Вінницький національний технічний університет
(повне найменування вищого навчального закладу)
Факультет інтелектуальних інформаційних технологій та автоматизації
(повне найменування інституту, назва факультету (відділення))
Кафедра Автоматизації та інтелектуальних інформаційних технологій
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до магістерської дипломної роботи
магістр
(освітньо-кваліфікаційний рівень)

на тему: «Розробка та інтеграція сервісу доставки їжі з використанням методів предметно-орієнтованого проектування»

Виконав: студент групи 1АКІТ-21м
Напрям підготовки
151– «Автоматизація та комп'ютерно-інтегровані технології»
(шифр і назва напряму підготовки, спеціальності)

Московко С. Г.
(прізвище та ініціали)
Керівник д.т.н. проф. Квєтний Р.Н.
(прізвище та ініціали)

Рецензент Дубовой В.М.
(прізвище та ініціали)

Вінниця 2022

Вінницький національний технічний університет

(повне найменування вищого навчального закладу)

Факультет інтелектуальних інформаційних технологій та автоматизації

Кафедра Автоматизації та інтелектуальних інформаційних технологій

Освітньо-кваліфікаційний рівень магістр

Напрям підготовки 151– «Автоматизація та комп'ютерно-інтегровані технології»

(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри АІІТ
О.В. Бісікало

"__" _____2022 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ (ПРОЕКТ) СТУДЕНТУ

Московко Сергій Геннадійович

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Розробка та інтеграція сервісу доставки їжі з використанням методів предметно-орієнтованого проектування.

керівник проекту (роботи) Квєтний Р.Н., д.т.н., проф. каф. АІІТ

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від " __ " _____ 2022 року №__

2. Строк подання студентом проекту (роботи) _____

3. Вихідні дані до проекту (роботи) удосконалена методика проектування предметно орієнтованого програмного забезпечення на прикладі розробки онлайн сервісу доставки їжі, що підвищує продуктивність проектування та розробки ПЗ за рахунок покращення взаємодії команди розробників з експертами предметної галузі.

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

1 Сучасний стан проблеми та основні задачі роботи. 2 Дослідження методологій розробки програмного забезпечення. 3 Проектування моделі предметно-орієнтованої системи. 4 Розробка компонентів системи. 5 Тестування та фактори якості програмного забезпечення. 6 Підхід до розгортання програмного забезпечення. 7 Економічний розділ.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Додаток А (обов'язковий) – BPMN діаграма бізнесу у сфері доставки їжі, Додаток Б (обов'язковий) – Діаграма об'єктної взаємодії. Додаток В (обов'язковий) – Діаграма відповідальності об'єкта. Додаток Г (обов'язковий) – UML діаграма класів. Додаток Д (обов'язковий) – Діаграма доменів та обмежених контекстів. Додаток Е (обов'язковий) – Діаграма карти контексту.

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Спеціальна частина	Квєтний Р.Н., д.т.н., проф. каф. АІІТ		
Економічний розділ	Козловський В.О., к.е.н., проф. каф. ЕПОВ		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту роботи)	Примітка
1	Дослідження методологій розробки програмного забезпечення		
2	Проектування моделі предметно-орієнтованої системи		
3	Розробка компонентів системи		
4	Реалізація підходу до розгортки програмного забезпечення		
5	Економічний розділ		

Студент _____ Московко С.Г.
(підпис) (прізвище та ініціали)

Керівник проекту (роботи) _____ Квєтний Р.Н.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

У даній магістерській кваліфікаційній роботі було розроблено та інтегровано високоефективний сервіс доставки їжі з використанням практик предметно орієнтованого проектування. Було удосконалено методику предметно орієнтованого проектування за рахунок представлення нового підходу до проектування бізнес моделі розроблюваного програмного забезпечення.

ABSTRACT

In this paper developed and integrated highly efficient delivery service by using various practices of domain-driven design. The method of domain-driven design was improved by a presentation of a new approach to designing the business model of the developed software.

ЗМІСТ

ВСТУП

Актуальність. Економічне зростання та розвиток широкосмугового інтернету є рушійною силою глобальної експансії електронної комерції. Споживачі все частіше використовують онлайн-послуги, оскільки їхні наявні доходи зростають, електронні платежі стають більш надійними, а асортимент постачальників і розмір їх мереж доставки розширюються [1-3].

Від онлайн до офлайн (Online to offline, O2O) – це форма електронної комерції, у якій споживачів приваблює продукт або послуги онлайн і спонукають до завершення транзакції в офлайн режимі. Зона комерції O2O що швидко розширюється є використанням онлайн-платформ доставки їжі. У всьому світі розвиток сервісів онлайн доставки їжі змінило спосіб взаємодії споживачів і постачальників продуктів харчування, і вплив сталого розвитку (що визначається за економічними, соціальними та екологічними показниками) [2].

Натомість, як і у будь-яких сферах економіки з часом з’являються фактори які перешкоджають розвитку комерційного сектору. Одним з таких факторів є великі компанії які на етапі зародження ринку та за відсутності конкуренції утворили монопольне становище [4,5].

Актуальним є створення сервісу доставки їжі з використанням практик предметно орієнтованого проектування та відкритим вихідним кодом, що дозволить малому бізнесу зменшити витрати на опанування нового комерційного сектору та спонукати його розвиток.

Метою роботи є розробка та інтеграція високоефективного сервісу доставки їжі з використанням практик предметно орієнтованого проектування.

Для досягнення мети необхідно розв’язати наступні задачі:

1. Провести огляд сектору онлайн-доставки їжі.
2. Провести аналіз існуючих практик проектування програмного забезпечення.
3. Розробити модель домену предметної області.

4. Розробити бізнес-правила додатку, що реалізують усі випадки використання системи.

5. Реалізувати клієнтську частину для взаємодії з системою.

6. Розробити тести для тестування компонентів системи.

7. Реалізувати підхід до розгортання програмного забезпечення

Об'єктом дослідження є процеси надання послуг онлайн.

Предметом дослідження є методи та засоби надання послуг онлайн у секторі доставки їжі.

Методи дослідження. У роботі використовуються методи дослідження, а саме аналіз, моделювання, класифікація, узагальнення, спостереження, прогнозування та експерименту; методи передачі даних та методі представлення результату.

Основний науково-технічний результат роботи полягає в застосуванні та удосконаленні методики проектування предметно орієнтованого програмного забезпечення на прикладі розробки онлайн сервісу доставки їжі, що надає можливість підвищити продуктивність проектування та розробки програмного забезпечення за рахунок покращення взаємодії команди розробників з експертами предметної галузі.

Практична цінність роботи полягає в удосконаленні методики проектування предметно орієнтованого програмного забезпечення та розробці онлайн сервісу для взаємодії кінцевого користувача з бізнесом в сфері доставки їжі.

Апробація результатів роботи. Результати даної роботи було представлено на L науково-технічній конференції факультету інтелектуальних інформаційних технологій та автоматизації Вінницького національного технічного університету на кафедрі автоматизації та інтелектуальних інформаційних технологій та опубліковані у вигляді тез доповіді [13].

1 СУЧАСНИЙ СТАН ПРОБЛЕМИ ТА ОСНОВНІ ЗАДАЧІ РОБОТИ

1.1 Огляд сектору онлайн-доставки їжі

1.1.1 Розмір сектору електронної комерції

За останнє десятиліття ринок електронної комерції зазнав значного зростання, оскільки клієнти все частіше переходять в Інтернет. Цю зміну в тому, як споживачі здійснюють покупки, спричинив широкий спектр різноманітних товарів деякі з них залежать від ринку чи країни, інші виникають у результаті світових змін. Ці зміни включають: збільшення прибутку від продажу, особливо в країнах, що розвиваються; більш тривала робота і час поїздок; збільшення проникнення широкосмугового зв'язку та підвищення безпеки електронних платежів; послаблення торгових бар'єрів; збільшення кількості роздрібних торговців, які присутні в Інтернеті; і підвищення обізнаності клієнтів про електронну комерцію [2].

Найбільше зростання електронної комерції за останні кілька років відбулося в Китаї, де в у 2019 році продажі склали 1,935 трильйона доларів США — сума, яка більш ніж у три рази перевищує витрати в Сполучених Штатах (586,92 мільярда доларів США), другому за величиною ринку. Сам по собі Китай становить 54,7% світового ринку електронної комерції, частка, яка майже вдвічі перевищує ринкову частку п’яти більших країн світу (США, Великобританія, Японія, Південна Корея, Німеччина) разом [1].

Тим часом в Україні близько 67 % інтернет-користувачів відвідують сайти, пов’язані зі сферою електронної комерції. Найбільш динамічною частиною, що розвивається, є здійснення саме електронної торгівлі (таблиця 1.1.) [3].

