 1.7.

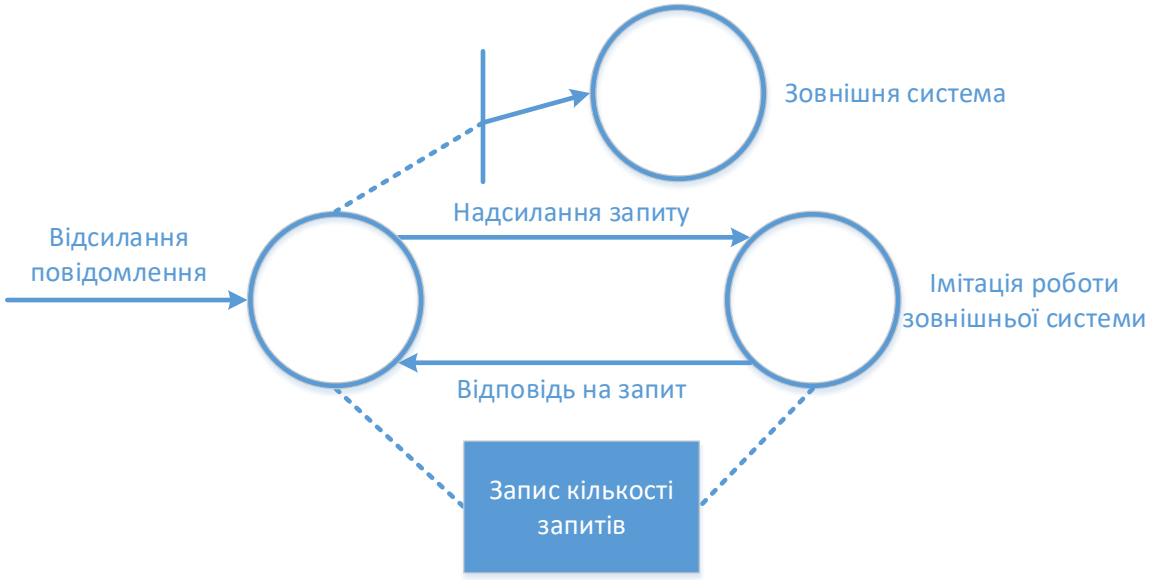

Рис.1.7. Логічна схема роботи програми з використанням імітації

Відповідно до рисунка 1.7 імітація перевіряє коректність параметрів, що передаються, порядок викликів і повертає результати деяким інтелектуальним чином. Імітація не є робочим рішенням, але може нагадувати таке, хоч і з деякими спрощеннями [4].

Виділяють також:

− макет: динамічно створюється бібліотекою макетів (інші типи мають на увазі роботу з кодом від розробника тесту). Розробник тесту не бачить реального коду, що реалізує інтерфейс, або базовий клас, але може налаштувати макет на надання даних, що повертаються, очікування виклику певних елементів і так далі. Залежно від налаштування макет може поводитися як фіктивний модуль, заглушка або шпигун;

− тестовий шпигун: подібний до заглушки, але крім видачі клієнту екземпляра, для виклику елементів, шпигун також записує, які елементи викликалися, так що модульні тести можуть показати, що елементи викликалися відповідно до очікувань. Однією з форм цього може бути служба електронної пошти, яка записує, скільки повідомлень було надіслано [18].

Отже, при автоматизованому тестуванні IT-продукту побудованого на мікросервісній архітектурі, доцільно використовувати тестові дублери, які виглядають і поводяться як мікросервіси, але насправді вони спрощені. Головна перевага використання тестових дублерів – можливість ізольовано та поетапно тестувати кожен мікросервіс, а згодом і взаємозв'язок мікросервісів між собою. Такий підхід зменшує складність і підвищує якість тестування IT продукту.

1.3. Методи автоматизованого тестування програм

Автоматизоване тестування програмного забезпечення – етап процесу тестування під час контролю якості під час розробки програмного забезпечення. При цьому використовуються програмні засоби для проходження тестового набору та перевірки результатів виконання. Все це в комплексі дозволяє скоротити час тестування та спростити його процес.

Головними факторами, що ініціюють впровадження автоматизованого тестування, можна назвати підвищення якості IT-продукту, що розробляється, і економія часу, що витрачається на безпосередньо процес тестування. Підвищення якості передбачає те, що при автоматизованому тестуванні можна охопити більший

набір тест-кейсів та/або тести, які неможливо виконати вручну, і той фактор, що при автоматизованому покритті можливо мінімізувати збитки від помилок людини в результатах тестування. У зв'язку з тим, що на виконання автоматизованих тестів йде набагато менше часу, ніж на аналогічне тестування вручну відбувається економія часу.

До переваг автоматизованого тестування можна віднести:

− скорочення часу на проходження тестового набору порівняно з ручним тестуванням;

− можливість проведення деяких тестових навантажень, які неможливо провести вручну;

− підвищення незалежності тестування від людини (виключається людський фактор помилки);

− можливість проводити тестування в будь-який час, у тому числі і поза робочим часом фахівця з тестування [3].

Крім переваг, необхідно розглянути і недоліки автоматизованого тестування, основні з них такі:

− автоматизоване тестування потребує значних трудовитрат та попередньої підготовки технологій та персоналу;

− необхідний досвідченіший і кваліфікований персонал;

− після проведення тестування слід провести ретельний аналіз результатів;

− будь-яка зміна в структурі коду програми тягне за собою модифікації у кодуванні автоматичних тестів;

− в результаті одного помилкового автотесту можуть вийти некоректні результати для проходження наступних тестів;

− саме написання автоматичного тесту необхідно зробити якісно, та без помилок, дотримуючись правильної логічної схеми реалізації;

− не всі функціональні вимоги в системі можна перевірити автоматизованими тестами за допомогою вибраного інструментарію.

Отже, доцільність застосування автоматизованого тестування який завжди виправдана і зрозуміла [1]. Наприклад, необхідність автоматизації очевидна, якщо

стоїть питання про проходження часто повторюваних ідентичних тестів, а також при тестуванні навантаження, тестуванні швидкодії компонентів системи або в інших ситуаціях, коли досить складно або неможливо забезпечити необхідні умови для проходження тесту вручну. Однак необхідність автоматизації часто піддається сумнівам, якщо виникає питання про автоматизацію тестування функціональних критеріїв.

Також важливу роль відіграє віртуалізація. Вона є ключовим фактором у питаннях, що пов'язані з керуванням над тестовим середовищем. Налаштування віртуальної машини надає додатковий простір як спеціалістам з розробки, і тестувальникам в організацію тестування досліджуваного докладання. Віртуалізація застосовується для зменшення витрат, на конфігурування операційної системи та програмних компонентів. Учасники часто використовують документ для збору різних вимог до середовища тестування з метою спланувати управління своїм існуючим середовищем або створити новим [22].

Ліза Кріспен та Джанет Григорі у своїй книзі описали матрицю, яка допомагає переконатися, що всі тести, які потрібні для забезпечення якості продукту, можна здійснити. Сектори тестування представлені рис. 1.8.

Відповідно до рисунка 1.8 квадрат Q1 - модульні тести. Мета використання – перевірити роботу певної функції чи методу. До цієї категорії тестування належить і тестування, засноване на методології розробки під контролем тестування (Test-Driven Design (TDD)) та всі види тестового покриття на основі властивостей (Property-Based Testing). За таких тестів не відбувається запуск сервісу, тестування відбувається без використання зовнішніх систем або підключень до мережі.

Отже таких тестів запускається дуже багато, але їх проходження відбувається досить швидко, якщо логічно тести правильно сконструйовані. У сучасних умовах прогнозується проходження багатьох тисяч таких тестів за час менший за 1 хвилину. Головне завдання при модульному тестуванні – отримання миттєвих відповідей за запити, які говорять про коректне функціонування написаного коду конкретної функції чи методу.

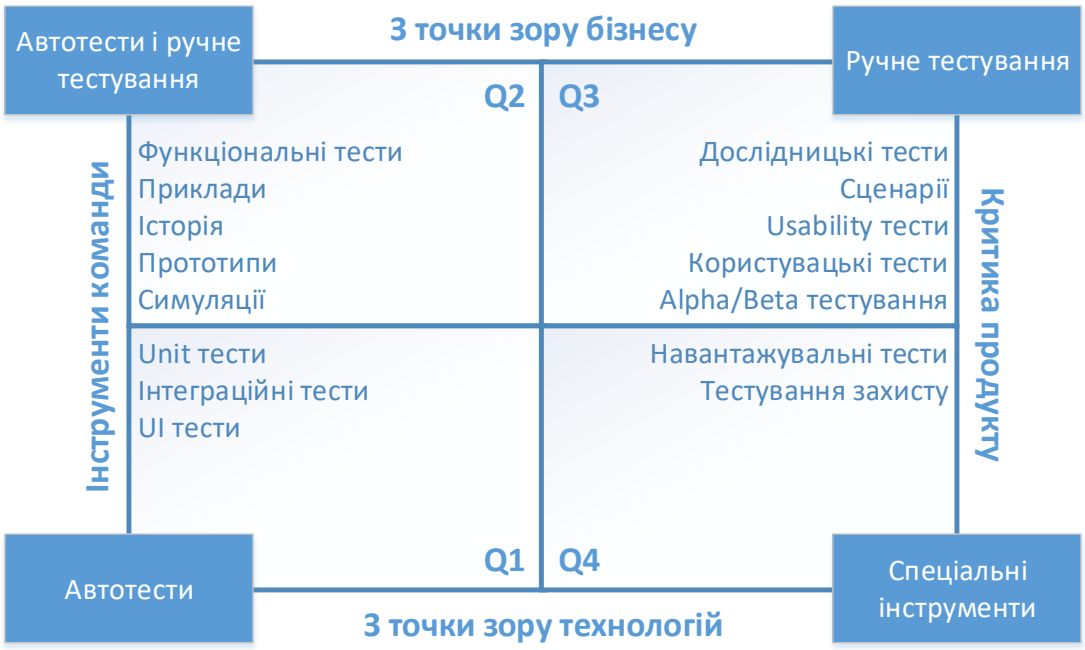

Рис. 1.8. Сектори тестування за Лізою Кріспен та Джанет Грегорі

Ідея розробки інтеграційних тестів у тому, щоб без участі користувача провести безпосереднє тестування сервісу. У разі написання програми за принципом монолітної системи доцільно тестувати колекції класів, які забезпечують сервіс інтерфейсу користувача. Якщо ж продукт, що розробляється, складається з кількох мікросервісних модулів, тестування сервісу використовується з метою оцінки можливостей конкретного ізольованого сервісу. Головний чинник, що впливає такий поділ – бажання ізолювати тест від решти функціоналу з більш якісного тестування.

Квадрат Q2. Функціональні тести допомагають розробникам у тестування системи, але більш абстрактному рівні. Вони формуються на основі функціональних вимог замовника та тестують роботу певного функціоналу. Таке тестування необхідно провести для всієї системи, що розробляється. Запускати та контролювати таке тестування найзручніше через графічний інтерфейс користувача у браузері. Але важливим є і перевірка інших функцій взаємодії з користувачем, наприклад викладання файлу. Результатом функціонального тестування є упевненість у тому, що протестований продукт працює у виробничому режимі.

Квадрат Q3. Дослідницькі тести перевіряють, чи здатна система коректно працювати з погляду користувача. Таке тестування можна виконувати лише вручну, оскільки необхідно оцінити зручність та правильність створеного функціоналу. Найчастіше таке тестування проводиться разом із представниками замовника під час так званої UAT фази (User Acceptance Testing). Під час такого тестування спільно із замовником можливо не тільки виявити недоліки поточного рішення, а й запланувати розробку нового функціоналу у програмі, що створюється.

Квадрат Q4. Навантаження тестування вимагає застосування спеціальних програмних інструментів та передбачає перевірку відповідності створеного IT-продукту заданим нефункціональним вимогам. До таких вимог відносять властивості, що визначають як програма повинна демонструвати при роботі, або якісь обмеження, які вона повинна дотримуватись, що не впливають на поведінку системи. Наприклад, максимальна продуктивність, стійкість, зручність використання, розширюваність, надійність, умови експлуатації, підтримуваність.

Дані вимоги дуже важливі для бізнесу, нарівні з функціональними – наприклад, повільне відкриття стартової сторінки інтернет-магазину загрожує втратою клієнтів. Інший варіант - розголошення або втрата особистих даних користувачів спричиняє судові витрати та репутаційні втрати. [8]

Отже, тестування проводиться з безлічі причин: знайти помилки в коді програми, переконатися, що створений програмний продукт надійний, і що він вирішує поставлені перед ним завдання. Знання всіх функціональних та нефункціональних вимог, планованих інтеграцій, безпеки, полегшує розробку системи та дозволяє краще вибрати як архітектуру, так і використовуваний інструментарій для тестування.

Проведений огляд теоретичних основ та методів автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі, можна зробити висновок про те, що для створення якісного, конкурентоспроможного програмного продукту необхідний комплексний підхід до вибору необхідних інформаційних ресурсів та технологій. Вибравши оптимальну

модель використання хмарного простору, можна отримати зручний та швидкий доступ до ресурсів без додаткових грошових та часових витрат.

Мікросервісна архітектура IT-продукту дозволяє швидше розробляти та тестувати кожен мікросервіс, який націлений на виконання лише одного завдання.

При такій організації розробки програмних продуктів доцільно використовувати автоматизоване тестування, де відповіді на запити досліджуваного мікросервісу посилатимуть тестові дублери, що імітують зв'язок із підсистемами, що взаємодіють між собою. Розглянувши категорії тестових дублерів та їх логічну схему роботи, стає можливим визначитися з оптимальним видом тестового дублера і налаштувати його для взаємодії з розроблюваним мікросервісом.

Проте слід зазначити, що впровадження автоматизованого тестування потребує детального аналізу, раціональної обґрунтованості, кваліфікованих співробітників та часових та економічних витрат.

РОЗДІЛ 2
АНАЛІЗ МЕТОДІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ІТ - ПРОДУКТУ, ПОБУДОВАНОГО НА МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

2.1. Критерії ефективності тестування програмного забезпечення

Класифікація основних критеріїв формування тестових наборів під час тестування програмного забезпечення представлена рис. 2.1.

Рис.2.1. Класифікація критеріїв тестування

Відповідно до рис. 2.1 виділяють наступні критерії:

1. Стохастичні критерії: застосовуються для тестування комплексних програмних модулів – у ситуаціях, коли набір детермінованих тестів є досить великим. Спочатку складаються програми - імітатори випадкових потоків вхідної інформації. Далі незалежним способом генеруються значення {y} для відповідних вхідних значень {x} й у результаті формується тестовий набір {x,y}.

2. Критерії тестування потоків керування: передбачають перевірку системи за принципом "білої скриньки". Цей підхід вимагає розуміння написаного коду програми відповідно до специфікації як потокового графа керування. Цей вид критеріїв виправдано під час планування модульного та інтеграційного тестування (Unit testing, Integration testing).

3. Критерії тестування потоків даних припускають формування наборів тест-кейсів, які в сукупності забезпечують одноразову перевірку кожного екземпляра класу вхідних даних. Відбувається зіставлення класів даних з режимами функціонування блоку або мікросервісу програми, що тестується. Такий підхід

суттєво зменшує варіації переборів, які слід враховувати під час розробки тестових наборів. Крім обмежень на величини вхідних даних, необхідно враховувати обмеження, пов'язані з даними у різних комбінаціях. Також важливим чинником є перевірка функціонування системи у разі збоїв у значеннях чи структурах вхідних даних. Контроль аналогічних помилок - процес дуже трудомісткий і витратний за часом, що тягне за собою обмеження у використанні даного критерію.

4. Функціональні критерії є найважливішими у процесі тестування. Вони забезпечують контроль відповідності вимог замовника до програмного продукту. Працюють за моделлю "чорної скриньки".

До найбільш застосовуваних видів функціонального тестування слід зарахувати тестування функцій та тестування специфікацій.

Тестування функцій є найпопулярнішим критерієм, яким перевіряється кожна функція (спосіб), реалізована програмним модулем (класом). Тестування функцій поєднує в собі особливості структурних і функціональних критеріїв і базується на принципі "напівпрозорого ящика", при якому явно простежуються не тільки входи та виходи модуля, що тестується, але також склад і структура використовуваних методів (функцій, процедур) і класів.