Показник	2012	2013	2014	2015	2016	2017
Обсяги роздрібної торгівлі, млрд. дол.	99,5	111,0	76,0	47,4	45,4	43,5
Обсяги інтернет-торгівлі, млрд дол.	0,57	0,88	1,04	1,17	1,50	1,70
Річний індекс зростання обсягу роздрібної торгівлі, %	20,3	9,4	1,5	14,4	12,4	6,0
Річний індекс зростання інтернет-торгівлі, %	46,8	53,3	75,2	107,6	50,4	25,0
Проникнення інтернет-торгівлі в Україні, %	0,6	0,8	1,4	2,5	3,3	3,9

Таблиця 1.1 – Динаміка показників розвитку електронної торгівлі в Україні

1.1.2 Від інтернету до офлайн-бізнесу та онлайн доставка їжі

Швидке зростання електронної комерції породило багато нових форм бізнесу, таких як B2B (бізнес до бізнесу), C2C (від клієнта до клієнта), B2C (від бізнесу до клієнта) та O2O (онлайн до офлайн).

Бізнес O2O — це метод маркетингу, заснований на інформаційно-комунікаційних технологіях (ІКТ), за допомогою якого споживачі розміщують замовлення на товари чи послуги онлайн і отримують товари чи послуги за адресою офлайн [4].

Однією з важливих подій, що сприяли вибуху комерції O2O, стало поширення смартфонів і планшетів і розвиток інфраструктури для підтримки платежів і доставки. У 2019 році було 5,2 мільярда підключень смартфонів, а до кінця 2020 року прогнозується, що половина людей у світі матиме доступ до послуг мобільного Інтернету.

Послуги O2O з’явилися в різних сферах, включаючи придбання різноманітних категорій продуктів і послуг, таких як їжа, готельні номери,

нерухомість або оренда автомобілів. Доставка їжі онлайн стосується процесу, за допомогою якого їжа, замовлена в Інтернеті, готується та доставляється споживачеві. Розвиток онлайн доставки їжі був підкріплений розробкою інтегрованих онлайн-платформ, таких як Uber Eats, Bolt Food, Glovo та Rocket. Онлайн-платформи доставки їжі виконують різноманітні функції, включаючи надання споживачам широкого вибору їжі, приймання замовлень і передачу цих замовлень виробнику продуктів харчування, моніторинг платежів, організацію доставки їжі та забезпечення засобів відстеження який зображений на рисунку 1.1.

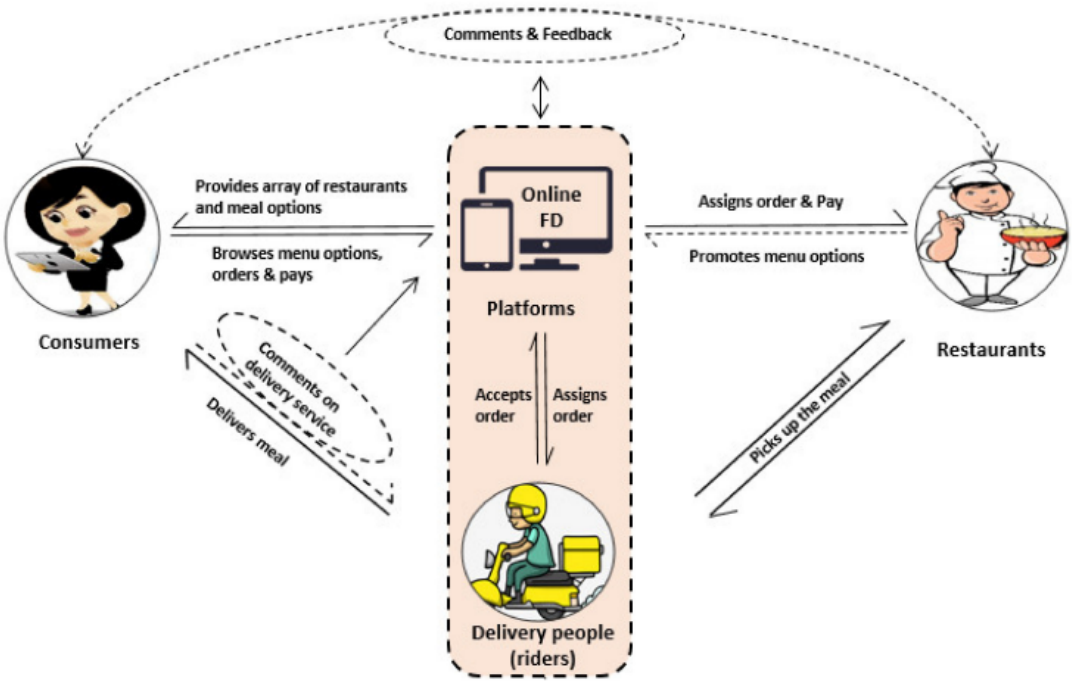

Рисунок 1.1 – Функції, пов’язані з онлайн-платформами доставки їжі

1.1.3 Постачальники та їх системи доставки

Постачальників доставки їжі можна класифікувати як такі, що здійснюють доставку з ресторану до споживача (Restaurant-to-Consumer) або з платформи до споживача (Platform-to-Consumer). Постачальники послуг доставки від

ресторану до споживача роблять їжу та доставляють її, як це характерно для таких постачальників, як KFC, McDonald’s і Domino’s. Замовлення можна зробити безпосередньо через онлайн-платформу ресторану або через сторонню платформу. Ці платформи сторонніх розробників відрізняються в різних країнах і включають такі приклади, як Uber Eats у США, Eleme у Китаї, Just Eat у Великобританії та Bolt Food в Україні. Платформи третіх сторін також надають онлайн послуги доставки від ресторанів-партнерів, які, натомість, не обов’язково пропонують послуги доставки; цей процес визначається як доставка від платформи до споживача [2].

Онлайн доставка їжі вимагає високоефективних і масштабованих служб доставки в реальному часі. Ресторани можуть використовувати наявний персонал для самостійної доставки, наприклад, офіціантів у деяких невеликих ресторанах, або вони можуть використовувати спеціалізовані команди доставки, які спеціально найняті та навчені для цієї ролі, як це спостерігається у деяких великих брендів ресторанів, таких як KFC, Domino's і Xibei. Крім того, ресторани можуть використовувати краудсорсингову логістику, мережу кур’єрів, які є незалежними підрядниками, така модель яка забезпечує ефективний, недорогий підхід до доставки їжі. Онлайн-платформи доставки їжі можуть або відповідати за набір і навчання професійних кур’єрів, або вони також можуть вдаватися до логістики краудсорсингу, використовуючи кур’єрів, які не обов’язково працюють на онлайн-платформі доставки їжі. Професійні кур’єри зазвичай проходять навчання, і принаймні частина їхньої зарплати гарантована, а частина – на основі комісійних. На відміну від цього, незалежні кур’єри отримують оплату на основі комісії (за замовлення) (рисунок 1.2) [6].

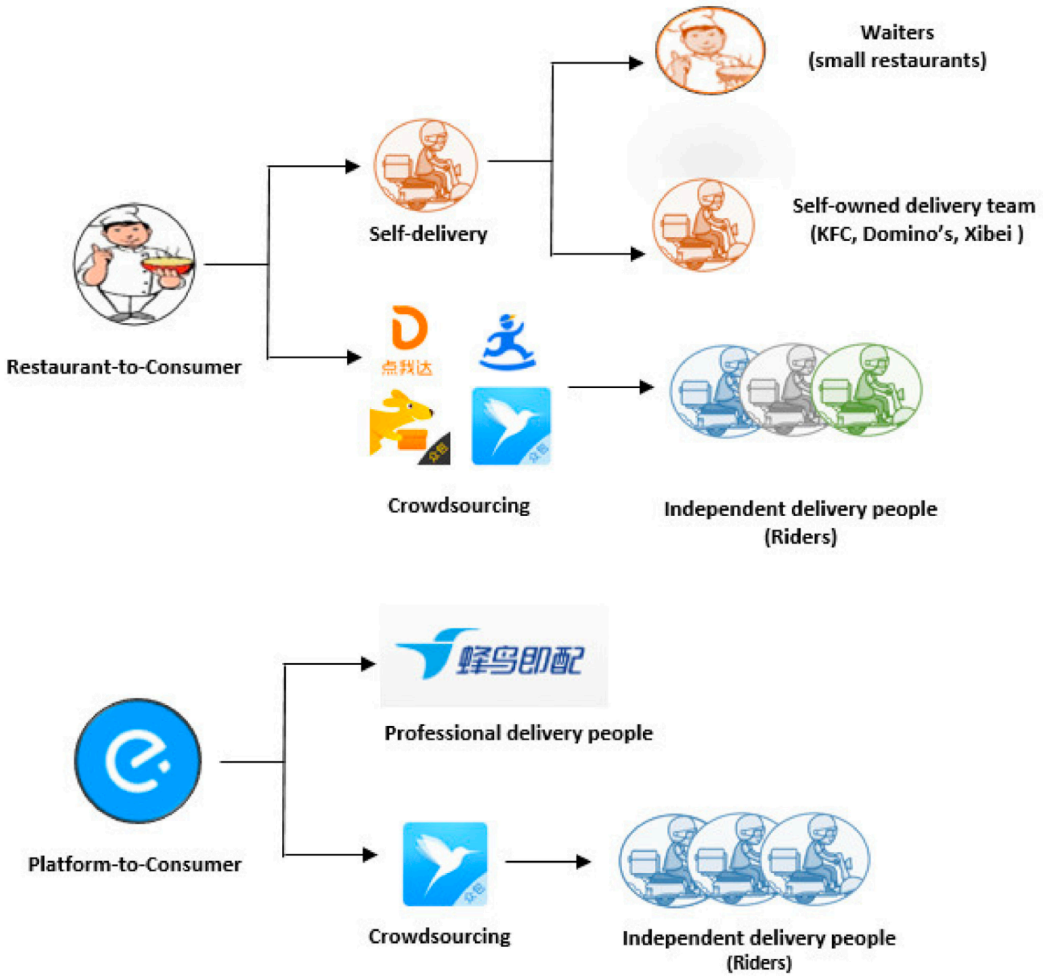

Рисунок 1.1 – Функції, пов'язані з онлайн-платформами доставки їжі

1.2 Концепція предметно-орієнтованого проектування

Предметно-орієнтоване проектування (Domaindriven design, DDD) це підхід до розробки програмного забезпечення, який зосереджує розробку на [21] програмуванні моделі предметної області, що має глибоке розуміння процесів та правил домену. Назва походить від книги Еріка Еванса 2003 року,А яка описує підхід через каталог шаблонів [7]. З тих пір спільнота практиків розвивала ідеї, породжуючи різні інші книги та навчальні курси. Підхід особливо підходить для складних доменів, де потрібно організувати багато часто безладної логіки.

Основні сфери, де DDD може знадобитись:

– розуміння предметної області;

– співпраця експертів предметної області з розробниками;

– проектування систем;

– рефакторинг.

1.2.1 Домени

Домен це, в основному, те, що робить організація. Це сфера діяльності компанії, і програмне забезпечення призначене для вирішення проблем у цій галузі [8]. Домен може бути складений на менші частини, субдомени.

Управління доменами визначає диференціацію трьох типів субдоменів:

– основний домен;

– підтримуючий домен;

– загальний домен.

Основний домен найважливіша частина; це серце бізнесу. Це те, що робить гроші, і на що найбільше зосереджений бізнес, і це має головне значення для успіху організації [8]. Основний домен є субдоменом в якому потрібно найбільше застосовувати предметноорієнтоване проектування. Важливо, щоб основному домену приділяли найбільшу увагу, і щоб він був змодельований, спроектований та розроблений якнайкраще. Цього слід досягти за рахунок більшого залучення експертів з предметної області та участі найкращих розробників у команді [24].

Підтримуючі домени це домени, які не є головним напрямком організації, але вони допомагають підтримувати основні домени.

Загальний домен є найменш важливим доменом для організації. Це субдомен, який існує у багатьох системах. Загальний домен не вимагає особливої уваги, і в ідеальному випадку існуючий продукт може бути використаний для економії часу, який можна скоріше вкласти в основний домен.

1.2.2 Розуміння предметної області

Предметно-орієнтоване проектування ставить домен в центр кожного етапу розробки програмного забезпечення. Щоб розробити корисну систему, розробники повинні розуміти домен, процеси, що відбуваються в певних ситуаціях і як це досягається. Людей, які надають знання про домен, називають експертами предметної області.

Експерти та розробники доменів повинні часто взаємодіяти протягом всього процесу розробки. Хоча ця взаємодія може зайняти багато часу як для експертів доменів, так і для розробників, вона в довгостроковій перспективі окупиться. Ідеальна ситуація, коли експерт з доменів може бути постійною частиною команди розробників. Важливою частиною є пряма та постійна взаємодія між експертами домену та розробниками, обробка знань на основі традиційної моделі розробки програмного забезпечення водоспаду, в якій аналітик доменів на етапі вимог інженерії отримуватиме знання від експертів, а потім передаватиме їх розробникам, швидше за все, не призведе до найкращого результати. Це пояснюється тим, що розробники обмежуються знаннями, які отримав аналітик доменів і які він визнав корисними. Крім того, якщо розробникам потрібна додаткова інформація або пояснення, вони або не можуть 17 отримати її від експерта, оскільки це було зроблено на першому етапі розробки, або вони можуть робити це дуже рідко, щоб не турбувати експертів домену.

Щоб полегшити обмін знаннями, розробники та експерти доменів повинні мати спільну мову, яку Ерік Еванс визначає як загальна мова [7]. Загальна мова є результатом поєднання знань розробників та експертів предметної області. Експерти з доменів повинні надавати правильні терміни для різних ситуацій. Усі терміни повинні мати точне значення, і не повинно бути двозначності. Це одна з причин, чому надмірно використовувані слова, такі як менеджер, контролер або служба, як правило, не є добрими іменами. Загальна мова повинна використовуватися в усіх аспектах розробки програмного забезпечення

протягом усього процесу розробки. Загальна мова повинна використовуватися в коді з тими самими термінами та поняттями, що використовуються як імена класів, властивостей, назв методів, тощо [24].

1.2.3 Модель домену

Отримані знання про домен зображені в моделі домену. Модель домену являє собою уявлення про домен, розроблену з урахуванням потреб випадків використання бізнесу [24]. Модель домену описана за допомогою загальної мови і працює як зв’язок між експертами предметної області та розробниками, які пов’язані між собою використовуваною мовою. Модель домену не є діаграмою (хоча її можна зобразити як таку), це ідея, що діаграма передбачається передати [7]. Не важливо, щоб модель домену була досконалою, їй не потрібно повністю відображати реальність. Мета моделі домену бути наближеною до реальності, але лише з ділової точки зору, і вона повинна відображати те, що має значення для бізнесу.

Щоб зробити модель домену найбільш корисною, необхідно синхронізувати її з фактичним кодом. Модель домену, яка не відображається в коді, 18 може стати неактуальною або навіть ввести в оману. Тому було створено модельований підхід до проектування, який виступає за тісно пов’язану модель домену та код. Коли в коді відбуваються серйозні структурні зміни, наприклад, в результаті постійного поєднання знань, модель домену слід оновити, щоб відобразити зміни [7].

1.2.4 Обмежений контекст

Обмежений контекст це мовна межа навколо доменної моделі [8]. Усередині обмеженого контексту поняття моделі, як властивості та операції, мають особливе значення, і загальна мова використовується для опису моделі. Один термін в одному обмеженому контексті повинен мати одне точне значення. Однак той самий термін можна використовувати в різному обмеженому контексті, щоб описати щось інше.

Обмежений контекст дуже корисний у великих доменах з багатим словниковим запасом. У цих сферах може бути дуже важко встановити, що всі терміни мають глобальне, точне, єдине і чітке значення.

В ідеалі, одна команда розробників повинна відповідати за один обмежений контекст. Таким чином, цілісність обмеженого контексту буде краще захищена оскільки менша кількість людей буде працювати над одним обмеженим контекстом, їм буде простіше домовитись про один і той же словниковий запас, використовувати одну і ту ж загальну мову і не витікати терміни в інші обмежені контексти. Важливо, що команди формуються навколо створених обмежених контекстів, а не що обмежені контексти створюються на основі існуючої структури команди. Пізніше це змусило б створити необмежений контекст, який не відповідає своїй меті, виступати як лінгвістична межа, вони були б змушені бути більшими або меншими залежно від розміру команди, що може призвести до неприродньо зміщеної контекстної межі

1.2.4 Карта контексту

Карта контексту – це огляд на обмежені контексти та їхні зв’язки. Це діаграма високого рівня, яка допомагає візуалізувати кордони обмежених контекстів, які контексти пов’язані і як. Карта контексту має бути достатньо простою, щоб її зрозуміли розробники та експерти предметної області, і вона має

відображати поточну реальність. Використовуючи контекстні карти, розробники отримують кращу картину всієї системи, що захищає цілісність кожного обмеженого контексту.