У разі тестування специфікації важливо спланувати набір тестів, який би перевірив виконання кожного пункту вимог замовника мінімум раз. Оскільки специфікація найчастіше включає тисячі вимог, то тестування специфікації має перевірити їх усі, торкаючись всіх вимог до кожного методу. Крім того, перевірка має відбутися і щодо всіх вимог до класів та системи загалом.

5. Мутаційні критерії реалізують підхід, який дозволяє на основі незначних помилок зробити висновок про сукупну кількість помилок у системі, що розробляється. Мутаціями вважаються невеликі несправності, а мутанти - це програми, що відрізняються один від одного мутаціями. Принцип мутаційного тестування заснований на тому, що розроблювану програму P вносять мутації, тобто штучно створюють програми-мутанти P1, P2 ... Потім програма P та її мутанти тестуються на тому самому наборі тестів (X,Y).

У разі успішного проходження набору тестів формується судження про коректну роботу програми P. Крім того, фіксуються всі внесені в програми-мутанти помилки, робиться висновок про те, що набір тестів (X,Y) відповідає мутаційному критерію, а система, що тестується, вважається коректною [1].

Отже, розглянувши основні критерії формування тестового набору можна зробити висновок про те, що основна складність у тестуванні - визначити і сформувати достатню кількість тестових наборів, які дозволять всебічно перевірити систему, що розробляється, але при цьому кількість тестів не буде надмірною.

2.2. Методи автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі

Основною проблемою тестування є визначення того, яка кількість тестів буде достатньою для висновку про коректну роботу системи.

З метою вирішення цього завдання виділяють такі методики:

- еквівалентний поділ (Equivalence Partitioning),
- метод аналізу граничних значень (Boundary Value Analysis),
- спосіб складання діаграм причинно-наслідкових зв'язків (Cause/Effect) [1].

Еквівалентний поділ передбачає поділ вхідних даних на класи еквівалентності. Ідея в тому, щоб виключити набір вхідних даних, які змушують систему поводитися однаково і давати аналогічний результат під час тестування програми. Ідея - зменшення кількості надлишкових тестових сценаріїв, позбавляючись тих, які генерують один і той же результат і не завжди виявляють дефекти у функціональності програми [28].

На рис. 2.2 представлений приклад виділення класів еквівалентності (показані області коректних та некоректних значень, а кружками - самі класи еквівалентності).

Метод аналізу граничних значень передбачає підхід, у якому слід аналізувати як граничні значення, а й дані поблизу (вище і нижче) виділених границь. Як правило, методики тестування класів еквівалентності та аналізу граничних значень слід поєднувати. Як недолік такого методу слід зазначити те, що при такому

тестуванні не покриваються всі можливі комбінації безлічі вхідних даних і виникає велика ймовірність пропустити критичне значення, що викликає відмову системи.

Рис. 2.2. Виділення класів еквівалентності

Складання функціональних діаграм допомагає складати результативні набори тест-кейсів, які виявляють велику кількість помилок. Генерація тестових наборів за цим методом відбувається в наступній послідовності:

- вивчення специфікації (виділяються чинники (класи еквівалентності) і наслідки (вихідне обмеження чи зміна програми), вибудовується їх взаємозв'язок);
- відповідно до 1-го етапу складається булевий граф (демонструються події, їх наслідки та сукупність зв'язків між ними);
- на основі графа складається таблиця рішень (відбувається перетворення булевого графа на таблицю рішень шляхом методичного відстеження станів умов на схемі);
- формування тестового комплекту по шпальтах таблиці рішень [16].

Приклад булевого графа тестування представлений рис. 2.3.

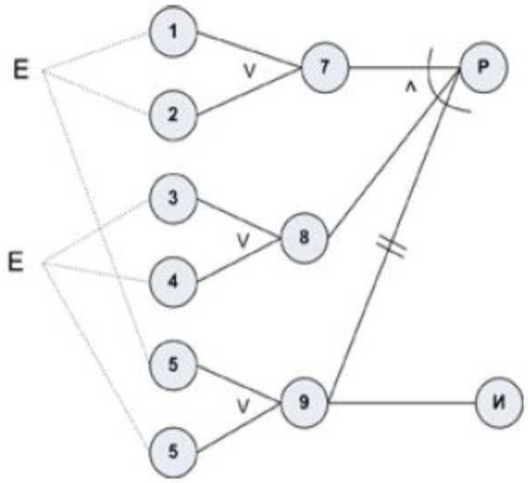

Рис. 2.3. Приклад булевого графа тестування

Розглянутий метод забезпечує високу результативність тестових наборів, що формуються, але розглянутий процес займає невиправдано багато часу і при ручному виконанні є надзвичайно трудомістким. Етап перетворення булевого графа в таблицю рішень та складання тестового набору по складеній таблиці можна частково автоматизувати.

Для визначення необхідної методики тестування необхідно скласти специфікацію процедури тестування. Весь процес проектування та реалізації автоматизованого тестування програмного продукту представлений рис. 2.4.

Рис.2.4. Проектування та реалізація процесу тестування

Відповідно до рисунка 2.4 початковим етапом є визначення переліку особливостей та переліку умов. Потім досліджується область тестового покриття та складається ранжування тестових випадків. Далі складається специфікація процедури тестування, що включає опис процедури тестування та остаточний набір тестів. Доцільно розбивати тестові набори на дрібніші, які можуть бути повторно використані у різних тестових випадках.

Дуже важливим аспектом у тестуванні є можливість багаторазового використання тестових наборів. Автоматизоване тестування використовується для

повторного застосування створених тестів. Життєвий цикл тесту представлений рис. 2.5.

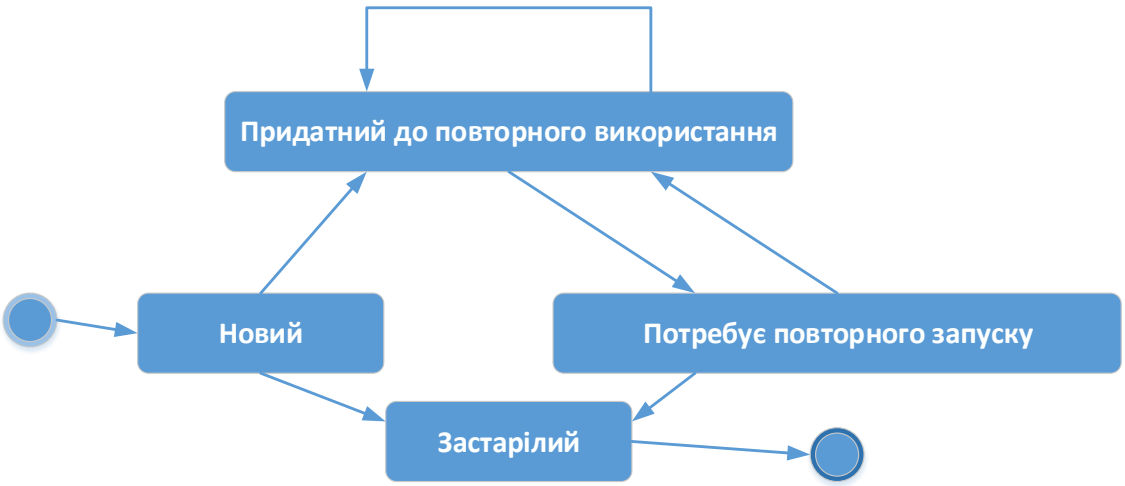

Рис. 2.5. Життєвий цикл тесту

Отже, відповідно до рис. 2.5 життєвий цикл будь-якого тесту складається з послідовних етапів: новий тест, який згодом стає або застарілим, або придатним для повторного використання. Далі можливі варіанти: тест може вимагати повторного запуску та залишатися придатним для повторного використання, а може стати застарілим, що говорить про завершення життєвого циклу.

До необхідності модифікації створеного тестового набору наводять три ситуації:

- складання нових актуальних тест-кейсів,
- проходження тест-кейсів,
- модифікація коду.

Старий тест перестає бути актуальним і відбувається створення нового тесту, якщо змінюється вхідна інформація чи супутні умови. Зміна вхідних даних спричиняє зміну траєкторії і, як наслідок, результуючих даних. Отже, слід розглянути різні варіанти використання тестів повторного.

На першому рівні тест використовувати повторно не передбачається.

На другому рівні допускається повторне використання лише вхідних даних тесту. Розглядаються випадки, коли частина програми, що тестується включається в роботу раніше новостворених команд, тоді вхідні дані можуть бути використані

повторно для покриття цих елементів. Але після змін у системі та/або специфікації на розробку алгоритм та результуючі дані можуть суттєво відрізнятись від результатів попередніх запусків.

На третьому рівні можна повторно використовувати вхідні і вихідні дані тесту. Функціональне тестування відноситься до цього рівня. Якщо у системі відбувається зміна коду зі збереженням основного функціоналу, стає можливим багаторазове застосування існуючих функціональних тестів для перевірки коректності системи [27]. В результаті повторного проходження тестового набору підсумкові результати мають бути ідентичними результатам попередніх запусків тестів.

Четвертий рівень є найвищим рівнем повторного використання тесту Він допускає повторне використання вхідних даних, вихідних даних та траєкторії тесту. Бувають випадки, коли на траєкторії тесту не змінився жоден оператор, тоді недоцільно запускати такі тести кілька разів, так як і результати, і траєкторія залишаться колишніми [11]. Ідея доцільності впровадження тестування полягає в тому, що кількість випадків, що тестуються, буде збільшуватися з кожним завершеним модулем проекту.

Отже, проаналізувавши методи автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі, можна сформулювати висновок, що вибір конкретного методу залежить від специфікації програмного продукту, рівня автоматизації тестування та критеріїв ефективності роботи готової системи. Проблему планування тестування можна як проблему практичного вирішення мінімізації витрат під час написання програмний продуктів. Моніторинг записів раніше виконаних тестових випадків може вплинути на результати нових тестових випадків, особливо на рівні інтеграційного тестування, де тестові приклади більш взаємозалежні.

2.3. Сценарії автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі

Мікросервісна архітектура стала найкращою альтернативою для монолітних, складних та негнучких додатків. Монолітна програма має слабку масштабованість, а оновлення на рівні процедур потребує повного оновлення системи.

При тестуванні програмного продукту, побудованого на мікросервісної архітектурі, вся система поділяється більш дрібні, модульні, спільно працюючі компоненти. Їх набагато простіше створювати, оновлювати та тестувати, ніж додаток загалом. Крім того, ця архітектура забезпечує більшу масштабованість, що дуже ефективно для великої кількості розгортань на різних платформах.

Особливостями автоматизованого тестування програмний продуктів, побудованих на мікросервісній архітектурі є:

- необхідність автоматизованої інфраструктури, забезпечує управління життєвим циклом, найменування, адресацію та масштабування мікросервісів залежно від поточного завантаження;
- необхідність безперервної інтеграції розроблених та/або модифікованих мікросервісів у існуючу систему вимагає всебічного тестування як окремих мікросервісів, так і їхнього спільного функціонування в комплексі з іншими мікросервісами.

Весь процес розробки та тестування мікросервісів представлений рис. 2.6.

Відповідно до рисунка 2.6 при тестуванні мікросервісів велике значення слід приділити компонентному тестуванню, тестуванню контейнера з мікросервісом та інтеграційному тестуванню.

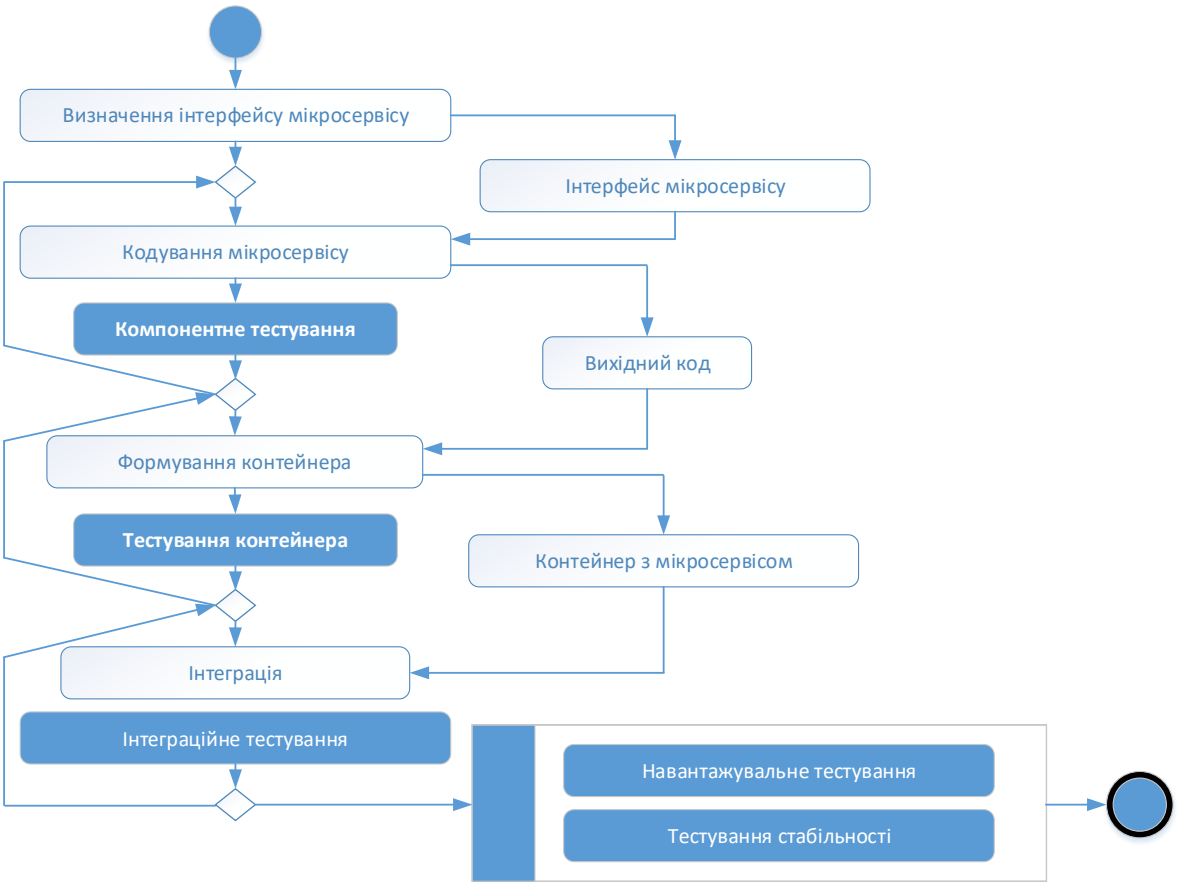

Рис.2.6. Процес розробки і тестування мікросервісів

Компонентне тестування передбачає тестування кожного мікросервісу окремо, ізольовано. Логічна схема компонентного тестування мікросервісних систем представлена рис. 2.7.

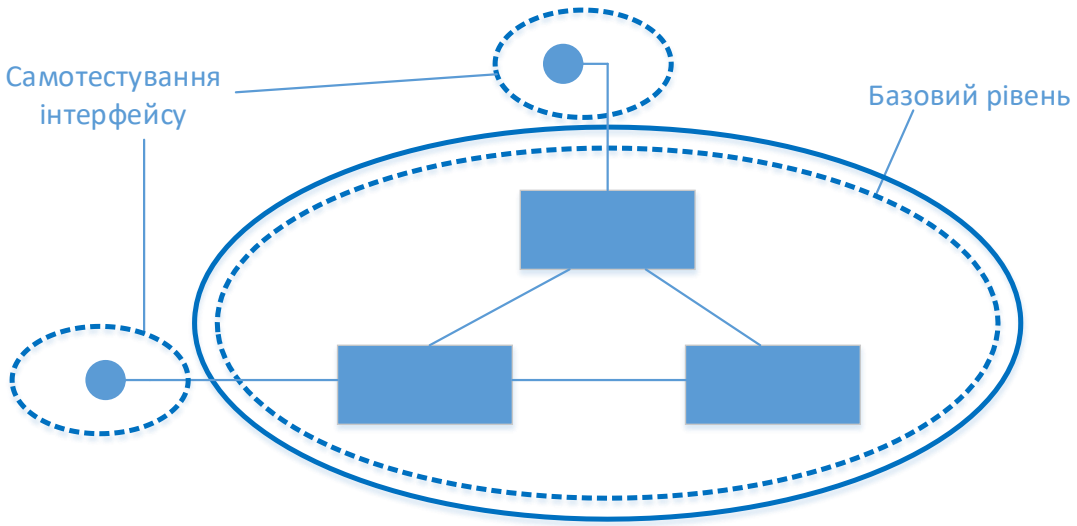

Рис.2.7. Компонентне тестування мікросервісних систем

На рис. 2.7 наочно зображено, що компонентне тестування передбачає самотестування інтерфейсу кожного мікросервісу всередині базового (окремого) рівня системи.