Існує кілька закономірностей, які описують зв’язки між обмеженими контекстами:

– *Спільне ядро*. Спільне ядро — це частина моделі, яка спільно використовується кількома окремими обмеженими контекстами. Спільне ядро корисне, коли є обмежені контексти, які мають багато спільних концепцій предметної області, логіки і використання карт перекладу для перекладу з одного контексту в інший було б занадто складно. Через існування спільної залежності, обидві команди повинні знати та бути обережними з цим. Тому слід ретельно обдумати, чи використовувати цей шаблон.

– *Клієнт-постачальник*. Коли два обмежені контексти перебувають у відносинах «зверху-вниз», це означає, що контекст на нижньому кінці відношення залежить від верхнього. Зміни верхньої частини, ймовірно, також вплинуть на нижню частину. Клієнт-постачальник є більш колаборативним підходом до відношення «зверху-вниз», де команди з обох обмежених контекстів співпрацюють разом домовившись про інтерфейс, який задовольнить їх обох.

– *Конформіст*. Конформіст - це також відношення «зверху-вниз», але на відміну від клієнта-постачальника, нижній контекст не може очікувати змін від верхнього контексту, він має відповідати тому що надає верхній контекст. Найпоширенішим прикладом цього зв'язку є залежність від зовнішнього постачальника. У цьому сценарії нижній контекст не може очікувати, що зовнішній постачальник змінить свій API для одного клієнта.

– *Антикорупційний рівень*. Антикорупційний рівень корисний, коли дві моделі занадто складні для легкої інтеграції, а тісна інтеграція може поставити під загрозу цілісність моделі в обмеженому контексті. Особливо це може статися під час інтеграції старих, успадкованих або зовнішніх контекстів. Щоб уникнути залежності від поганого коду, слід використовувати рівень захисту від корупції, який служить рівнем перекладу між моделями обох контекстів. Таким чином,

модель у «хорошому» контексті залежатиме лише від антикорупційного рівня, який також є частиною її контексту.

– *Окремі шляхи*. Окремі шляхи — це прагматичний шаблон, який виступає за те, щоб узагалі не інтегрувати обмежений контекст, якщо це не потрібно. Інтеграція коштує дорого, а переваги іноді невеликі.

– *Служба відкритого хосту.* Коли кілька контекстів створюють антикорупційні рівні для перекладу складної моделі того самого обмеженого контексту, часто це може бути занадто багато непотрібної, повторюваної роботи. Натомість складна модель може чітко визначити свій контракт, відомий як служба відкритого хосту, і інші контексти безпосередньо використовуватимуть цей контракт.

– *Опублікована мова*. При перекладі з однієї моделі обмеженого контексту на іншу, необхідно, щоб вони використовували спільну мову. Використовувана мова називається опублікованою мовою і повинна бути добре задокументована. Опубліковану мову часто поєднують з службою відкритого хосту.

1.4 Висновки

В першому розділі розглянуто основні аспекти роботи: проведено огляд онлайн сектору доставки їжі, представлено основні поняття по темі, наведено переваги використання предметно- орієнтованого проектування.

2 ДОСЛІДЖЕННЯ МЕТОДОЛОГІЙ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Огляд існуючих практик проектування та розробки програмного забезпечення

При проектуванні та розробці програмного забезпечення команда розробників залучається до проекту, основною метою проектування та розробки є постачання якісного продукту. Якість означає повне дотримання вимог, відсутність помилок, високий рівень безпеки та здатність витримувати великі навантаження [14].

Додаток або вебсайт також повинні додавати цінності клієнтам, працю вати за призначенням та забезпечувати інтуїтивно зрозумілий інтерфейс, щоб ним можна було користуватися навіть не думаючи. Не дивлячись на те, що все це досить складно, для спрощення та вдосконалення розробки програмних продуктів з’явилися найкращі практики розробки програмного забезпечення [15].

2.1.1 Керована тестами розробка

Керована тестами розробка (Testdriven development, TDD) це підхід до розробки програмного забезпечення, в якому розробляються тестові кейси, які визначають необхідні покращення або нові функції. Якщо говорити простими словами, спочатку створюються і перевіряються тестові кейси для кожної функціональності, а якщо пройти тест не вдається, то для проходження тесту пишеться новий код [16].

Основними перевагами керованою тестами розробки є:

– поліпшення якості шляхом виправлення помилок якнайшвидше під час розробки;

– значне підвищення якості коду;

– покращення розуміння коду оскільки рефакторинг вимагає регулярного вдосконалення;

– покращення швидкості розробки, оскільки розробникам не потрібно витрачати час на відлагодження програми.

Принцип роботи керованою тестами розробки зображений на рисунку 2.1.

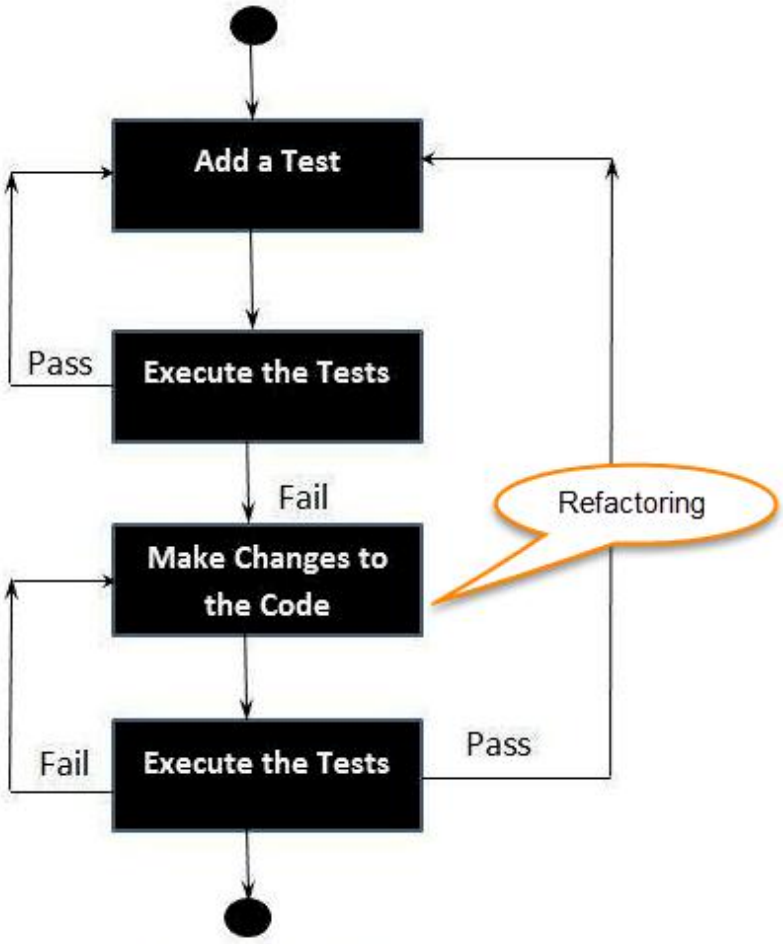

Рисунок 2.1 – Цикл керованою тестами розробки

Відповідно до досліджень недоліками використання підходу є:

- неможливість гарантувати відсутність помилок у програмі, навіть за наявності широкого спектру тестових кейсів;

- велика витрата часу на розробку тестових кейсів та підтримку належних наборів тестів [17].

2.1.2 Керована поведінкою розробка

Керована поведінкою розробка (Behaviordriven development, BDD) це синтез та вдосконалення практик, що випливають з керованої тестами розробки (TDD) та керованою тестами розробки прийняття (Acceptance test–driven development, ATDD) [18]. BDD доповнює TDD та ATDD за допомогою наступних технік:

– Мислення «ззовні всередину», іншими словами, застосовувати лише ті способи поведінки, які найбільше сприяють цим результатам бізнесу, щоб мінімізувати витрати.

– Описування поведінки в одній нотації, яка є безпосередньо доступною для експертів області, тестувальників та розробників, з метою покращення комунікації.

– Застосування цих методів аж до найнижчих рівнів абстрагування програмного забезпечення, приділяючи особливу увагу розподілу поведінки, щоб прогресування залишалося дешевим.

Команди, які вже використовують TDD або ATDD, можуть захотіти роз глянути BDD саме з таких причин:

– BDD пропонує більш точні вказівки щодо організації бесіди між розробниками, тестувальниками та експертами предметної області.

– Інструменти, орієнтовані на підхід BDD, як правило, дозволяють автоматично створювати технічну документацію та документацію для кінцевих користувачів із “специфікацій” BDD.