У компонентному тестуванні мікросервісів слід розглянути:

- функціональне компонентне тестування спрямоване на діагностику роботи кожного мікросервісу залежно від функціональних вимог замовника;
- навантажувальне компонентне тестування полягає в оцінці працездатності та максимальної стійкості кожного мікросервісу при певному навантаженні;
- компонентне тестування безпеки включає тестування безпеки та ізольованості окремого мікросервісу.

Тестування контейнера із мікросервісом передбачає наступний сценарій. Кожен мікросервіс викликається незалежно один від одного і перевіряються їх відповіді. Будь-які залежності повинні допомогти мікросервісу функціонувати, але не має бути взаємодії з іншими сервісами. Це дозволить уникнути будь-якої непередбачуваної поведінки, яка може бути спричинена впливом іншого сервісу. Під час тестування перевіряється відповідь чи інший результат, який залежить від вхідних даних. Кожен споживач повинен отримувати той самий результат від мікросервісу, навіть якщо його внутрішня реалізація змінюється. Кожен мікросервіс повинен бути здатний до гнучкої зміни функціоналу, але раніше реалізований функціонал не повинен змінюватися і тягнути за собою зміну мікросервісу, що викликає.

Кожен мікросервіс має кілька екземплярів часу виконання. При тестуванні контейнера з мікросервісом кожен екземпляр має бути налаштований, розгорнутий, масштабований та відстежений [28].

Далі необхідно провести інтеграційне тестування мікросервісних систем. Інтеграційне тестування – це етап тестування системи, що складається з двох і більше модулів та спрямований на виявлення помилок у реалізації та інтерпретації інтерфейсної взаємодії між модулями. Інтеграційне тестування поєднує у собі такі види:

- функціональне інтеграційне тестування забезпечує тестування взаємодії окремих мікросервісів між собою:

а) протоколи та формати обміну повідомленнями,

б) взаємні блокування та спільне використання спільних ресурсів;

- навантажувальне інтеграційне мікросервісне тестування оцінює стан та працездатність системи при автоматичному розгортанні або масштабуванні мікросервісів;

- інтеграційне тестування безпеки мікросервісів передбачає тестування безпеки взаємодії між мікросервісами, і навіть перевірку можливість перехоплення повідомлень зловмисниками ззовні чи всередині системи.

У процесі інтеграційного тестування відбувається збирання мікросервісів (модулів). Розрізняють такі сценарії [18]:

- монолітний, що передбачає одночасну інтеграцію всіх мікросервісів у єдину систему. Для імітації роботи відсутніх, але необхідних для тестування модулів, необхідно використовувати тестові дублери.

- інкрементальний, коли передбачається поетапне тестування шляхом поступової інтеграції двох або логічно пов'язаних модулів. Поступово приєднуються інші модулі до моменту повної інтеграції. Далі відбувається комплексне тестування усієї системи.

Виділяють дві стратегії додавання модулів: «знизу догори» і «згори донизу».

"Знизу вгору" (висхідна інтеграція): кожен модуль на нижчих рівнях тестується з вищими модулями, доки не перевіриться робота всіх модулів. Часто вдаються до використання тестових дублерів (заглушок) з отриманням у результаті цілої програми відповідно до рисунка 2.8.

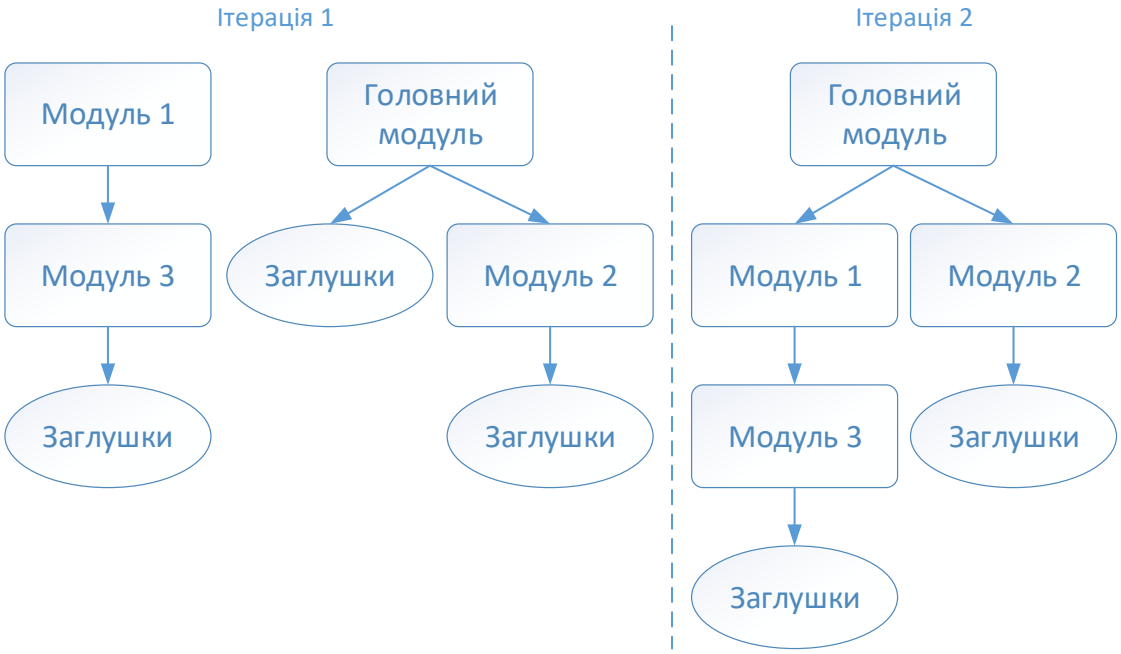

Рис. 2.8. Схема висхідної інтеграції

"Зверху вниз" (низхідна інтеграція) передбачає послідовне приєднання модулів до головного модуля, з використанням тестових дублерів (заглушок) відповідно до рисунка 2.9.

Часто найзручнішим способом стає використання змішаної інтеграції (Mixed Integration), коли спочатку збірка йде по висхідній інтеграції (модулі групуються в блоки), а далі по низхідній інтеграції (блоки приєднуються до модуля, що управляє). Змішана інтеграція дуже зручна і дозволяє уникнути недоліків традиційних підходів [8].

Виділяють теорію «Великого вибуху» («Big Bang» Integration), яка передбачає одномоментну інтеграцію всіх модулів до однієї системи та реалізацію інтеграційного тестування. Цей підхід суттєво скорочує тимчасовий фактор тестування, проте може призвести до миттєвого виникнення багатьох критичних помилок.

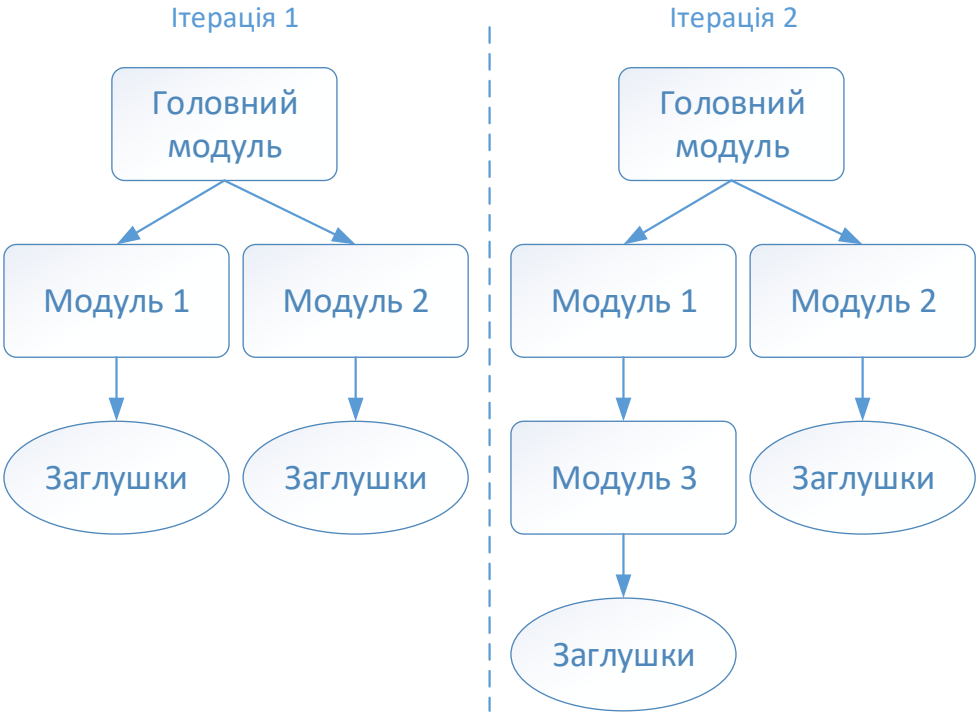

Рис. 2.9. Схема низхідної інтеграції

Таким чином, при виборі будь-якого сценарію тестування мікросервісного програмного продукту необхідне всестороннє тестування як окремих мікросервісів, так і їхнього спільного функціонування в комплексі з іншими мікросервісами. Ефективне тестування мікросервісного програмного продукту неможливе без використання тестових дублерів, які спеціально розробляються для забезпечення необхідного функціоналу недоступних модулів.

2.4. Методологія розробки програмного забезпечення з інтегрованим процесом тестування

В даний момент в IT індустрії існує безліч методологій розробки програмного забезпечення, але найпопулярнішими вважаються - каскадна модель (Waterfall Model) і різні варіації гнучкої розробки (Agile).

Традиційною та найконсервативнішою моделлю розробки та тестування ПЗ є каскадна модель (Waterfall Model). Суть цієї моделі полягає у дотриманні заздалегідь спланованих етапів розробки та тестування. Сувора ієрархічна послідовність означає, що неможливо приступити до наступної стадії без успішного

завершення попередньої. Зараз таку модель застосовують рідко і використовують лише в тому випадку, якщо всі функціональні та нефункціональні вимоги та критерії заздалегідь визначені та не підлягають зміні.

Етапи розробки програмного забезпечення за каскадною моделлю представлені рис. 2.10.

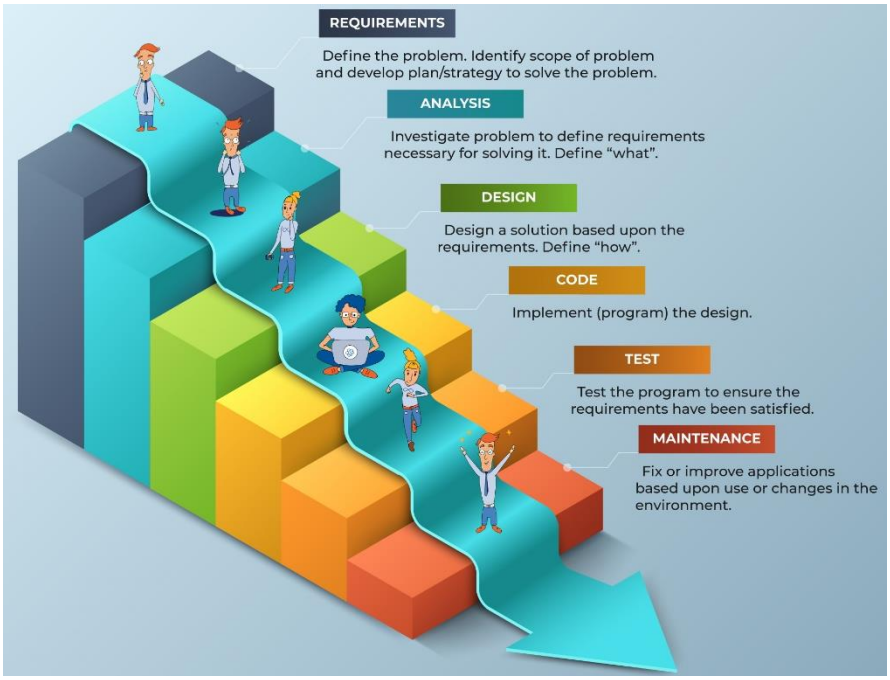

Рис. 2.10. Етапи розробки програмного забезпечення за каскадною моделлю (Waterfall Model)

Працюючи за допомогою каскадної моделі розробки ПЗ юніт-тести та інтеграційні тести не обов'язкові, оскільки процес тестування починається вже після завершеного процесу розробки. Але такі види тестування дають змогу виявити недоліки програмного продукту, які необхідно виправити до дати завершення проекту. Таким чином, виходить неефективний процес повернення процесу розробки, потім знову стадія тестування і так кілька разів.

При використанні методології гнучкої розробки (Agile) процес розробки програмного забезпечення стає максимально гнучким, розробка йде малими ітераціями [10].

Мета застосування методології гнучкої розробки - зробити нову функціональність якнайшвидше доступною споживачеві, дозволяючи швидше

замовнику почати отримувати прибуток від продукту, що розробляється.

Процес методології гнучкої розробки представлений рис. 2.11.

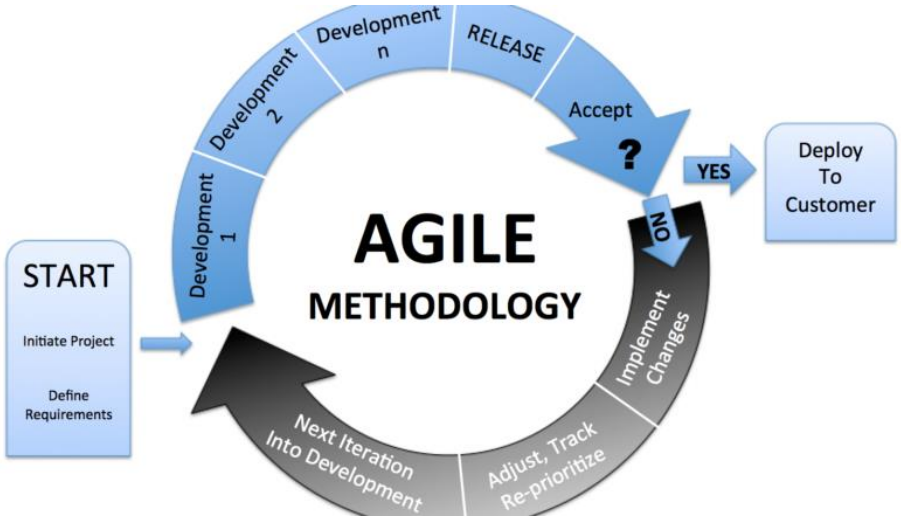

Рис. 2.11. Процес розробки програмного забезпечення з методології Agile

З метою підвищення якості готового програмного продукту та прискорення процесу виконання проектів було створено такі моделі розробки програмного забезпечення:

- розробка через тестування (Test Driven Development, TDD);
- розробка на основі поведінки (Behavior Driven Development, BDD);
- розробка через приймальне тестування (Acceptance Test-Driven Development, ATDD).

Розробка через тестування (Test Driven Development, TDD) передбачає створення розробником автоматизованих перевірочних тестів кожного модуля (функції) перед написанням самої програми. Тести створюються на основі вимог, зазначених у специфікації і включають перевірку умов, які можуть або виконуватися, або ні. Коли вони виконуються, вважається, що тест пройдено. Проходження тесту підтверджує поведінку, яку очікує програміст.

Послідовність дій під час розробки через тестування така:

- написати тест для функціональності, що розробляється;
- написати код, який передбачається тестувати;
- провести рефакторинг нового та старого коду.

Цикл розробки відповідно до TDD представлений рис. 2.12.

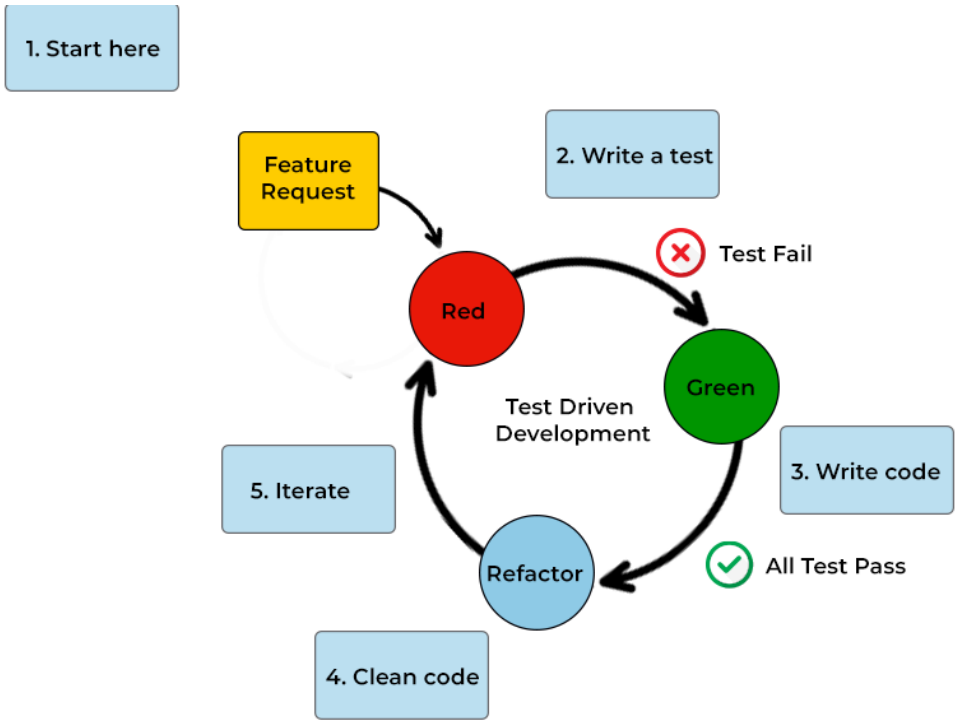

Рис. 2.12. Процес розробки програмного забезпечення методології TDD

Відповідно до рисунка 2.12 перша фаза (RED) полягає в тому, що програміст пише тест і метод-заглушку на класі, що тестується, необхідні тільки для того, щоб тест запустився. Функція-заглушка (тестовий дублер) - функція, що не виконує жодної певної дії, повертає або порожній результат, або вхідні дані в незмінному вигляді. Заглушку слід розробити таку, щоб тест не пройшов - тоді робиться висновок, що тест коректно реагує на помилковий запит.

На наступному етапі (GREEN) тестовий дублер редагується так, щоб тест почав працювати правильно (при цьому пишеться мінімальна кількість коду).

Третя фаза (REFACTORING) має на увазі процес рефакторингу - модифікація внутрішньої структури програмного продукту, яка не тягне за собою зміни у її зовнішній поведінці. Мета такої модифікації - удосконалити і спростити логічний порядок коду продукту, що розробляється. Цей етап є обов'язковим при розробці програмного забезпечення. Фахівці з розробки спочатку пишуть новий функціонал та тестові набори для його перевірки, а потім роблять рефакторинг, удосконалюючи

логічність виконання операцій. Автоматичне модульне тестування дозволяє переконатися, що рефакторинг не порушив функціональність.

Розробка з урахуванням поведінки (Behavior Driven Development, BDD) - техніка розробки, коли він аналізується не результат виконання будь-якого модуля, а та робота, що він виконує. BDD передбачає опис тестувальником або бізнес-аналітиком сценаріїв користувача природною мовою — мовою бізнесу. Написання тестів у вигляді цілих пропозицій мовою бізнес-функцій полегшує як замовникам та аналітикам, так і розробникам розуміння цілей створення програмного продукту та складання технічної документації [7].

Цикл розробки ПЗ відповідно до BDD представлений рис. 2.13.

Рис. 2.13. Цикл розробки програмного забезпечення з методології BDD

Виходячи з рисунку 2.13 кожна специфікація діє як вхідна точка в цикл розробки та описує, як повинна поводитися система з точки зору користувача (у покроковому вигляді). Основна увага приділяється не тестам модулів або об'єктів, а цілям користувачів та покроковим операціям, які вони роблять для досягнення цих цілей. У цьому вигляді тестування тестові дублери не емулюють роботу програмних модулів мікросервіса. Натомість емулюються більші об'єкти - цілі системи підсумкового рішення, а саме:

- підсистеми, з якими програма інтегруватиметься;
- мікросервіси, які ще готові;
- будь-які елементи системи для перевірки спеціальних сценаріїв (негативні, тести навантаження).

Розробка через приймальні тести (Acceptance Test-Driven Development, ATDD). Цикл розробки відповідно до ATDD представлений рис. 2.14.

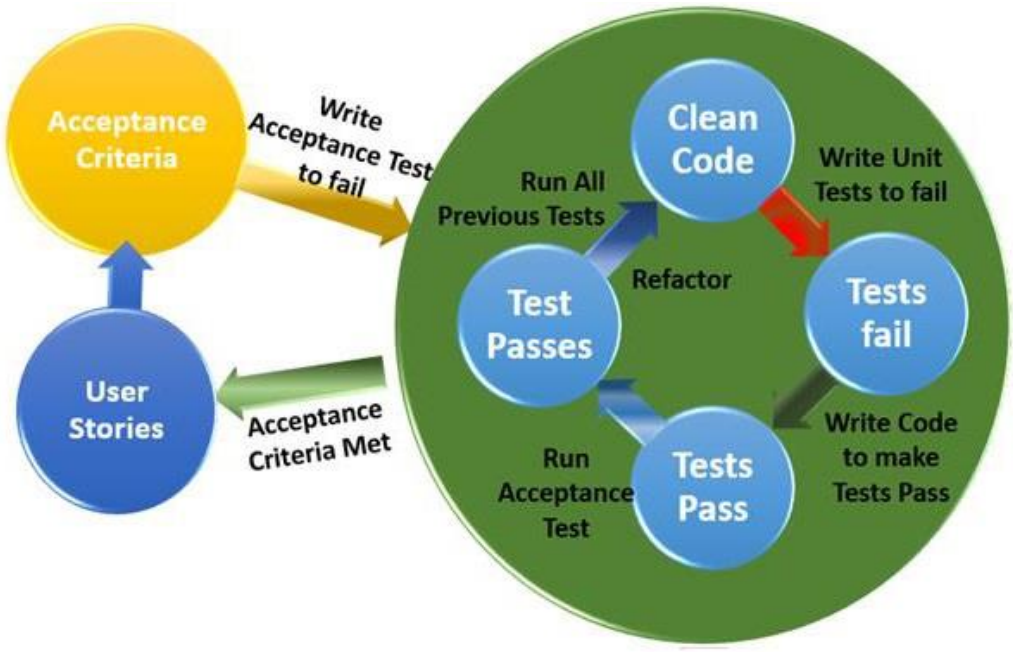

Рис. 2.14. Цикл розробки програмного забезпечення ATDD

Ідея ATDD, зображеної рис. 2.14, у тому, що як щось розробляти, необхідно визначити критерій виконаного запиту та критерій того, що отримано коректну відповідь на запит. Дані критерії дозволяють на ранніх стадіях зрозуміти, що саме при тестуванні готового програмного забезпечення слід вважати коректним результатом.

Створюються сценарії з урахуванням виявлених критеріїв, ці сценарії мають відбивати поведінка проектованої системи. Коли всі кроки сценарію виконані і кінцевий результат збігається з очікуваним, то завдання вважається вирішеним. Комплект створених сценаріїв є приймальним тестуванням.

Приймальні тести дозволяють перевірити розроблюваний продукт шляхом

використання інтерфейсу користувача, але вони не відображають оптимальне внутрішній пристрій і технічні характеристики розробки. Використовуючи цю методологію, всі учасники розробки (замовники, фахівці з розробки та тестування) разом складають технічну документацію на основі критичних тестових вимог, що дає змогу в найкоротший термін розробляти якісний IT- продукт.

Розробка по ATDD має на увазі, що всі учасники, залучені до процесу створення програмного продукту, точно знають, які вимоги до проекту і що необхідно зробити. Приймальні випробування складені відповідно до вимог клієнта та можуть також використовуватись при документуванні вимог. Команда розробників може використовувати приймальні тести, щоб переконатися, що програма працює коректно і що система, над якою вони працюють, справді вирішує завдання клієнтів. Застосування тестових дублерів у цій методології повністю ідентичне BDD методології - також емулюються великі підсистеми підсумкового рішення перевірки спеціальних і негативних тестів.

Кожна з розглянутих методологій надає унікальний набір можливостей командам розробки та ставить їх завдання, але вони націлені на «тестування як частину проектування».

Всі перераховані вище методології розробки програмного забезпечення так чи інакше використовують тестові дублери (testing doubles), але вони особливо важливі для BDD і ATDD методик. Однією з основних переваг тестових дублерів є можливість переконатися, що основні сценарії використання системи працюють та відповідають усім функціональним та нефункціональним вимогам. Даний тестовий набір корисний не тільки при тестуванні продукту, що розробляється, але також і після встановлення на обладнання замовника, з метою переконатися, що на його конфігурації обладнання все працює так, як і було заплановано. У процесі приймального тестування користувачами (User Acceptance Testing) тестовий дублер, що входить до складу програмного продукту, що видається, є ефективною і часто необхідною опцією, що дозволяє без затримок і додаткових узгоджень перевірити весь функціонал програми на працездатність за допомогою вже написаних і налагоджених BDD і ATDD тестів.

Проаналізувавши методи проведення автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі, можна зробити висновок, що вибір критеріїв та підмножини згенерованих тестових випадків або їх ранжування для виконання може призвести до більш ефективного використання виділених ресурсів тестування. Сгенерований тестовий набір для тестування програмного продукту може мати різні показники якості і тому немає однакових сценаріїв для виконання.

Визначення властивостей тестових випадків та вимірювання їх значення для виконання можна вважати основним ключем до вирішення задачі оптимізації процесу тестування. Визначення критичних критеріїв та бажаних цілей залежить від кількох факторів, таких як розмір тестів, їх складність, різноманітність, а також процедура тестування. Кількість прогнозованих повторних запусків тест-кейсів є одним із ключових факторів при обґрунтуванні використання автоматизованого тестування.

В результаті вивчення нових методологій розробки програмних продуктів можна дійти до висновку у тому, що методологія BDD дозволяє зробити процес тестування зручнішим, дешевим і корисним, підвищуючи якість готового IT - рішення.

Використання тестових дублерів як частини готового рішення, побудованого на мікросервісній архітектурі, дозволяє ізольовано розробляти та тестувати сервіси, збудовані навколо бізнес-функцій. Вони можуть бути розгорнуті незалежно через повністю автоматизований механізм розгортання.

Ці мікросервіси можуть бути розроблені на базі різних мов програмування та можуть використовувати різні технології для зберігання даних. Необхідність безперервного інтеграційного тестування розроблених або змінених мікросервісів потребує комплексного підходу до автоматизації тестування.

Процес тестування повинен бути адаптований до зміни адресації, а також масштабування мікросервісів в залежності від поточного завантаження.

РОЗДІЛ 3
АПРОБАЦІЯ МЕТОДІВ ПРОВЕДЕННЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ІТ - ПРОДУКТУ ПОБУДОВАНОГО НА МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

3.1. Інформаційні моделі автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі

На етапі проектування програмного продукту доцільно створювати інформаційні моделі, що відображають загальні принципи та алгоритми роботи системи, що створюється. Однією з таких інформаційних моделей є діаграма варіантів використання, на якій представляється загальна концептуальна схема системи, що розробляється. Виділяють такі цілі складання діаграми варіантів використання:

- на ранніх стадіях проектування визначити межі створюваної системи з урахуванням її призначення;
- встановити функціональні вимоги;
- спроектувати стартову концептуальну модель системи з метою її вдосконалення на основі фізичних та логічних зв'язків;
- створити технічну документацію для більш предметного спілкування розробників із замовниками.

Розроблена діаграма показує, які варіанти використання програмного продукту, що розробляється, повинні бути доступні. Крім того, визначається взаємодія із системою ззовні. Особами, які здійснюють таку взаємодію, можуть вважатися будь-які об'єкти, суб'єкти чи системи. Вони видаються як зовнішні сутності або актори, що ініціюють певну поведінку створюваної системи.

Діаграма варіантів використання тестування мікросервісу з використанням тестового дублера представлена на рис. 3.1. Варіанти використання мікросервісу: формування запиту на читання даних, формування запиту модифікацію даних, обробка відповідей. Варіанти використання тестового дублера:

- обробка запиту,
- формування відповіді запити.

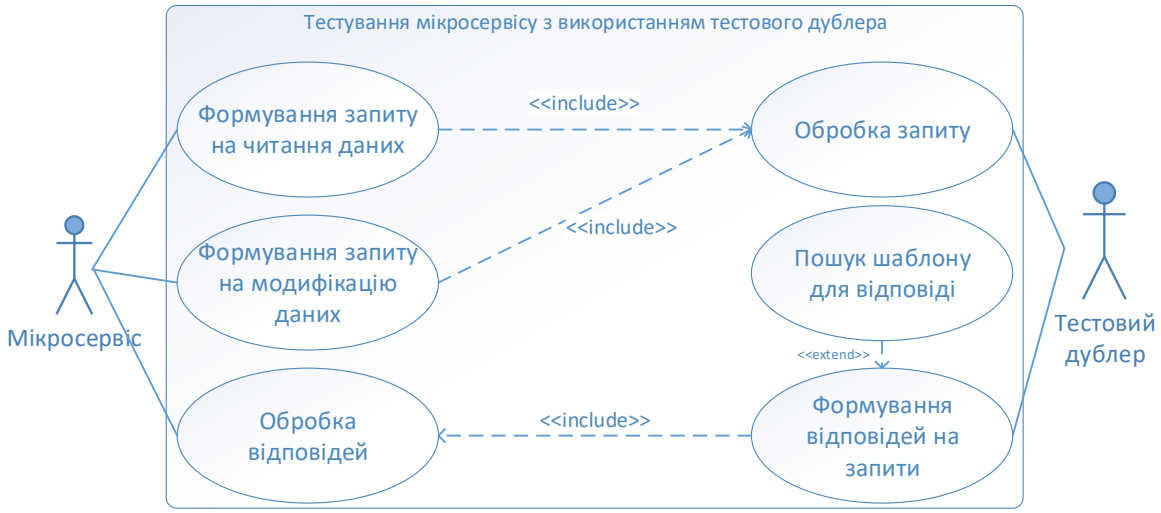

Рис. 3.1. Діаграма варіантів використання процесу тестування мікросервісу

На рис. 3.1 представлений мікросервіс, що взаємодіє з тестовим дублером. При формуванні запиту на читання даних або формування запиту на модифікацію даних завжди слідує Обробка запиту. А формування відповідей запити завжди тягне у себе обробку відповідей. Тому таке ставлення має характер «включення» або <<include>>.

Під час формування відповіді запити з'являється варіант пошуку шаблону для відповіді. Але, оскільки шаблон може бути знайдений не завжди, ставлення має характер «розширення» або «extend».

Діаграма класів - це діаграма, де представлено графічне зображення набору елементів, представлене як зв'язкового графа вершин (сутностей) і шляхів (зв'язків). Діаграми представляються як одна з форм статичного опису стану системи з погляду її проектування, відбиваючи її структуру. При цьому діаграма класів не показує дії об'єктів у динаміці, тут відображаються лише статичний стан класів, інтерфейсів та його відносин.