Недоліком даного підходу є необхідність представити команду розробників для роботи з клієнтом. Короткий час реакції, необхідний для процесу, означає високий рівень доступності. Однак, якщо клієнт добре розуміє,

що задіяно у проекті розробки, заснованому на принципах Agile, експертклієнт буде доступний у разі потреби. І якщо команди розробників працюють максимально ефективно, їх вимоги до експертаклієнта будуть мінімізовані [19].

2.1.3 Неперервна інтеграція

Неперервна інтеграція (Continuous Integration, CI) це практика розробки програмного забезпечення, при якій зміни кодової бази є інтегрованими в сховища потоків після побудови та перевірки за допомогою автоматизованого робочого процесу [20]. Принцип роботи неперервної інтеграції зображений на рисунку 2.2.

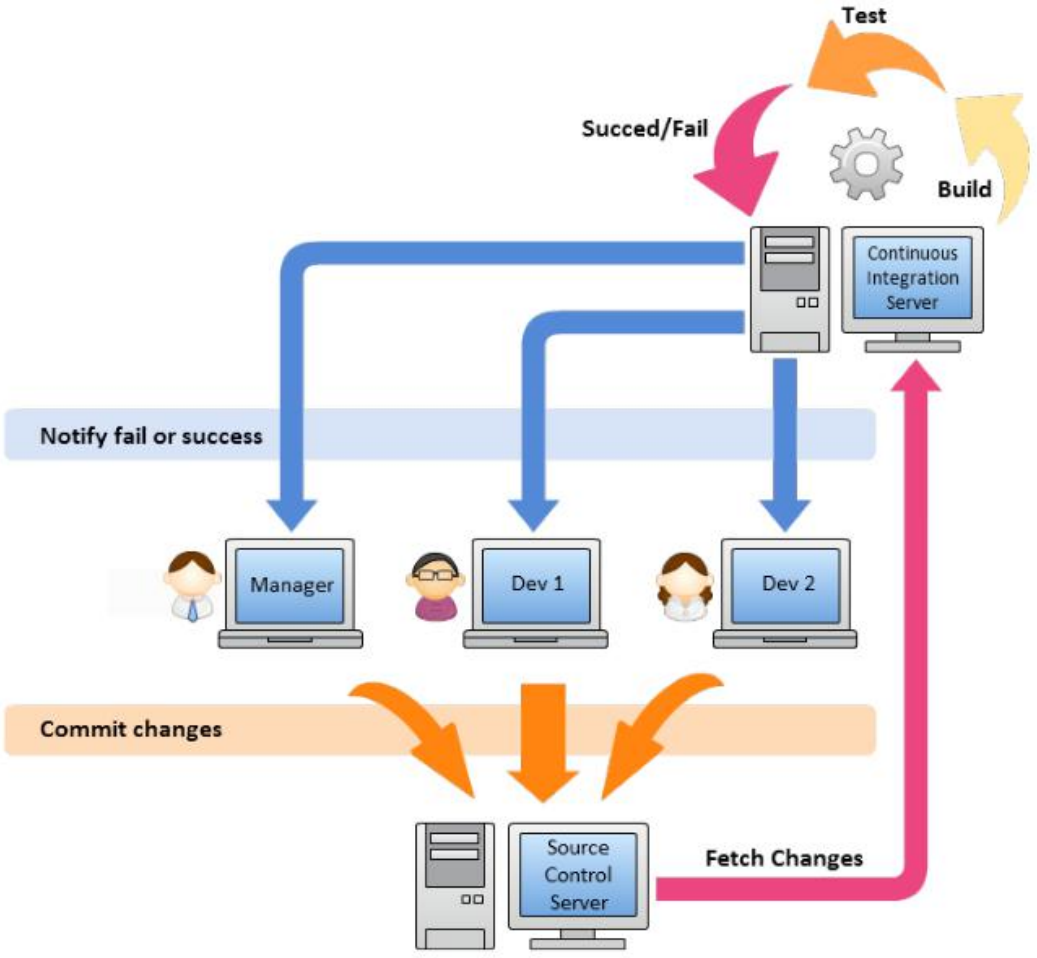

Рисунок 2.2 – Цикл роботи неперервної інтеграції

Основні переваги використання неперервної інтеграції:

– середній час до роздільної здатності (Mean time to resolution, MTTR) швидший і коротший;

– ізоляція несправностей менша і швидша;

– підвищений рівень випуску допомагає швидше виявляти та виправляти несправності;

– автоматизація в CI зменшує кількість помилок, які можуть виникнути на багатьох етапах.

Недоліки використання неперервної інтеграції:

– кодова база повинна бути готова і негайно впроваджена у виробництво, як тільки поточний результат буде успішним;

– підхід вимагає суворої дисципліни з боку учасників. Невдачі у дотриманні процесів незмінно породжуватимуть помилки, витрачаючи час і гроші.

– деякі галузеві середовища не підходять для неперервної інтеграції.

Медична сфера та авіація вимагають багато випробувань, щоб включити код у загальну систему [21].

2.1.4 Безперервна доставка

Безперервна доставка (Continuous delivery, CD) - це підхід до програмної інженерії, при якому команди продовжують виробляти цінне програмне забезпечення за короткі цикли та забезпечують надійний випуск програмного забезпечення в будь-який час [22]. Модель безперервної доставки зображена на рисунку 2.3

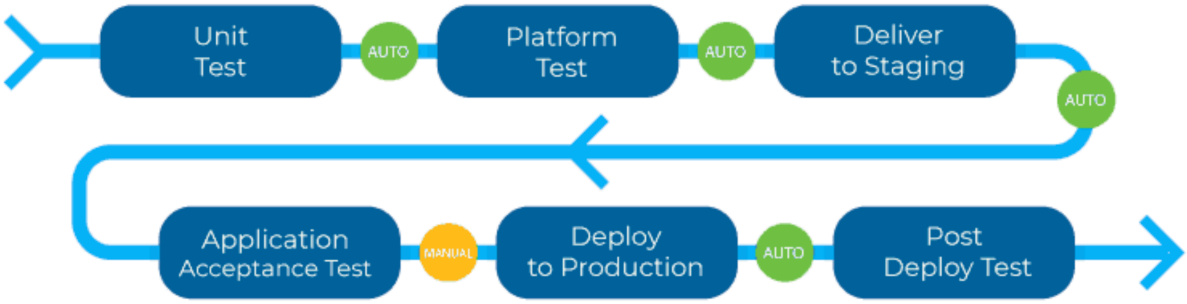

Рисунок 2.3 – Модель безперервної доставки

Переваги використання безперервної доставки:

– Прискорений час виходу на ринок. Час циклу від задуму історій користувачів до виробництва зменшується з кількох місяців до двох-п’яти днів. Частота вивільнення зростає від одного разу на один до шести місяців, до одного разу на тиждень [22].

– Створення правильного продукту. Часті випуски дозволяють командам розробників додатків швидше отримувати відгуки користувачів. Це дозволяє їм працювати лише над корисними функціями. Якщо вони виявляють, що якась функція не є корисною, вони не витрачають на неї подальших зусиль. Це допомагає їм створити правильний продукт.

– Надійні релізи. Зникає високий рівень стресу та невизначеності, пов’язаний з випуском продукту.

Недоліки використання безперервної доставки:

– Вартість переходу. Реалізація постійної доставки вимагає великих зусиль, часу та грошей. Зміна робочого циклу, автоматизація процесу тестування та розміщення ваших сховищ у Git - це лише деякі процеси, з якими вам доведеться впоратись.

– Важкість в обслуговуванні. Практики безперервної доставки потребують постійної підтримки. Це може бути важко для великих фінансових організацій, які пропонують різноманітні послуги. Такі компанії матимуть не один, а кілька

трубопроводів, деякі з яких можуть навіть закінчуватися на різних етапах постачання. Це ускладнює порівняння пропускної здатності та часу циклу [23].

2.2 Аналіз архітектурних шаблонів вищого рівня

У цьому розділі описано архітектурні шаблони, які можна використовувати на найвищому рівні системи. Використання цих методів розглядається лише у випадку наявності двох або більше обмежених контекстів, оскільки для простої системи лише з одним обмеженим контекстом ці ідеї не потрібні.

2.2.1 Монолітна архітектура

Монолітний додаток — це вид програмного забезпечення, який використовує єдину кодову базу для обслуговування багатьох різних сервісів і різних інтерфейсів, таких як REST (передача репрезентативного стану), API та сторінки HTML. Монолітний підхід вважається стандартним способом початку розробки програм. Єдина кодова база спрощує розробку, розгортання та масштабування додатка доти розмір кодової бази відносно невеликий. Монолітна кодова база є гарним вибором на початку проекту через вищезазначені якості, тому що відсутній розподіл у коді, який би додав складності у розробці.