Приклад діаграми класів взаємодії запитів та відповідей з використанням тестового дублера зображено на рис. 3.2:

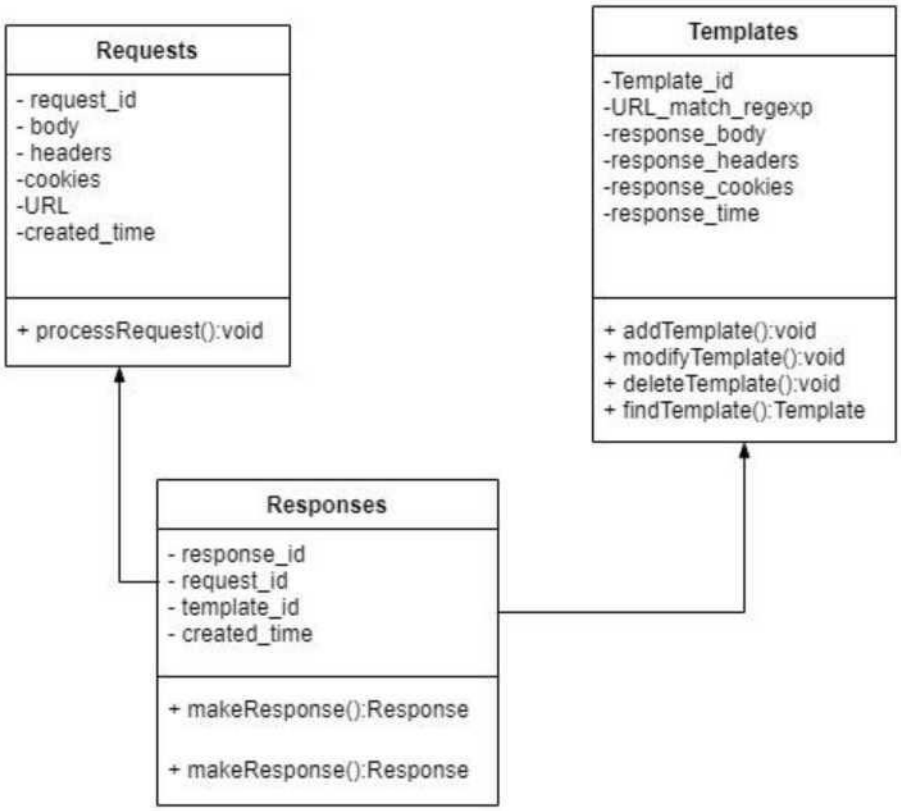

Рис. 3.2. Діаграма класів тестового дублера

На рис. 3.2 у діаграмі класів виділено 3 класи: Запити (Requests), Відповіді (Responses), Шаблони (Templates). У кожному класі виділені атрибути-властивості класу та методи-операції класу. Взаємодія між класами носить характер асоціації, тому що жоден клас не включає інший, і клас Responses посилається на клас Templates і на клас Requests (атрибути класу Responses включають response_id і request_id). Таким чином, тестовий дублер на кожен запит, що входить, шукає шаблон відповіді, і якщо шаблон не знайдений, то повинна повернутися помилка відповіді.

Отже, на підставі діаграми варіантів використання та діаграми класів процесу тестування мікросервісу з використанням тестового дублера стає очевидно, що проектована програмна система взаємодіє із зовнішньою системою, роботу якої імітує тестовий дублер. На кожен запит тестовий дублер шукає шаблон для відповіді, але якщо шаблон не знайдено, то має бути помилка відповіді.

3.2. Алгоритм тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі розробленого на основі BDD

Відповідно до методології BDD необхідно визначити наступні пункти:

- із чого починається процес тестування;
- яку функціональність слід тестувати, а яку ні;
- скільки перевірок відбувається одномоментно;
- який тест є перевіркою;
- у разі тест вважається незавершеним чи результат некоректним.

Відповідно до вищевикладеного, дана методологія передбачає, що імена тестів повинні являти собою цілі пропозиції, які починаються з дієслова в умовному способі і відображають суть бізнес-мети. Бізнес-мета - це результат, який має бути отриманий користувачем у результаті виконання певних кроків.

BDD -сценарій складається із пропозицій, побудованих з деяких елементів:

- конструкція Given визначає початкові умови здійснення операції (визначає те, що спочатку "дано"). Наприклад, вікно введення команди, пошуковий рядок тощо;
- слово When визначає дії, які здійснюють користувач або підсистема і ініціюють процес тестування функції (відповідає на питання «коли?»);
- фраза Then описує очікуваний результат тестування (наприклад, перехід іншу сторінку чи вибірка за заданими критеріями).

На рис. 3.3 представлений приклад сценарію входу в систему системи, що тестується.

```
Feature: To check that home page has loaded in Sistem Informasi SPI
Skenario: To check that home page has loaded
Given I am on Sistem Informasi SPI
When I click on the Login Link
And input username and password
Then I should be on the Home SPI page
```

Рис. 3.3. Вхід до тестованої системи відповідно до методології BDD

Відповідно до рисунка 3.3 сценарій входу в тестовану систему описує, що користувач знаходиться на сайті System Informatsi SPI (GIVEN). При натисканні на клавішу «Авторизуватися» (WHEN) та введення імені користувача та пароля (AND) потрапляє на сторінку Home SPI page (THEN).

Підхід до автоматизованого тестування за методологією BDD значно відрізняється від стандартного. Порівняння алгоритмів підходу до автоматизованого тестування представлено на рис. 3.4.

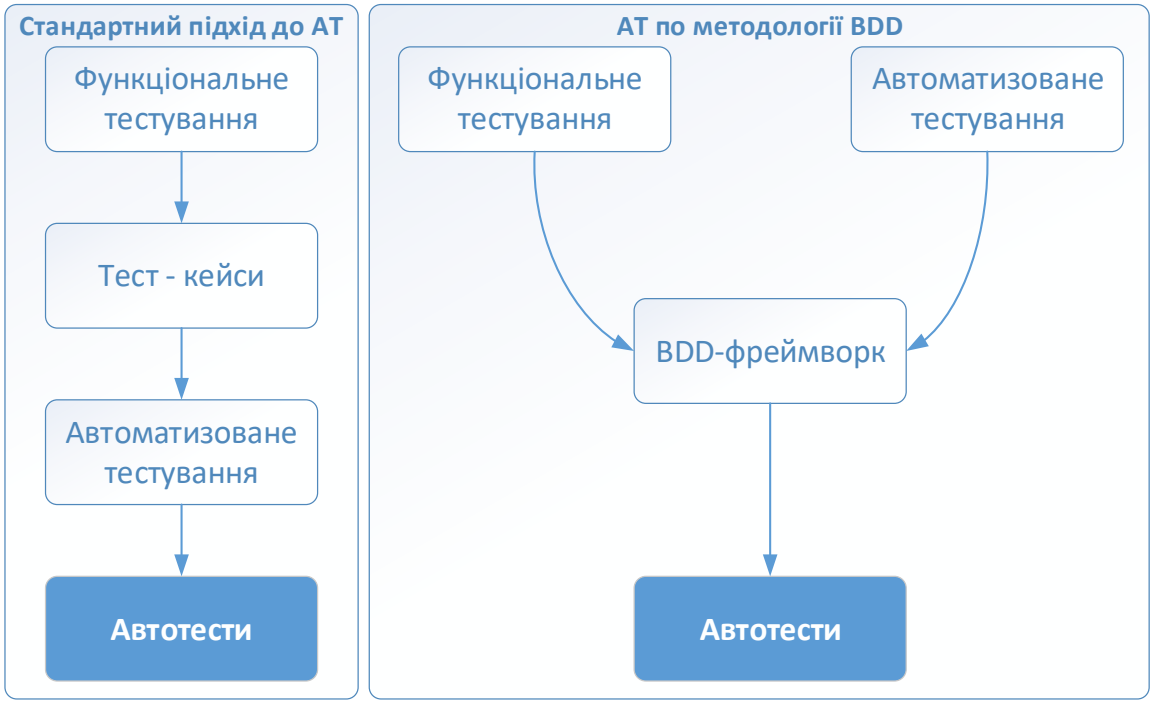

Рис. 3.4. Підходи до автоматизованого тестування

Відповідно до рисунка 3.4 очевидно, що при автоматизації тестування відповідно до методології BDD автотести створюються одночасно за участю і функціональних тестувальників, і фахівців з тестування, що дозволяє заощадити робочий час і бюджет IT- проекту. Крім того, до даного процесу планування тестування доцільно залучати менеджерів програмного продукту з бізнес-аналітиками для того, щоб вони вказували, якому функціоналу необхідно приділити особливу увагу та автоматизувати насамперед з погляду бізнес-значущості функціонального критерію.

Особливе значення слід надати приймальному тестуванню. Схема організації

приймального тестування методології BDD представлена рис. 3.5.

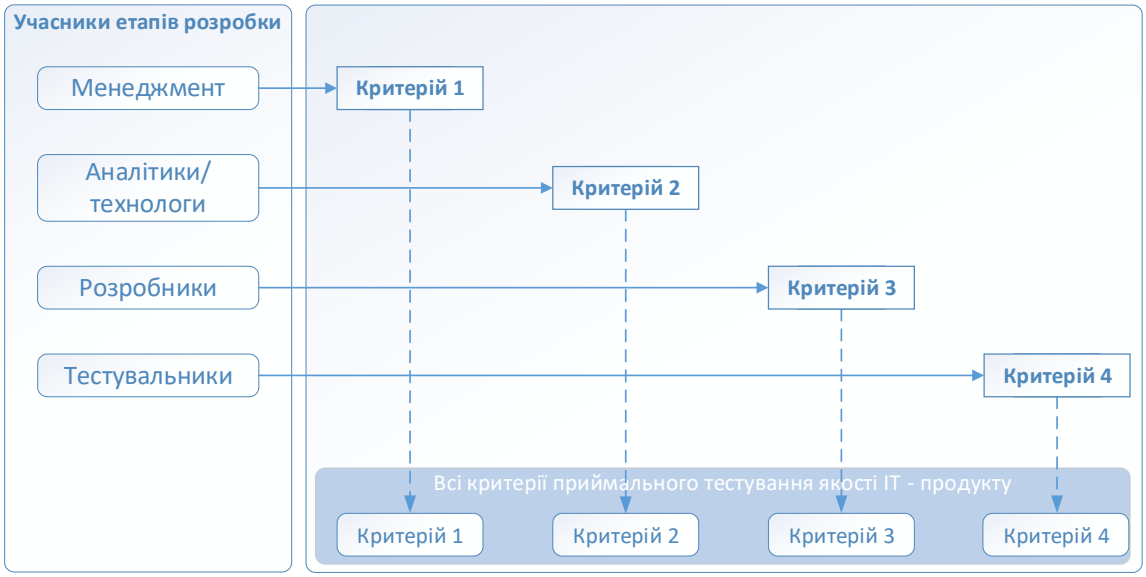

Рис. 3.5. Формування приймального тестування за методологією BDD

Аналіз рис. 3.5 показує, що повний набір критеріїв якості програмного продукту формується не тільки тестувальниками, як це часто відбувається, а всіма учасниками процесу розробки. Даний підхід найбільш корисний і ефективний з точки зору автоматизації, він дає швидке та дуже ефективне складання BDD - сценарію тестування.

Отже, сценарії роботи системи, написані відповідно до методології BDD, зрозумілі і розробникам, і клієнтам (користувачам системи).

3.3. Інструментальні засоби для тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі та розробленого на основі BDD

Специфікація, складена на основі поведінки користувача, вимагає використання обмеженого набору пропозицій, синтаксис яких обмежений. У зв'язку з цим фактором інструментальні засоби, що підтримують BDD, будуються відповідно до таких особливостей:

- парсер розбиває специфікацію деякі частини, наприклад, за ключовими словами мови Gherkin. В результаті формується набір пропозицій, побудованих

відповідно до BDD -синтаксису;

- кожна пропозиція визначає один крок у тестуванні;
- процесор регулярних виразів здійснює захоплення тієї частини пропозиції, в якій містяться вхідні параметри, що ініціюють операцію. Решта пропозиції не використовується, вона необхідна тільки для розуміння події, що відбувається. Захоплені параметри конвертуються та відправляються на вхід до певної операції.

Відповідно до цих принципів працюють такі програмні засоби, як Cucumber, JBehave та JGiven.

Cucumber - кросплатформна програма для автоматизації тестування на основі підходу BDD. Це спеціально розроблена бібліотека для тестування, яка дозволяє розробляти тестові набори природною мовою з подальшою конвертацією в текстовий файл з розширенням .feature. У Cucumber для написання тестів використовується мова Gherkin, яка визначає структуру тесту та набір ключових слів. Шаблоном складання сценарію тестування є ключові команди, такі як Given, When, Then.

JBehave - це додаток, написаний мовою Java, допомагає автоматизувати приймальні тести та підтримує процес розробки у BDD стилі. Бізнес-вимоги та їх приймальні критерії описуються у вигляді користувацьких сценаріїв. Далі ці сценарії читаються як звичайний текстовий файл. Для того, щоб зіставляти команди з пропозицією Gherkin, використовуються Java -анотації, які представлені у фреймворку JBehave. JGiven – це зручний інструмент BDD для Java. Розробники пишуть сценарії простою мовою Java, використовуючи вільний API для конкретного домену. JGiven генерує звіти, які можуть бути прочитані експертами у предметній галузі.

Переваги та недоліки розглянутих інструментальних засобів представлені у таблиці 3.1.

Таблиця 3.1

Порівняльний аналіз інструментальних засобів методології BDD

додаток	Переваги	Недоліки

Cucumber-JVM	1. Детальна документація 2. Підтримка багатьох мов при описі сценаріїв 3. За результатами проходження тестів генерується звіт у досить детальній та легкочитаній формі 4. Підтримує функції	Не підтримує паралельні випробування JUnit. Однак працюватиме з паралельними збираннями Maven 3
JBehave	1. Відкритий вихідний код 2. Має безліч додаткових конфігурацій для точного налаштування інструменту BDD відповідно до ваших уподобань	1. Підтримує лише історії, а не функції 2. відсутні ключові особливості мови Gherkin, такі як backgrounds, doc strings та tags
JGiven	1. Працює з усіма існуючими IDE та інструментами складання для Java 2. Генерує HTML -звіти, які можуть бути легко прочитані та зрозумілі 3. Підтримка параметризованих кроків 4. Підтримка тегів для організації сценаріїв	Більше підходить для модульних та інтеграційних тестів, ніж для автоматичних системних та регресійних тестів.

Таким чином, всі розглянуті інструменти для тестування працюють відповідно до методології BDD, є Java -додатками, в яких для написання сценаріїв тестування використовуються кроки Given, When, Then. Але найпоширенішим і найзручнішим з погляду доступності технічної документації, а також для отримання згенерованих та зрозумілих звітів про тестування доцільно використовувати Cucumber-JVM. У цій бібліотеці кожному кроці відповідає інструкція, яка з допомогою регулярного висловлювання пов'язує метод, з якого оголошено, з рядком у текстовому описі сценарію. Етапи тестування формуються у сценарії (Scenario), які описують певну функціональність (Feature).

Структура проекту з використанням бібліотеки Cucumber-JVM представлена рис. 3.6.

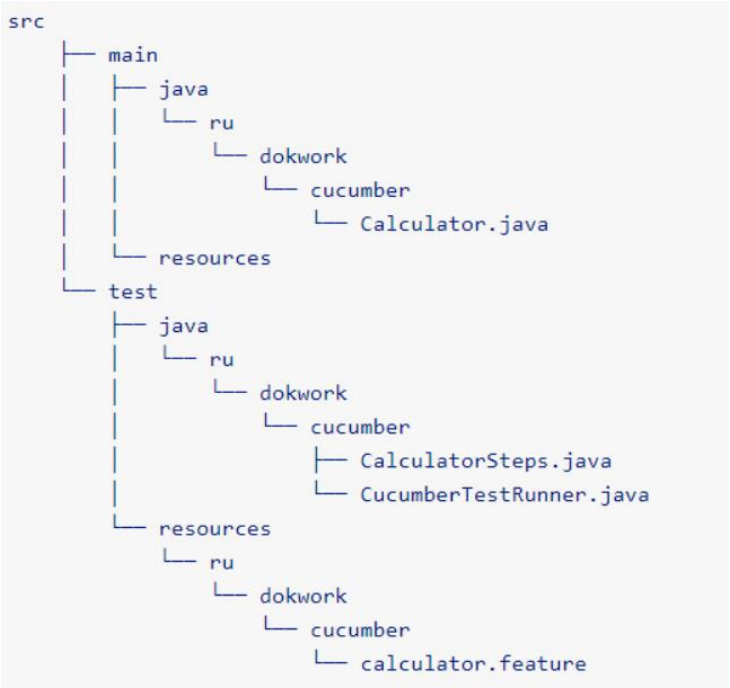

Рис. 3.6. Структура проекту з використанням бібліотеки Cucumber-JVM

Отже, на підставі рис. 3.6 можна дійти до висновку, що основний файл - це текстовый. feature файл із описом сценаріїв і .java файли з описом реалізації кроків виконання сценаріїв. Жодних обмежень, крім розширення, на імена цих файлів не накладається. Але звідси випливає ще один момент - крок, описаний в одному текстовому файлі, буде використовуватися у всіх java -реалізаціях.

Таким чином, можна зробити висновок, що існує безліч інструментальних засобів для тестування, але остаточний вибір залежить від методології розробки, мови програмування та зручності використання.

3.4. Сценарій автоматизованого тестування хмарного програмного продукту, збудованого на мікросервісній архітектурі, з використанням тестових дублерів

Для складання сценарію тестування розглянемо етап тестування будь-якого Інтернет-магазину. Інтернет-магазин взаємодіє з трьома мікросервісами: сервіс доставки товарів, сервіс оплати та сервіс повідомлень про поточний стан замовлення. Моделювана система взаємодії мікросервісів представлена рис. 3.7.

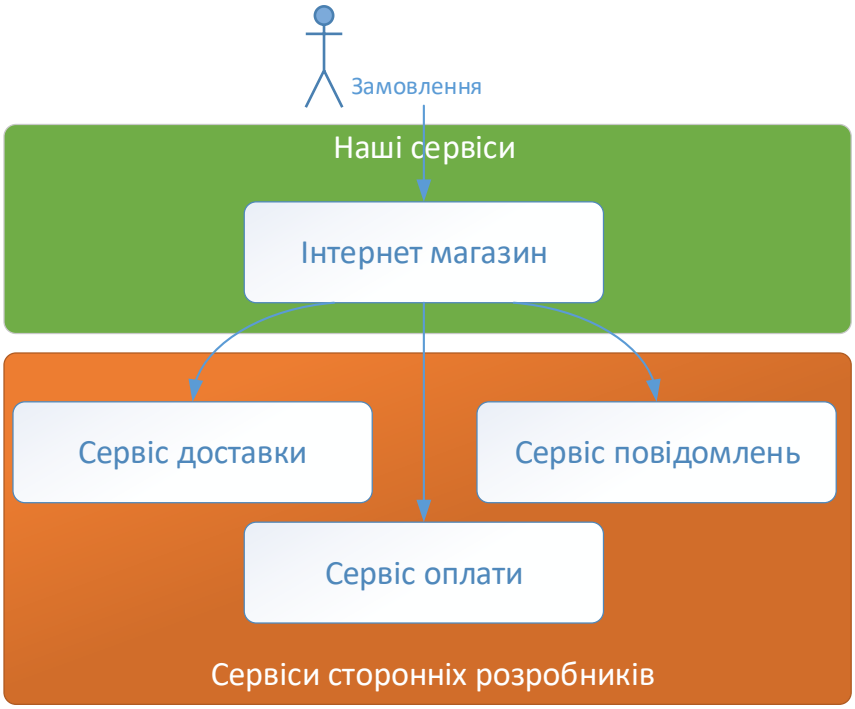

Рис. 3.7. Система взаємодії Інтернет-магазину з зовнішніми мікросервісами

Відповідно до рис. 3.7 нашою зоною відповідальності є лише робота Інтернет-магазину. Але для тестування його роботи необхідно провести інтеграційне тестування із зовнішніми мікросервісами, які використовуються і за які відповідає стороння компанія.

Для знаходження найбільшої кількості помилок під час тестування змодельованої системи необхідно поетапно перевіряти взаємодію Інтернет-магазину з кожною зовнішньою підсистемою. Для цього доцільно заглушити всі пов'язані мікросервіси, крім тих, що тестуються, щоб в область тестування потрапила мінімально допустима кількість операцій.

Діаграма послідовності взаємодії Інтернет-магазину із зовнішніми мікросервісами представлена рис. 3.8.

На рис. 3.8 представлений життєвий цикл змодельованого Інтернет-магазину та його взаємодія із зовнішніми мікросервісами, що співпрацюють (система оплати, система доставки, система повідомлень). Для того, щоб сфокусуватися тільки на тестуванні певних мікросервісів, необхідно заглушити решту всіх підсистем. Таким чином, щоб перевірити взаємодію Інтернет-магазину з мікросервісом "система оплати", потрібно заглушити "систему доставки" та "систему повідомлень". Тоді

може виникнути необхідність використання тестового дублера (сервісу-заглушки) для імітації комплексної взаємодії всієї системи.

Цей спосіб дозволить ізолювати мікросервіс, що тестується, але гарантує те, що інші мікросервіси доступні і можливо налаштувати підключення тестованої підсистеми до співпрацюючих сервісів-заглушок. Далі заглушки конфігуруються для відправлення зворотних відповідей із єдиною метою зімітувати взаємодія.

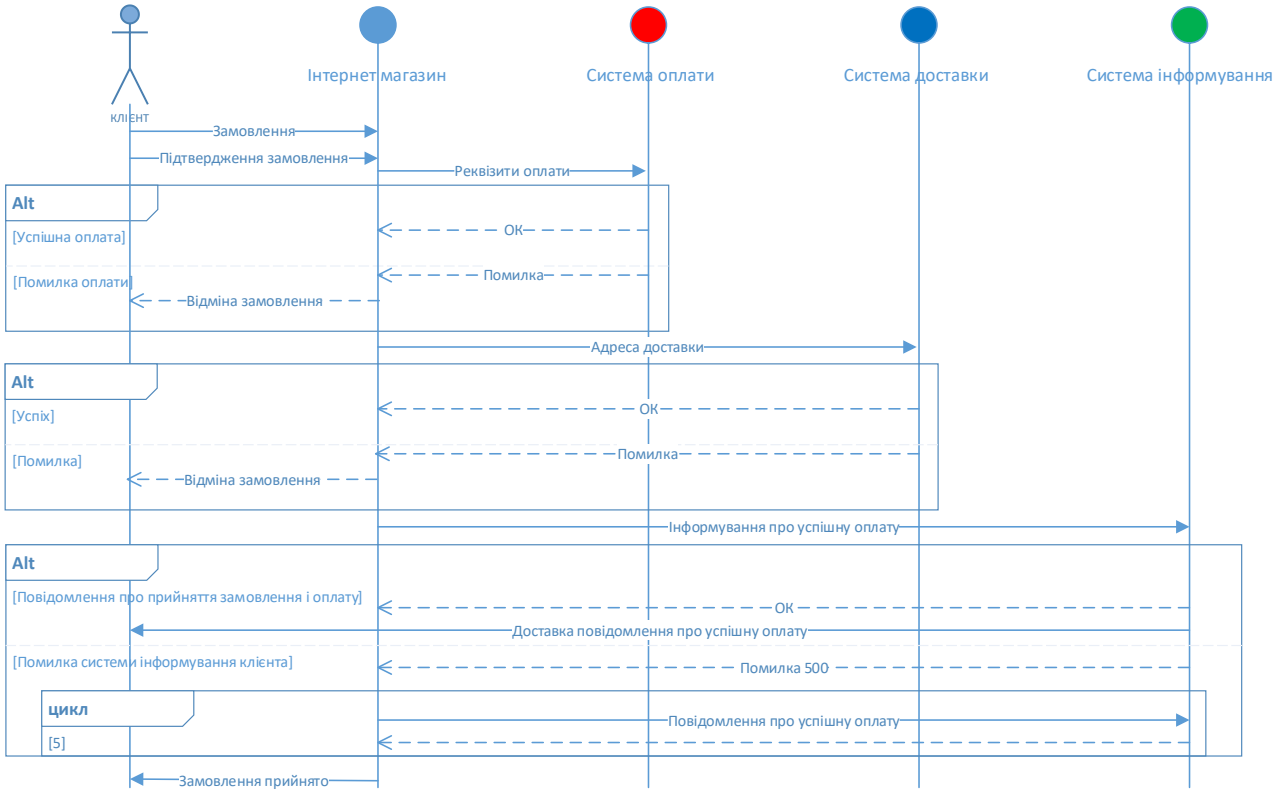

Рис. 3.8. Діаграма послідовності взаємодії Інтернет-магазину з зовнішніми мікросервісами

У разі успішного оформлення та підтвердження замовлення Інтернет-магазин надсилає запит мікросервісу «сервіс оплати». Повідомленням у відповідь може бути як підтвердження оплати («ОК»), так і помилка при оплаті («помилка»). Внаслідок помилки відбувається скасування замовлення, і взаємодії із системою доставки та системою повідомлень не відбувається. Для інтеграційного тестування Інтернет-магазину із мікросервісом «система доставки» необхідно отримати відповідь від «сервісу оплати». Імітацію його діяльності виконує тестовий дублер, надсилаючи

повідомлення у відповідь про успішну оплату замовлення. Аналогічно відбувається налаштування тестового дублера для «системи доставки» для тестування взаємодії Інтернет-магазину із зовнішнім мікросервісом «система повідомлень».

Для тестування досліджуваної системи необхідний наступний набір тест-кейсів:

- Тестування успішного замовлення. Користувач виконує кроки: створює замовлення, вводить картки для оплати, підтверджує замовлення. Очікуваний результат поведінки системи, що тестується:

а) тестовий дублер платіжної підсистеми отримав запит;

б) перевіряємо запит, отриманий тестовим дублером платіжної підсистеми на відповідність специфікації;

в) перевіряємо, що тестовий дублер підсистеми доставки отримав запит;

г) перевіряємо запит, отриманий тестовим дублером підсистеми доставки на відповідність специфікації "Успішна доставка";

д) перевіряємо, що тестовий дублер підсистеми повідомлень отримав запит;

е) перевіряємо запит, отриманий тестовим дублером підсистеми повідомлень клієнта на відповідність специфікації "Успішна оплата";

ж) перевіряємо статус замовлення = "Підтверджено".

- Перевірка введення невірних даних під час оплати замовлення. Користувач виконує кроки: створює замовлення, вводить невірні дані картки для оплати, підтверджує замовлення. Очікуваний результат поведінки системи, що тестується:

а) перевіряємо, що тестовий дублер підсистеми повідомлень отримав запит;

б) перевіряємо запит, отриманий тестовим дублером підсистеми повідомлень клієнта на відповідність специфікації "Помилка оплати";

в) перевіряємо, що тестовий дублер підсистеми доставки не отримав запит;

г) перевіряємо статус замовлення = "У процесі";

д) перевіряємо помилку = "Введено неправильні дані для оплати".

- Перевірка недоступності доставки. Користувач виконує кроки: створює замовлення, вводить правильні картки для оплати, підтверджує замовлення. Очікуваний результат поведінки системи, що тестується:

а) перевіряємо, що тестовий дублер підсистеми повідомлень отримав запит;

б) перевіряємо запит, отриманий тестовим дублером підсистеми повідомлень клієнта на відповідність специфікації "Успішна оплата";

в) перевіряємо, що тестовий дублер підсистеми доставки отримав запит;

г) перевіряємо, що тестовий дублер підсистеми доставки відповів помилкою;

д) перевіряємо статус замовлення = "Скасовано";

е) перевіряємо помилку = "Неможливо призначити доставку";

- Перевірка роботи сервісу повідомлень (недоступність повідомлень). Користувач виконує кроки: створює замовлення, вводить правильні картки для оплати, підтверджує замовлення. Очікуваний результат поведінки системи, що тестується:

а) перевіряємо, що тестовий дублер підсистеми повідомлень отримав запит;

б) перевіряємо запит, отриманий тестовим дублером підсистеми повідомлень клієнта на відповідність специфікації "Успішна оплата";

в) перевіряємо, що тестовий дублер підсистеми повідомлень відповів помилкою;

г) перевіряємо статус замовлення = "Підтверджено";

д) перевіряємо, що в полі «Помилка» замовлення виходить помилка «Збій системи повідомлень»;

е) чекаємо 5 хвилин, перевіряємо, що тестовий дублер підсистеми повідомлень клієнта отримав лише 5 запитів.

Ідея розглянутих тест-кейсів полягає в пошуку максимальної кількості помилок при інтеграційному тестуванні Інтернет-магазину із зовнішніми мікросервісами, що співпрацюють. Слід зазначити, що при тестуванні сервісу

повідомлень важливим моментом є налаштування часу очікування (тест-кейс 1.4 крок № 6, для прикладу ми взяли час очікування 5 хвилин). Для правильної інтеграції системи з нижчими мікросервісами цей етап є ключовим. Якщо час очікування зробити занадто маленьким, потенційно працездатний сервіс можна вважати непрацюючим. У протилежному випадку, якщо налаштувати дуже великий час очікування, то вся система може «зависати» на дуже довгий час, неприйнятний для комфортної роботи. Ідеальним варіантом є налаштування часу очікування для всіх дзвінків, що адресуються за межі своєї системи. Для проведення автоматизованого тестування змодельованої системи використали Cucumber-JVM. У зв'язку з тим, що шаблоном сценарію тестування є кроки, такі як Given, When, Then, всі тест-кейси, розглянуті в таблиці 1, необхідно подати у відповідному вигляді. У додатку Б представлені створені тест-кейси у бібліотеці Cucumber-JVM.

Отже, після запуску всіх чотирьох тест-кейсів, представлених у додатку Б, Cucumber-JVM повідомляє про коректну роботу всієї системи. Успішне виконання всіх тестів представлено рис. 3.9.

З рис. 3.9 можна дійти до висновку, що чотири сценарії пройдено успішно, виконано 40 кроків послідовного тестування змодельованої системи.

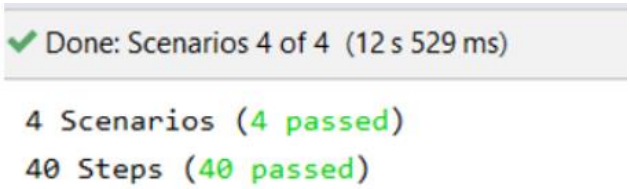

Рис. 3.9. Результат успішного завершення тестування змодельованої системи у бібліотеці Cucumber-JVM

Таким чином, проведення тестування відповідно до методології BDD на основі користувальницьких сценаріїв є зручним і зрозумілим як для замовника, так і для IT- фахівця. Ізольованість кожного зовнішнього мікросервісу дозволяє поетапно тестувати взаємодію змодельованої системи з кожним компонентом, що співпрацює.

Тестові дублери, що імітують відповідні виклики, дозволяють провести тестування своєї системи найбільш комплексно та оперативно.

Отже, в результаті проведення апробації методів проведення автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі, можна зробити висновок про те, що практичне складання BDD -сценарію проведення тестування програмного продукту відбувається за спільною участю замовника, розробників та тестувальників.

В результаті визначається повний набір критеріїв якості готового програмного продукту, складається набір тест-кейсів, написаних мовою BDD -синтаксису. Наступним важливим етапом є визначення інструментального засобу, що працює відповідно до методології BDD. Після проведення тестування має бути згенерований зрозумілий звіт про результати тестування відповідно до складеного сценарію, а також загальний час проведення тестування.

3.5. Переваги використання тестових дублерів як частини програмного продукту

В результаті проведення автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі, з використанням тестових дублерів (сервісів-заглушок) стає очевидним, що такий спосіб дозволяє провести тестування найбільш комплексно.

Але комплексне тестування повністю готового програмного продукту може бути необхідним не тільки під час розробки, а й під час впровадження, встановлення та супроводу готового рішення у замовника.