Монолітна додаток зазвичай використовує одну єдину базу даних для обробки всіх даних. Базу даних можна масштабувати до різних розділів за допомогою сегментування, але все одно розділи використовують однакову схему. З єдиною базою даних транзакції зазвичай легко обробляти, оскільки більшість систем баз даних забезпечують набір властивостей ACID (атомарність, узгодженість, ізольованість, довговічність). Розробники можуть легко визначати

транзакції та зосередитися на наданні нових функцій кінцевим користувачам. Єдина база даних має свої обмеження. Монолітна програма може мати кілька різних типів даних. Деякі з даних можуть бути більш придатними для зберігання в базі даних NoSQL, а деякі з даних – у реляційній базі даних. Однак за монолітного підходу розробникам зазвичай доводиться вибирати лише одну систему баз даних і використовувати її для всіх видів даних.

Більшість програм можуть бути досить простими на початку, але як додаток зростає, зростає і його складність. Типовий спосіб обробки складності програми який має монолітну архітектуру, це розділити програму на різні рівні. Багаторівневий підхід широко використовується в мережевих і операційних системах. Багаторівнева монолітна архітектура, показана на рисунку 2.4, дуже відома серед розробників. Програма розділена на рівень інтерфейсу користувача, рівень обслуговування та на рівень доступу до даних. Рівень доступу до даних зазвичай отримує доступ до однієї бази даних, яка обробляє всі дані які пов’язані з монолітною програмою.

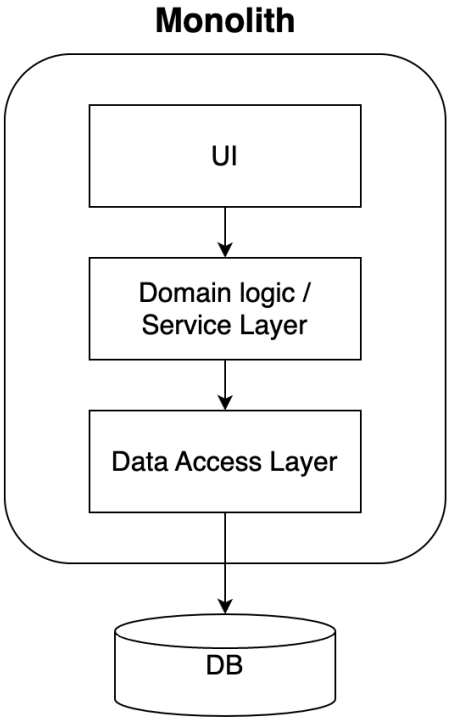

Рисунок 2.4 – Типовий монолітний додаток з багаторівневою архітектурою

Такий монолітний підхід робить процеси розробки, розгортання та масштабування легкими, якщо розмір програми відносно малий. Розробка програми є простою, тому що архітектура добре відома серед розробників, і вони знають як орієнтуватись в кодовій базі.

Розгортання окремого артефакту, наприклад одного екземпляра Node.js, у середовищі тестування чи виробництва легко. Це означає, що автоматизувати розгортання просто. Завдяки монолітній кодовій базі розгортання завжди містить кожну частину програми. Результат полягає в тому, що коли змінюється лише один компонент, всю програму необхідно повторно розгорнути. Це робить процес безперервної розгортки складним. Оскільки кожна частина програми завжди повторно розгортається, організації зазвичай прагнуть мати довший цикл між випусками, що сповільнює ітераційну розробку.

Масштабування монолітної програми здійснюється шляхом додавання нових вузлів з тим самим артефактом. Хоча це робить масштабування дуже простим, це також зменшує параметри масштабування. Компоненти програми, які потребують більше ресурсів, повинні бути масштабовані разом з компонентами, які могли б заповнити своє робоче навантаження з меншими ресурсами. Це означає додаткові витрати для організації, оскільки вузли вимагають все більше ресурсів чим більше стає додаток.

Легка розробка, розгортання та масштабування, якщо кодова база невелика, робить монолітну архітектуру найкращим способом розпочати розробку нової програми. Нові функції легко додавати, а це означає, що час виходу на ринок дуже швидкий, і існує велика кількість існуючих фреймворків, які підтримують багаторівневу архітектуру.

Однак із зростанням розміру програми та організації зростають і різні рівні програми. Якщо не приділяти багато уваги архітектурі та якості кодової бази, дуже ймовірно, що якість різних рівнів погіршиться. Погіршення відбувається через вимоги бізнесу, які змушують розробників робити не оптимальні рішення. Ці неоптимальні рішення слід переробити, але часу на рефакторинг може не вистачати, що призводить до того, що короткострокові рішення стають

довгостроковими. Оскільки розмір кодової бази зростає, а якість кодової бази погіршується, стає важче додавати нові функції та змінювати старі функції, оскільки розробник має знайти правильне місце для застосування цих змін. Це призводить до уповільнення циклів розробки.

Цикл розробки нової функції може сповільнитися ще більше, оскільки зміни можуть вплинути на кілька місць. Вплив змін може бути важко зрозуміти. Розробник може подумати, що зміна невелика, але насправді вона може вплинути на кілька місць. Це призводить до ситуації, коли потрібне широке ручне тестування тому цикли регресійних тестів можуть стати довгими. Усе це додає та сповільнює процес випуску нових функцій.

Одним із недоліків великої монолітної програми є те, що для ознайомлення з великою кодовою базою потрібно багато часу. Новим розробникам потрібен час, щоб прискоритися, оскільки вони почуваються загубленими у великій кодовій базі та не можуть знайти правильне місце для застосування змін. Цього можна уникнути, зберігаючи модульність всередині шарів і за допомогою постійно рефакторингу кодової бази, щоб зберегти код чистим. Хороше тестове покриття також допомагає з рефакторингом. Однак, якщо тестове покриття відсутнє, розробники можуть боятися рефакторингу кодової бази, оскільки їхні зміни можуть вплинути на інші частини в коді, і ручне тестування всіх цих місць є великим завданням.

Чіткі модульні межі всередині монолітної кодової бази може бути важко досягти та підтримувати під час розробки. Мови програмування надають розробникам деякі інструменти для забезпечення модульності та слабкого зв’язку в монолітній кодовій базі. Наприклад, наразі в Node.js можна забезпечити певні межі за допомогою модулів або області видимості. Однак порушити ці межі легко, оскільки розробники можуть легко змінити видимість і таким чином порушити модульність програми. Оскільки в монолітній кодовій базі немає жорстких модульних меж, можливо, що модульність кодової бази зменшується.

Крім того, якість коду з часом може знизитися, оскільки розробникам може бути важко зрозуміти, як правильно внести зміни в кодову базу.

Зниження якості коду ускладнює написання комплексних тестів. Це порочне коло, яке призводить до застарілого коду, якщо розробники та організація не зрозуміють, що необхідний постійний рефакторинг, щоб зберегти кодову базу чистою. Однак із дисциплінованими розробниками та вмілими архітекторами програмного забезпечення можна мати монолітну програму з хорошою модульною структурою та хорошим тестовим покриттям.

Велика монолітна програма з хорошим тестовим покриттям також може зіткнутися з проблемами з неперервною інтеграцією. Час створення конвеєрів неперервної інтеграції може стати довшим, оскільки потрібно скомпілювати багато коду та виконати тисячі автоматизованих тестів. Коли кілька людей перевіряють код до монолітної кодової бази, це може призвести до ситуації, коли зламані збірки є проблемою когось іншого, і стає важче точно визначити проблемний комміт, який порушив збірку.

Навіть із швидким і надійним конвеєром неперервної інтеграції, створити безперервну доставку з монолітною програмою дуже складно й виконується дуже рідко. З монолітною кодовою базою зміни можуть впливати на кілька місць, що означає, що регресійне тестування має бути надмірним. Релізи потрібно узгоджувати з кількома командами, які можуть працювати в різних географічних регіонах.

Експериментувати з новими технологіями та вносити зміни в поточний стек технологій важко з монолітною кодовою базою. Існуючий стек технологій обмежує технологічний вибір, який можна зробити. Випробування нових мов програмування обмежено монолітною кодовою базою. Наприклад, якщо середовищем виконання є Node.js, вибір мови програмування обмежений лише JavaScript. Кожен вибір технології має бути ретельно продуманий, тому що, як тільки нову технологію буде представлено та адаптовано до кодової бази, змінити її на іншу може бути надзвичайно важко. Зроблені технологічні вибори мають бути дійсними протягом багатьох років, оскільки переписи усього коду коштують дорого.