Таким чином, постачання тестових дублерів як частини готового програмного продукту має такі переваги:

1.Дозволяє перевірити правильність встановлення та налаштування програмного продукту на обладнанні замовника. Справа в тому, що тестовий стенд, на якому проводилися розробка та тестування функціональних можливостей, може суттєво відрізнятись від обладнання та конфігурації у замовника. Відмінності

можуть бути як моделі конкретного обладнання (серверів, мережного устаткування), і у конфігурації мережі (наявність додаткових систем безпеки тощо.). Використовуючи BDD тести, створені на етапі тестування програмного продукту, ми зможемо переконатися, що в рамках нашої зони відповідальності всі підсистеми працюють згідно з вимогами замовника, і інтеграція з усіма зовнішніми системами відбувається за узгодженою специфікацією.

2.Дозволяє перевірити роботу всієї IT- системи у складних умовах (навантажувальне тестування, прикордонні ситуації). Під час приймального тестування замовник перевіряє систему на відповідність усім вимогам відповідно до специфікації, у тому числі обов'язковим є проведення тесту навантаження. У зв'язку з тим, що це надзвичайно трудомістке, взаємодія з іншими підсистемами може бути утруднена (у разі, якщо вони не готові), або зовсім неможливо (робота з платіжними підсистемами або підсистемами сповіщення клієнта поштою або телефоном завжди оплачується окремо). У разі доцільно використовувати тестових дублерів.

3.Дає можливість отримати додаткову вигоду. Замовник зацікавлений самостійно перевірити систему на етапі приймального тестування, але перед ним постають ті самі проблеми, що й у команди тестування під час розробки програмного продукту. Маючи готове рішення для комплексного тестування, стає можливим продати його замовнику, скоротивши таким чином його час на розробку власної автоматизованої системи перевірки, без якої не обійтись при використанні Agile- методології та частого оновлення робочих серверів. Навіть якщо замовник і не зацікавлений у використанні нашого сценарію тестування готового програмного продукту, тестові дублери можуть бути корисними для його власного алгоритму перевірки IT- системи.

4.Дозволяє суттєво спростити роботу команди підтримки та впровадження програмного продукту. При впровадженні програмного продукту можуть виникати несподівані непередбачувані обставини: наприклад, порушення вимог специфікації у сусідніх зовнішніх систем, з якими IT -продукт інтегрується. У разі наявність готових сконфігурованих тестових дублерів полегшує дослідження даних інцидентів, скорочуючи розслідування і виправлення виявлених дефектів,

підвищуючи задоволеність клієнта купленим продуктом.

Отже, розглянувши переваги використання та постачання тестових дублерів як частини готового програмного продукту, слід зазначити один недолік: розробка, встановлення та конфігурація тестового дублера займає певний час. Отже, необхідно враховувати трудовитрати, пов'язані з використанням тестових дублерів.

3.6. Оцінка трудовитрат на встановлення та конфігурацію тестових дублерів

Важливим етапом у розробці та тестуванні будь-якого програмного продукту є оцінка вартості робіт. Доцільно провести зразкову оцінку вартості розробки та впровадження тестових дублерів на устаткуванні замовника, якщо тестові дублери не були включені до програмного продукту на етапі складання специфікації та кошторису. Для оцінки вартості впровадження тестових дублерів на устаткуванні замовника доцільно використовувати триточковий метод або виважена триточкова оцінка.

Триточковий метод (PERT -оцінка) - спосіб оцінки часу та зусиль у діяльності з управління проектами. Відповідно до триточкового методу обчислення відбувається за такою формулою:

$$A = \frac{Best\ case + 4 * Likely\ case + Worst\ case}{6}, \quad (3.1)$$

де А - виважена оцінка (дні); Best case – оптимістична оцінка (дні); Likely case - нормальна оцінка або найімовірніший час виконання (дні); Worst case – песимістична оцінка (дні).

Цей метод розширює спосіб, який враховує лише найкращий і найгірший тимчасовий прогноз. З'являється додатковий критерій «найбільш ймовірний час виконання».

Обсяг роботи з впровадження тестових дублерів на устаткуванні замовника включає такі види робіт:

- планування роботи із замовником - включає обговорення плану робіт, їх узгодження з усіма структурами замовника;

- установка тестового дублера - саме собою установка необхідних мікросервісів, перевірка їх коректної функціональності;

- конфігурування тестового дублера - налаштування емуляцію необхідного поведінки. Залежить від самої задачі, просте налаштування для функціонального завдання, і нетривіальне для навантажувального сценарію;

- непередбачені обставини - існує невизначеність щодо точного змісту всіх елементів в оцінці, як буде виконано роботу, які будуть умови для роботи, тому для обліку даних ризиків робиться поправка, на основі минулого досвіду оцінювача - середня поправка визначається 30%.

Вихідна інформація для розрахунку виваженої оцінки представлено у таблиці 3.2.

Таблиця 3.2

Вихідні дані для розрахунку середньої оцінки триточковим методом

Вид праці	Оптимістична оцінка (дні);	Нормальна оцінка (дні);	Песимістична оцінка (дні);	Середня оцінка (дні);
Планування роботи із замовником	1	2	3	2
Встановлення тестового дублера	1	2	3	2
Конфігурування тестового дублера	1	3	5	3
Непередбачені обставини, 30%			0,3	2,1
Разом	3	7	11,3	9,1

Отже, виходячи з даних таблиці 3.2, загалом 9 днів може знадобитися запровадження тестового дублера на обладнанні замовника. Крім того, щоразу, коли виникне необхідність у перевірці нового сценарію роботи IT- системи, операцію конфігурування тестового дублера доведеться повторювати, а це додаткові тимчасові та грошові витрати.

В таблиці 3.3 наведена середня заробітна плата IT - фахівців в Україні (за даними сайту https://www.work.ua/)

Таблиця 3.3

Середня заробітна плата IT- фахівців за один календарний місяць

Найменування	Мінімум (грн.)	Середній рівень (грн.)	Максимум (грн.)
Програміст JavaScript	24000	56000	94000
HTML -верстальник	12500	23000	40000
Програміст PHP	20000	54000	116000
Програміст Java	18000	92500	151000
Програміст IOS	39000	85000	151000
Програміст 1С	17000	30000	60000
Системний адміністратор	10000	18500	48000

Отже, якщо взяти середню заробітну плату за 1 календарний місяць (22 робочих дні) Java -програміста рівну 56000 грн, а установка тестового дублера займає в середньому 9 днів, виходить, що підсумкова вартість роботи зі встановлення та конфігурування тестового дублера становить 56000*9/22=22909 грн, не враховуючи відрядження та інші витрати.

Без використання тестового дублера для тестування системи, що розробляється, необхідно чекати, коли всі мікросервіси які використовуються будуть розроблені і сконфігуровані на інтеграцію, цей процес може займати кілька місяців. Крім того, після кожного оновлення будь-якого з мікросервісів будуть потрібні додаткові витрати на підтримку працездатності всієї системи.

Таким чином, постачання тестового дублера як частини комплексного програмного продукту дозволяє не лише спростити процес впровадження, підтримки та тестування програмного продукту у замовника, а й полегшує самому замовнику процес тестування, а також відкриває можливості додаткового заробітку на продажі власних інструментів тестування замовнику. В результаті попереднього узгодження із замовником про розробку програмного продукту зі створенням

тестового дублера підсумкова вартість ІТ- рішення для компанії-розробника збільшується несуттєво. Але продати замовнику такий програмний продукт можна суттєво дорожче, аргументуючи тим, що це комплексна ІТ- система, налаштована та протестована зі зв'язаними підсистемами.

Крім того, за допомогою тестових дублерів як частини підсумкового ІТ-рішення можна перевіряти працездатність системи на всіх етапах розробки та тестування, навіть якщо сторонні системи ще не готові.

Тестові дублери, що імітують відповідні виклики, дозволяють провести тестування своєї системи найбільш комплексно та оперативно. Без них тестування сумісності з сервісами, що співпрацюють, стає непередбачуваним і дорожчим. Ізольованість кожного зовнішнього мікросервісу дозволяє поетапно протестувати взаємодію змодельованої системи з кожним компонентом, що співпрацює.

Комплекс усіх розглянутих заходів прискорює процес впровадження та терміни приймального тестування програмного продукту, підвищуючи якість готового ІТ - рішення, а також лояльність та задоволеність замовника.

Підтвердилася гіпотеза про можливість створення тестового дублера, що сприяє підвищенню якості тестування на всіх етапах розробки та тестування програмного продукту.

РОЗДІЛ 4
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1. Охорона праці

Усі дослідження методів тестування проводились з дотриманням правил та норм охорони праці і вимог техніки безпеки.

При роботі за ЕОМ необхідно особливу увагу звертати на правильне освітлення. Неправильне освітлення (пряма та відбита від екранів близькість, вуалюючі відбиття, несприятливий розподіл яскравості в полі зору, невірна орієнтація робочого місця відносно світлових отворів) призводить до негативних фізіологічних впливів на користувачів ЕОМ. Погана якість символів, що представлені на екрані, також може викликати зоровий дискомфорт, бути стресовим фактором та ін.

Вимоги до освітлення для візуального сприймання користувачами інформації з двох різних носіїв (з екрана ЕОМ та паперового носія) різні. Надто низький рівень освітленості погіршує сприймання інформації при читанні документів, а надто високий призводить до зменшення контрасту зображення знаків на екрані. Відношення яскравості екрана ЕОМ до яскравості оточуючих його поверхонь не перевищує у робочій зоні 3:1.

Наближено можна вважати, що при 10%-ному зменшенні освітленості працездатність знижується на 1%. Коли за характером роботи вимагається комбінація цих двох носіїв інформації, освітленість можна варіювати від 300 до 700 лк, причому чим рідшою є зміна полів зору в процесі роботи (з екрана на документ та навпаки), тим вищим може бути рівень освітленості. 300-500 лк — оптимальна освітленість робочих приміщень для роботи з ЕОМ. Стрибки яскравості при зміні полів зору мають бути мінімальними, тобто інтенсивність освітлення поверхні, де знаходяться рукописи та документи, не повинна перевищувати яскравості екрана дисплея.

Освітлення повинно відповідати нормальним рівням за ДБН В.2.5-28:2018 Природне і штучне освітлення.

Приміщення в якому виконувалась кваліфікаційна робота забезпечене природнім і штучним освітленням. При роботі за ЕОМ обрано місце, щоб в поле зору не потрапляли вікна або освітлювальні прилади. Регулювання світлових променів здійснюється за рахунок жалюзі на вікнах. Вікна приміщення орієнтовані на схід.

Штучне освітлення у приміщені реалізовано у вигляді комбінованої системи освітлення з використанням люмінесцентних джерел світла у світильниках загального освітлення, які слід розташовані над робочими поверхнями у рівномірно-прямокутному порядку. Для запобігання засвітленню екранів ЕОМ прямими світловими потоками лінії світильників розташовані з достатнім бічним зміщенням відносно робочих місць, а також паралельно до вікон.

На робочому місці забезпечена рівномірна освітленість за допомогою переважно відбитого або розсіяного світлорозподілу. Світлових відблисків з клавіатури, екрана та від інших частин ЕОМ у напрямку очей користувача немає. Дискомфорт від відбиття світла знижується при збільшенні яскравості екрана та зниженні рівня навколишнього освітлення.

Пульсація освітленості люмінесцентних ламп, що використовуються, відповідно до технічної документації 10%, що відповідає діючим вимогам.

Інформація, яку одержує користувач, генерується на екрані, а комфортність її сприймання залежить від чіткості символів. При обговоренні проблеми дискомфорту або негативних наслідків для здоров'я та ефективності роботи на ЕОМ слід враховувати ряд параметрів. Ці параметри поділені на три групи, пов'язані з

–мигтінням

–структурою

–яскравістю символів, що представляються на екрані.

Отже в даному підрозділі розглянуто вплив середовища на працездатність та здоров’я користувачів комп’ютерів. Як висновок можна сказати, що робоче місце

яке використовувалось для написання даного наукового дослідження відповідає вимогам з охорони праці.

Однак необхідно не забувати що надмірна робота з ПК може привезти до порушення роботи організму користувача. Тому необхідно дотримуватись вимог щодо планування робочого часу за ЕОМ.

4.2. Оцінка надійності захисту виробничого персоналу під час надзвичайних ситуацій

При оцінці надійності захисту виробничого персоналу необхідно враховувати, що практично будь-які наслідки НС можуть призвести до ураження людей та стати причиною їхньої смерті або призвести до втрати працездатності на тривалий час.

Надійність захисту виробничого персоналу є одним з важливих факторів, яки визначають стійкість роботи підприємств у надзвичайних ситуаціях мирного та воєнного часів.

Найбільш ефективним заходом захисту є укриття людей в захисних спорудах (ЗС) при дотриманні таких умов:

- загальна місткість ЗС дозволяє укрити всіх робітників та службовців, тобто весь виробничий персонал об’єкту;

- захисні споруди задовольняють вимогам захисту від усіх небезпечних наслідків НС;

- захисні споруди устатковані системами життєзабезпечення на необхідну тривалість перебування у них;

- розміщення ЗС відносно робочих місць дозволяє своєчасно укритися всім робітникам за сигналами сповіщення про НС;

- робітники та службовці своєчасно сповіщаються та навчені способам захисту та правилам дії за сигналами сповіщення.

За показник надійності захисту робітників та службовців з використанням ЗС можна прийняти коефіцієнт надійності захисту $К_{Н.\ З.}$, що показує яка частина

робітників та службовців забезпечується надійним захистом від усіх небезпечних наслідків виникнення НС.

Коефіцієнт надійності захисту визначається на основі окремих показників, що характеризують підготовленість об'єкту до виконання завдань захисту робітників та службовців за основними складовими задачами.

Оцінка надійності захисту виробничого персоналу проводиться в такій послідовності

1.Оцінюється інженерний захист робітників та службовців об'єкта. Показником інженерного захисту є коефіцієнт $К_{ІНЖ.\ ЗАХ.}$, що показує, яка частина виробничого персоналу працюючої зміни може укритися своєчасно в ЗС з достатніми захисними властивостями та системами життєзабезпечення, які дозволяють укривати людей протягом встановленого терміну:

$$К_{ІНЖ.ЗАХ.} = \frac{N_{ІНЖ.ЗАХ.}}{N},$$

де N – це чисельність найбільшої працюючої зміни.

2.Вивчається система сповіщення та оцінюється можливість своєчасного доведення сигналу сповіщення до робітників та службовців. Показником надійності з урахуванням сповіщення є коефіцієнт $К_{СП}$: $К_{СП} = \frac{N_{СП}}{N}$

3.Оцінюється навченість виробничого персоналу способам захисту та діям за сигналами сповіщення. Показник – коефіцієнт навченості $К_{НАВЧ}$:

$$К_{НАВЧ} = \frac{N_{НАВЧ}}{N}$$

4.Визначається готовність сховища до прийому людей. Для цього визначається час, протягом якого сховища, що використовуються за подвійним призначенням, можуть бути підготовлені до прийому людей (звільнюються від сторонніх речей, поновлюється запас їжі, води, здійснюється перевірка герметичності, функціонування систем життєзабезпечення). Порівнюючи фактичний час підготовки сховища $Т_{Г.\ ФАК.}$з потрібним $Т_{Г.\ ПОТ.}$, визначається готовність сховища до прийому людей. Для оцінки надійності захисту враховуються лише ті сховища, для яких:

$$\frac{T_{г.фак}}{T_{г.пот}} \leq 1.$$

5.Показником надійності захисту з урахуванням готовності є коефіцієнт готовності $К_{ГОТ}$:

$$К_{ГОТ} = \frac{N_{ГОТ}}{N}$$

На основі окремих показників визначається коефіцієнт надійності захисту робітників та службовців $К_{Н.З.}$за мінімальним значенням окремих показників: $К_{ІНЖ.ЗАХ}$, $К_{СП.}$, $К_{НАВЧ}$, $К_{ГОТ}$.

Визначаються слабкі місця в підготовці об’єкту до успішного вирішення задачі захисту виробничого персоналу у надзвичайних ситуаціях та передбачаються можливі шляхи підвищення показників надійності захисту.

У висновках вказується:

−надійність захисту робітників та службовців;

−необхідність підвищення захисних властивостей наявних захисних споруд та заходи для підвищення надійності;

−приміщення, які доцільно пристосувати під ЗС;

−кількість та тип ЗС, що швидко зводяться;

−заходи надійного захисту чергового персоналу;

−заходи з повного забезпечення персоналу ЗІЗ;

−заходи покращення умов зберігання, профілактики та ремонту ЗІЗ.