2.2.2 Мікросервісна архітектура

Архітектура мікросервісів — це новий стиль архітектури, який набув популярності в останні кілька років. Мікросервіси — це невеликі сервіси, які зосереджені на одному бізнес-контексті. Основним правилом щодо розміру мікросервісу може бути те, що його можна переписати за два тижні. Наприклад, один мікросервіс може обробляти створення замовлень, а інший мікросервіс може потім обробляти створення рахунка, який стосується замовлення. На рисунку 2.5 показано цей приклад, де інтерфейс програми викликає два різні сервіси: сервіс замовлення та сервіс виставлення рахунків. Потім ці сервіси окремо обробляють створення замовлення та створення рахунка. Ці два сервіси мають окремі кодові бази, і якщо є потреба спілкуватися між ними, зв’язок здійснюється через API, які надають ці сервіси.

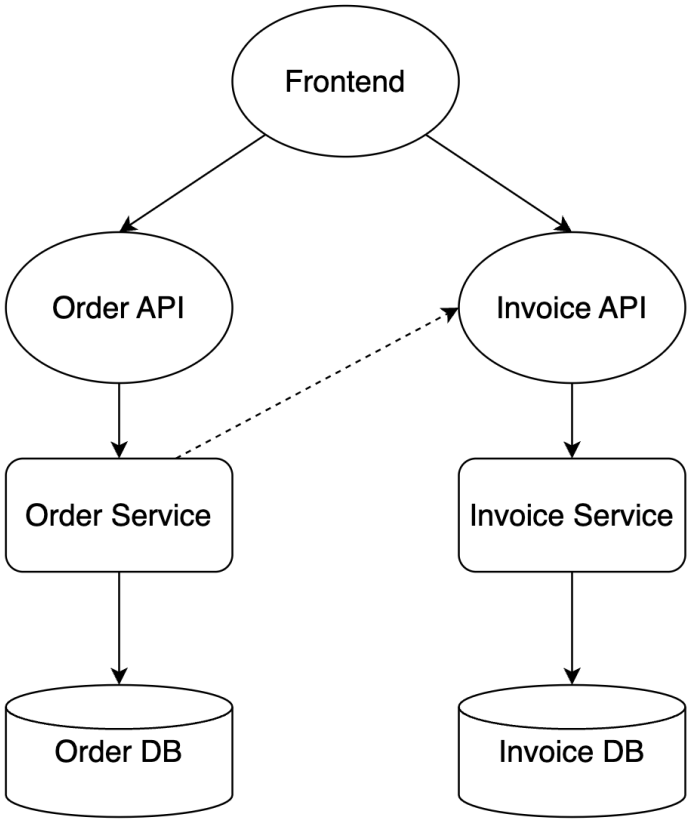

Рисунок 2.5 – Приклад додатку з мікросервісною архітектурою

Через невеликий розмір і зосередженість лише на одному бізнес-контексті мікросервіси дають змогу досягти гарної модульності кодової бази. Модульність означає проектування компонентів програми окремо та незалежно. Модульність полегшує розробку, оскільки робить зміни в одному модулі незалежними від інших. Модульність легко зберегти за допомогою мікросервісів, оскільки існують чіткі межі між кожним сервісом. Один мікросервіс може бачити лише інтерфейс інших мікросервісів, запобігаючи викликам внутрішніх методів інших мікросервісів. Це означає, що випадкове порушення модульності важче, а дотримання чіткої модульності не потребує дисципліни з боку розробників.

Чіткі модульні межі з REST інтерфейсом можуть спричинити проблеми з продуктивністю. Якщо для одного випадку використання потрібно виконати численні виклики сервісів, час виконання програми збільшується. Наприклад, якщо один виклик сервісу займає до 100 мс, то виклик 10 сервісів займає одну секунду. У цьому випадку кількість викликів між сервісами стала занадто великою, і, можливо, було б доцільно переглянути деталізацію сервісу. Одним із рішень є перепроектування та переоцінка деталізації сервісу шляхом об’єднання сервісів, але за збереженням принципів мікросервісного дизайну. Іншим рішенням є використання черг повідомлень і, якщо це можливо, асинхронне спілкування між сервісами.

Мікросервіси завжди повинні відповідати принципу єдиної відповідальності (SRP, Single Responsibility Principle). З архітектурою мікросервісів легко дотримуватися єдиної відповідальності, оскільки стиль архітектури заохочує створення невеликих модулів. Модульність також допомагає з єдиною відповідальністю, тому що коли межі чіткі, випадкове порушення єдиної відповідальності малоймовірне.

При розробці мікросервісів мета полягає в тому, щоб мати слабко зв’язані та високоузгоджені сервіси. Слабкий зв’язок означає, що сервіси не повинні знати нічого про внутрішню роботу інших сервісів. Це досягається за допомогою мікросервісів, оскільки вони мають чіткі межі за своєю природою та спілкуються лише через інтерфейси, які публікує кожен мікросервіс. Згуртованість можна

описати як щільність пов’язаних функцій у різних модулях. Якщо мікросервіси правильно відокремлені, вони дотримуються єдиної відповідальності і мають єдиний бізнес-контекст, у якому вони працюють. Якщо ці якості застосовуються до мікросервісів, це означає, що мікросервіси також мають велику зв’язність.

Щоб досягти слабкого зчеплення з мікросервісною архітектурою, може мати сенс дублювати частину коду. Системи з слабким зчепленням – це системи у яких компоненти слабко пов’язані (мають розривні зв’язки) один з одним, і, таким чином, зміни в одному компоненті системи менше впливають на роботу чи продуктивність іншого. Як правило, розробників навчають дотримуватися принципу DRY (Don’t Repeat Yourself). DRY стверджує, що той самий код не повинен повторюватися в кодовій базі, а замість цього код слід використовувати повторно. Це хороша порада в межах одного мікросервісу, але коли кілька мікросервісів використовують однаковий код, можуть виникнути проблеми. Якщо один сервіс потребує зміни спільного коду, це означає, що всі сервіси, які використовують ту саму спільну бібліотеку, також мають бути оновлені та розгорнуті. Це означає, що сервіси тепер тісно пов’язані. Ситуацію можна вирішити шляхом дублювання коду. Це дає можливість кожному із сервісів бути незалежними.

Слабке зчеплення, висока зв'язність, єдина відповідальність і модульність вважаються хорошими якостями архітектури програмного забезпечення. Ці якості та шаблони можна знайти в будь-якій добре спроектованій програмі, але з архітектурою мікросервісів більш імовірно, що ці шаблони та якості залишаться в кодовій базі під час еволюції програмного забезпечення. Як правило, коли розмір організації та розмір кодової бази зростає, стає важче зберегти ці якості в програмному забезпеченні. Архітектура мікросервісів має дві властивості, які дозволяють зберегти ці якості в кодовій базі протягом еволюції програмного забезпечення: чітке володіння кодом і мікросервіси мають менші кодові бази всередині сервісів. Право власності означає, що одна команда володіє мікросервісом. Невеликі кодові бази всередині сервісів полегшують роботу розробників.

Завдяки слабкому зчепленню, високій зв’язності, єдиній відповідальності і модульності цикли розробки можуть бути швидкими. Зміна функціональних можливостей і додавання нових можливостей усередині мікросервісу може бути відносно швидким, оскільки самі сервіси менші, і тому їх легше зрозуміти. Складна логіка сервісу змінюється до навколишньої комунікації. Завдяки простоті сервісів розробники можуть швидше розробляти нові функції, таким чином прискорюючи цикл розробки.

Архітектура мікросервісів вимагає неперервної інтеграції та включає безперервну доставку. Якщо існує сотні сервісів, перевірка кодів для кожного сервісу повинна бути автоматично підтверджена. Конвеєри неперервної інтеграції забезпечують валідність нових збірок. Кожна служба повинна мати власний конвеєр, щоб забезпечити швидкий зворотний зв’язок. Конвеєри неперервної інтеграції є досить швидкими, оскільки кожен із сервісів досить малий, і отже час, необхідний для запуску автоматичних тестів, не має бути надто великим. Архітектура мікросервісу забезпечує швидкий зворотній зв’язок для розробників у формі конвеєрів неперервної інтеграції. Оскільки над конкретним сервісом працює лише одна команда, буде зрозуміло чия це проблема, коли збірка ламається.

Швидкий конвеєр неперервної інтеграції і невеликі сервіси дозволяють включити безперервну доставку. Оскільки зміни містяться в службі, має бути зрозуміло, що змінилося, тому обсяг регресійного тестування залишається досить малим. Якщо всі автоматичні тести пройшли успішно, а охоплення тестами високе, можна з достатньою впевненістю здійснити розгортання безпосередньо у виробництві. Оскільки мікросервіси забезпечують швидке розгортання та відкат у випадках неправильного розгортання, проблемну службу можна відкотити до попередньої версії. Ці якості роблять безперервну доставку можливою.