Оцінка інженерного захисту полягає у визначенні показників, що характеризують здатність інженерних споруд забезпечити надійний захист людей: це показники за місткістю захисної споруди - КВМ., показник за здатністю захисної споруди відповідати захисним вимогам - КЗ.Т., показник за здатністю систем життєзабезпечення захисної споруди забезпечити усім необхідним тих хто укриваються протягом усього терміну укриття - КЖ.О, показник за здатністю виробничого персоналу своєчасно зайняти захисну споруду - КСВР.

При здійсненні оцінки надійності захисту виробничого персоналу спочатку визначають максимальні параметри тих вражаючих факторів, яки можуть суттєво впливати на надійність захисту:

–для землетрусу – за інтенсивністю землетрусу визначають відповідну йому величину надмірного тиску;

–для аварії на атомної електростанції – визначають напрямок розповсюдження хмари зараженого повітря, розраховують час початку формування сліду радіоактивної хмари на об’єкті і рівень радіації на об’єкті на одну годину після аварії;

–для аварії на хімічно небезпечному підприємстві - визначають напрямок розповсюдження хмари зараженого повітря, глибину зони хімічного зараження, час підходу хмари зараженого повітря до об’єкта, тривалість дії джерела забруднення ;

–для пожежі - визначають напрямок розповсюдження пожежі, час підходу пожежі до об’єкту, можливу тривалість горіння;

–для катастрофічного затоплення - визначають час підходу хвилі прориву до об’єкту, можливу висоту хвилі прориву на об’єкті, час спорожнення водосховища.

Далі визначаються значення окремих показників за різними напрямками інженерного захисту.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розглянуто основні методи та особливості автоматизованого тестування програмного продукту, побудованого на мікросервісній архітектурі. Виявлено, що використання хмарного простору надає зручний та швидкий доступ до ресурсів, що дозволяє з мінімальними управлінськими зусиллями розробити та протестувати програмний продукт.

Під час проведення досліджень у роботі отримано такі теоретичні та практичні результати.

1. В результаті розгляду теоретичних основ існуючих засобів та методів автоматизованого тестування програмного продукту було встановлено основні переваги та недоліки, а також виділено особливості проведення автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісної архітектури. Виділено критерії ефективності автоматизованого тестування, а також критерії формування тестових наборів.

2. Аналіз методів проведення автоматизованого тестування дозволив виділити методи та інструментальні засоби, що дозволяють найбільш повно та швидко протестувати хмарний програмний продукт.

3. На основі аналізу методів проведення автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі, було встановлено, що використання тестових дублерів, що імітують відповідні виклики підсистем, що співпрацюють, дозволяє провести тестування своєї системи найбільш комплексно і оперативно.

4. Результатом розгляду сучасних методологій розробки програмних продуктів з інтегрованим процесом тестування стало виділення методології, яка дозволяє найбільш швидко та якісно скласти план тестування, зрозумілий як замовникам, так і команді розробників та тестувальників.

5. Розглянуто низку інструментальних засобів, що надають можливості для проведення автоматизованого тестування хмарного програмного продукту. Виділено переваги та недоліки з точки зору зручності використання.

6. Складено сценарій проведення автоматизованого тестування хмарного програмного продукту, збудованого на мікросервісній архітектурі з використанням тестових дублерів.

7. В рамках апробації виділених методів, сценарію та інструментального засобу для проведення автоматизованого тестування було складено набір тест-кейсів для тестування змодельованого хмарного програмного продукту, побудованого на мікросервісній архітектурі. Описано хід та результат тестування.

8. В результаті аналізу використання тестових дублерів було виділено переваги даного підходу, проведено оцінку трудовитрат на встановлення та конфігурацію тестового дублера на обладнанні замовника. Дано низку рекомендацій для підвищення якості, надійності та швидкості доставки оновлень замовнику.

Таким чином, запропонований сценарій (алгоритм) проведення автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі, дозволяє значно покращити та оптимізувати кінцеву вартість, якість та час виходу на ринок програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Beck K. Test Driven Development: By Example. Addison-Wesley Professional, 2021. 242 p.

2.Rex Black Critical Testing Processes: Plan, Prepare, Perform, Perfect. 1st Edition. AddisonWesley Professional, 2003. 608 p.

3.Береза А. М. Основи створення інформаційних систем. К.: КНЕУ, 2001. 201 с.

4.Crispin, Lisa. Agile Testing A Practical Guide for Testers and Agile Teams. Addison-Wesley, 2008. 534 p.

5.Graham Lee Test-Driven iOS Development (Developer's Library). Addison-Wesley Professional, 2012. 256 p.

6.Dustin E. Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance. Addison-Wesley Professional, 1999. 608 p.

7.Crispin L. Gregory J. Agile Testing: A Practical Guide for Testers and Agile Teams 1st Edition. AddisonWesley Professional, 2008. 576 p.

8.Авраменко А.С., Авраменко В.С., Косенюк Г.В. Тестування програмного забезпечення. Навчальний посібник. Черкаси: ЧНУ імені Богдана Хмельницького, 2017. 284 с.

9.Канер К., Фолк Д., Нгуєн Е. Тестування програмного забезпечення. Фундаментальні концепції управління бізнес-додатків. К.: ДіаСофт, 2018. 544 с.

10.Robert Culbertson, Chris Brown, Gary Cobb Rapid testing. Pearson Education Limited, 2002. 416 p.

11.Software Quality Management Techniques and Best Practices. URL: https://www.xenonstack.com/insights/what-is-software-quality (дата звернення: 12.12.2022).

12.Elijah J. Automation of Requirement Analysis in Software Engineering. International Journal on Recent and Innovation Trends in Computing and Communication, Volume: 5 Issue: 5, 2017. 1173-1188 p.

13.The Art of Unit Testing: with examples in C# 2nd Edition. Manning; 2nd edition, 2013. 296 p.

14.Чайковський А.В., Жаровський Р.О., Лещишин Ю.З Конспект лекцій з дисципліни «Дослідження і проєктування комп'ютерних систем та мереж» для студентів спеціальності 123 - Комп'ютерна інженерія. Тернопіль, 2021. 148 с.

15.Свергун С., Жаровський Р. Тестування програмного забезпечення побудованого на мікросервісній архітектурі. Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі системи та технології» (7-8 грудня 2022 року). Тернопіль: ТНТУ. 2022. С. 92.

16.Свергун С., Жаровський Р. Тестування програмного продукту, побудованого на мікросервісній архітектурі на основі BDD. Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі системи та технології» (7-8 грудня 2022 року). Тернопіль: ТНТУ. 2022. С. 93.

17.Newman S. Building Microservices, 2nd Edition. O`Reilly, 2015. 280 p.

18.James A. Whittaker, Jason Arbon, Jeff Carollo How Google Tests Software. Addison-Wesley, 2012. 281 p.

19.Фуфаєв Д.Е. Розробка та експлуатація автоматизованих інформаційних систем. К.: Академія, 2017. 304 с.

20.Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, 2010. 512 p.

21.Baboi M., Iftenea A., Gîfu D. Dynamic Microservices to Create Scalable and Fault Tolerance Architecture. URL: https://www.sciencedirect.com/science/article/pii/S187705091931467X (дата звернення: 12.12.2022).

22.Banica L., Stefan C., Hagiu A. Leveraging microservice architecture for next generation IOT applications. URL: https://doaj.org/article/e72c88c1a95c4487b28a8daeb29b18c4 (дата звернення: 12.12.2022).

23.Boncea R., Zamfiroiu Alin, Bacivarov I. Складна архітектура для автоматичного monitoring of microservice. URL: https://doaj.org/article/

4371304a09964a60b26fa20a4fa90b97 (дата звернення: 1.12.2022)

24.Divya K., Mishra K. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. URL: https://www.sciencedirect.com/science/ article/pii/S1877050916001277?via%3Dih ub (дата звернення: 1.12.2022)

25.Dani A., Hadas Schwartz C., Yaron T., Bures M., Shlomo M. Conceptual Approach for Reuse of Test Automation Artifacts on Various Architectural Levels. URL: http://apps.webofknowledge.com/ full_record.do? product=WOS&search_mode=Ge neralSearch&qid=25&SID= D4YhQP7JFpNkFEdgrze&page=1&doc=10 (дата звернення: 1.12.2022)

26.Flemstrom D., Potena P. Similarity-based prioritization of test case automation URL: https://link.springer.com/article/10.1007 % 2Fs11219-017-9401-7 (дата звернення: 1.12.2022)

27.Sulabh T., Ritu S., Bharti S. Adopting Test Automation on Agile Development Projects: A Grounded Theory Study of Indian Software Organizations. URL: https://link.springer.com/chapter/10.1007/978-3-319-57633-6_12 (дата звернення: 1.12.2022)

28.Ulewicz S., Vogel-Heuser B. Increasing system test coverage in production automation systems. URL: https://www.researchgate.net/publication/ 323116397_Increasing_system_test_cov erage_ in_production_automation_systems (дата звернення: 1.12.2022)

29.Zheng L., Wei B. Application of microservice architecture in cloud environment project development URL: https://doaj.org/article/844c18ea51ab4f499c88df2704447fa8 (дата звернення: 1.12.2022)

Додаток А.

Тези конференцій

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

X НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

7–8 грудня 2022 року

ТЕРНОПІЛЬ
2022

УДК 004.416.2
С. Свергун, Р. Жаровський
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПОБУДОВАНОГО НА МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

UDC 004.416.2
S. Svergun, R. Zharovskyi

TESTING OF SOFTWARE BUILT ON MICROSERVICE ARCHITECTURE

Останнім часом особливої актуальності у сфері інформаційних технологій набувають питання якості програмного продукту. Якість програмного продукту безпосередньо залежить від того, наскільки результат розробки відповідає очікуванням замовника. Одним із методів забезпечення якості програмних продуктів є тестування.

Тестування програмного забезпечення є по суті послідовним проходженням програми через створені тестові набори (тест-кейси), які дозволять всебічно перевірити кожну окрему функцію та весь функціонал у цілому.

Мікросервісна архітектура стала найкращою альтернативою для монолітних, складних та негнучких додатків. При тестуванні IT -продукту, побудованого на мікросервісної архітектурі, вся система поділяється більш дрібні, модульні, спільно працюючі компоненти. Їх набагато простіше створювати, оновлювати та тестувати, ніж додаток загалом. Весь процес розробки та тестування мікросервісів представлений на рисунку 1.

Рисунок 1. Етапи розробки та тестування ПЗ побудованого на мікросервісній архітектурі

Особливістю автоматизованого тестування IT -продуктів, побудованих на мікросервісній архітектурі полягає в необхідності автоматизованої інфраструктури, що забезпечує управління життєвим циклом, адресацію та масштабування мікросервісів залежно від поточного завантаження. Необхідність безперервної інтеграції розроблених та/або модифікованих мікросервісів у існуючу систему вимагає всебічного тестування як окремих мікросервісів, так і їхнього спільного функціонування в комплексі з іншими мікросервісами

Таким чином, при виборі будь-якого сценарію тестування мікросервісного IT -продукту необхідне всебічне тестування як окремих мікросервісів, так і їхнього спільного функціонування в комплексі з іншими мікросервісами. Ефективне тестування мікросервісного IT -продукту неможливе без використання тестових дублерів, які спеціально розробляються для задоволення необхідних вимог недоступних модулів.

УДК 004.416.2
С. Свергун, Р. Жаровський
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ, ПОБУДОВАНОГО НА МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ НА ОСНОВІ BDD

UDC 004.416.2
S. Svergun, R. Zharovskyi

TESTING OF SOFTWARE PRODUCT BUILT ON MICROSERVICE ARCHITECTURE BASED ON BDD

Відповідно до методології BDD (Behavior Driven Development) тест-кейси в приймальному тесті повинні охоплювати всі функції програми за різними сценаріями користувачів. BDD методологія передбачає принцип, що полягає в тому, що перед написанням тест-кейсу слід спочатку визначити і сформулювати предметно-орієнтованою мовою результат від проектованої функціональності. Потім усі складені тести перебудовуються фахівцями в BDD -сценарій тестування.

Відповідно до методології BDD необхідно визначити наступні пункти:

- із чого починається процес тестування;
- яку функціональність слід тестувати, а яку ні;
- скільки перевірок відбувається одномоментно;
- який тест є перевіркою;
- у якому випадку тест вважається незавершеним чи результат некоректним.

Тобто дана методологія передбачає, що імена тестів повинні являти собою цілі пропозиції, які починаються з дієслова в умовному способі і відображають суть бізнес-мети. BDD -сценарій складається із пропозицій, побудованих з деяких елементів:

- конструкція Given визначає початкові умови здійснення операції (визначає те, що спочатку «дано»). Наприклад, вікно введення команди, пошуковий рядок тощо;
- слово When визначає дії, які здійснюють користувач або підсистема і ініціюють процес тестування функції (відповідає на питання «коли?»);
- фраза Then описує очікуваний результат тестування (наприклад, перехід іншу сторінку чи вибірка за заданими критеріями).

```
Feature: To check that home page has loaded in Sistem Informasi SPI
Skenario: To check that home page has loaded
Given I am on Sistem Informasi SPI
When I click on the Login Link
And input username and password
Then I should be on the Home SPI page
```

Рисунок 1. Вхід до тестованої системи відповідно до методології BDD

Відповідно до рисунка 1 сценарій входу в тестовану систему описує, що користувач знаходиться на сайті System Informatsi SPI (GIVEN). При натисканні на клавішу «Авторизуватися» (WHEN) та введення імені користувача та пароля (AND) потрапляє на сторінку Home SPI page (THEN).

Перевагою методології BDD є те, що автотести створюються одночасно за участю і функціональних тестувальників, і фахівців з тестування, що дозволяє заощадити робочий час і бюджет IT-проекту. Крім того повний набір критеріїв якості програмного продукту формується не тільки тестувальниками, а всіма учасниками процесу розробки. Тобто даний підхід є найбільш ефективний з точки зору автоматизації. Він дає швидке та дуже ефективне складання BDD -сценарію тестування.

Додаток Б.

Набір тест-кейсів, створених у бібліотеці Cucumber-JVM

```
- Scenario: Successful order

Given created order from template "orderTemplate1"
When user fills payment data from template "paymentTemplate1"
When user submits order
Then check, that payment subsystem received request
Then check, that request to payment subsystem matches spec
"SuccessfulPaymentSpec"
Then check, that notification subsystem received request
Then check, that request to notification subsystem matches spec
"SuccessfulPaymentNotificationSpec"
Then check, that delivery subsystem received request
Then check, that request to delivery subsystem matches spec
"SuccessfulDeliverySpec"
Then check, that order status is "Successful"

- Scenario: Wrong payment data

Given created order from template "orderTemplate1"
When user fills payment data from template "wrongPaymentTemplate1"
When user submits order
Then check, that payment subsystem received request
Then check, that request to payment subsystem matches spec
"WrongPaymentSpec"
Then check, that delivery subsystem did not receive request
Then check, that order status is "In progress"
Then check, that order's error is "Wrong payment data"

- Scenario: Delivery subsystem error

Given created order from template "orderTemplate1"
When user fills payment data from template "paymentTemplate1"
When user submits order
Then check, that payment subsystem received request
Then check, that request to payment subsystem matches spec
"SuccessfulPaymentSpec"
Then check, that notification subsystem received request
Then check, that request to notification subsystem matches spec
"SuccessfulPaymentNotificationSpec"
Then check, that delivery subsystem received request
Then check, that delivery subsystem responded with error 500
Then check, that order status is "Rejected"
Then check, that order's error is "Can't schedule the delivery"

-Scenario: Notification subsystem error

Given created order from template "orderTemplate1"
When user fills payment data from template "paymentTemplate1"
```

```
When user submits order
Then check, that payment subsystem received request
Then check, that request to payment subsystem matches spec
"SuccessfulPaymentSpec"
Then check, that notification subsystem received request
Then check, that notification subsystem responded with error 500
Then check, that order status is "Successful"
Then check, that there is no errors on order
When wait for 5 minutes
Then check, that notification subsystem received 5 requests total
matches spec "SuccessfulPaymentSpec"
```