2.2.3 Висновки

Як висновок, моноліт – це швидший і простіший спосіб почати розробку нових програм. Коли складність системи зростає – зростає і кодова база, саме тоді мікросервіси стають все більш привабливими. Можна зробити загальне зауваження, що мікросервіси не є хорошим вибором для невеликих організацій. Крім того, організації, які ще не добре розуміють бізнес-контекст, у якому вони працюють, не повинні починати з мікросервісів, тому що неправильно визначені межі сервісів можуть коштувати великих втрат у майбутньому.

Також існує варіант розпочати з монолітної архітектури, а потім перейти до мікросервісів, коли спрацюють фактори складності та розміру. Такі компанії, як LinkedIn, Netflix і Amazon пішли цим шляхом. Після того, як ці компанії отримали достатню популярність і їхній бізнес виріс до такого розміру, що монолітна архітектура не забезпечувала достатнього масштабування, вони здійснили трансформацію від моноліту до мікросервісів.

3 ПРОЕКТУВАННЯ МОДЕЛІ ПРЕДМЕТНО-ОРІЄНТОВАНОЇ СИСТЕМИ

Для того щоб розробити модель предметно-орієнтованої системи нам потрібно виконати наступні кроки:

– Розробити BPMN діаграму;

– Визначити історії користувачів;

– Визначити іменники в історіях користувачів;

– Визначити дієслова в історіях користувачів;

– Розробити діаграму об’єктної взаємодії;

– Розробити діаграму відповідальності об’єкта;

– Розробити UML діаграму классів.

– Розробити діаграму карти контексту.

3.1 Модель та нотація бізнес-процесів

Для проектування моделі домену та бізнес правил системи потрібне повне розуміння бізнес-процесів. Для цього нам допоможе складання BPMN діаграми.

Модель та нотація бізнес-процесів (BPMN, Business Process Model and Notation) – це система умовних позначень для моделювання бізнес-процесів. Розроблена Business Process Management Initiative (BPMI), що підтримується Object Management Group після їх злиття в 2005 році.

BPMN має кілька елементів, які дозволяють створювати моделі найрізноманітніших типів процесів. Навіть складні процеси можуть бути спочатку представлені за допомогою кількох елементів BPMN. З часом ці моделі можна постійно вдосконалювати. Серед елементів, які складають нотацію BPMN, можна знайти: пули, смуги, потоки, дії, підпроцеси, події, об’єкти даних, шлюзи тощо. Нижче наведено коротке пояснення основних елементів, які представлені на рисунку 3.1:

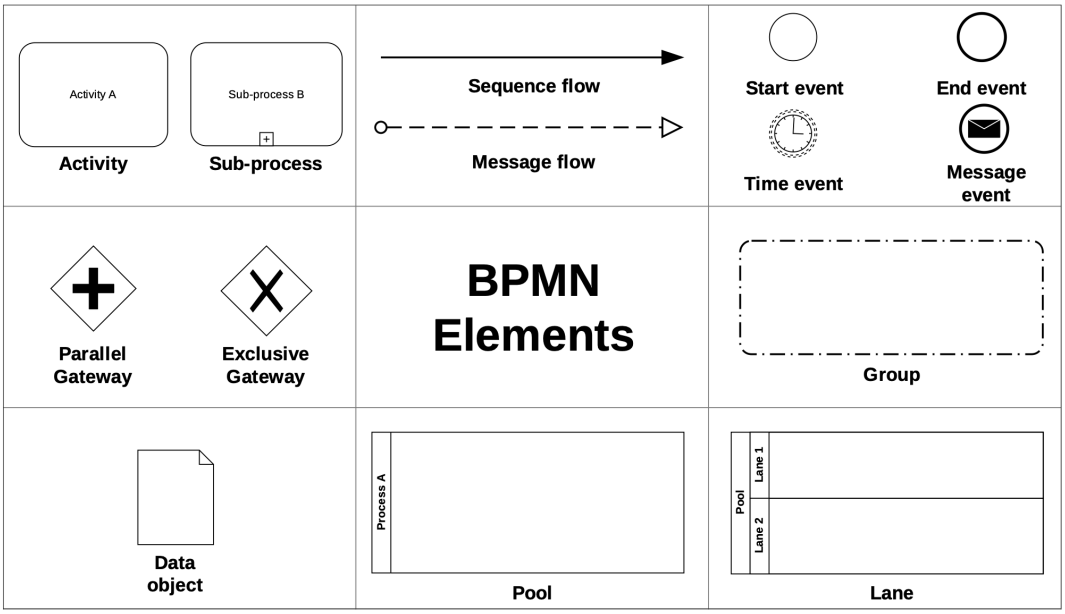

Рисунок 3.1 – Набір основних елементів, які визначені специфікацією BPMN

– **Події**: Представляють щось, що відбувається під час процесу. Події зазвичай мають причину (пусковий механізм) або наслідок. Представлені колом, існують три основні типи подій, а саме: *початкова подія*, *проміжна подія* та *кінцева подія*;

– **Діяльності**: Представляють роботу, яку потрібно виконати в рамках потоку процесу. Діяльність може бути завданням або підпроцесом (набором завдань);

– **Шлюзи**: Представляє розбіжність або конвергенцію послідовних потоків у процесі. Шлюз визначає розгалуження, злиття та приєднання шляхів.;

– **Послідовність потоків**: Демонструє порядок виконання дій у процесі;

– **Потік повідомлень**: Демонструє комунікацію між двома учасниками процесу (представленими двома окремими пулами на діаграмі співпраці);

– **Пул**: Представляє учасника співпраці. Він діє як контейнер, що розділяє набір дій з інших пулів;

– **Доріжки**: Являє собою підрозділ пулу та служить для організації та категоризації дій;

– **Повідомлення**: Відображає зміст комунікації між двома учасниками;

– **Об'єкти даних**: Представляють набір або сукупність інформації. Об’єкти даних можуть служити вхідними даними або створюватися в процесі діяльності;

– **Групи**: Представляють групування елементів в одній категорії. Групи зазвичай використовуються для документування та аналізу та не змінюють хід процесу.

Використовуючи умовні позначення та специфікації BPMN системи було розроблено модель бізнес-процесів бізнесу у сфері доставки їжі (рисунок В.1).

3.2 Історії користувачів

Отримавши явне представлення усіх бізнес процесів визначемо історї користувача нашого додатку.:

Історія користувача - це неформальне загальне пояснення функції програмного забезпечення, написане з точки зору кінцевого користувача. Його мета полягає в тому, щоб сформулювати, як функція програмного забезпечення забезпечить цінність для клієнта.

Історії користувачів полегшують розуміння та спілкування між розробниками і можуть допомогти командам документувати своє розуміння системи та її контексту.

Історії користувачів додатку:

– Як клієнт я хочу мати можливість додавати товари, які я хочу придбати, у кошик для покупок, щоб я міг швидко виконати замовлення пізніше;

– Як клієнт я хочу бачити список усіх товарів у наявності які я можу замовити;

– Як клієнт я хочу бачити загальну вартість кожного товару у кошику для покупок, щоб я міг повторно перевірити ціну на товари;

– Як клієнт я хочу мати можливість вказати адресу, куди будуть надсилатися моє замовлення;

– Як клієнт я хочу мати можливість додати примітку до адреси доставки, щоб я міг надати спеціальні інструкції кур’єру;

– Як клієнт я хочу мати різні способи оплати (готівкою, картою, онлайн), щоб я міг сплатити замовлення;

– Як клієнт я хочу отримати електронний лист із підтвердженням замовлення та номером замовлення, щоб у мене був доказ покупки;

– Як клієнт я хочу мати можливість перевірити статус замовлення;

– Як працівник бізнесу я хочу отримувати підтвердження оплати замовлення клієнтом;

– Як працівник бізнесу я хочу мати можливість змінювати статус замовлення;

– Як працівник бізнесу я хочу мати можливість отримувати деталі замовлення для підтвердження;

– Як кур’єр я хочу мати можливість отримувати повідомлення брати термінал чи ні;

– Як кур’єр я хочу мати можливість отримувати адресу для доставки замовлення;

– Як кур’єр я хочу мати можливість отримувати примітки до замовлення у випадку наявності спеціальних інструкцій до доставки.

Далі витягнемо іменники та дієслова із історій вище. Шукаємо іменники, які стануть головними об’єктами, а не атрибутами.

Іменники:

– Клієнт;

– Оператор;

– Кур’єр;

– Замовлення;

– Деталі доставки;

– Продукт;

– Кошик;

– Оплата;

– Доставка;

З наведених іменників було видалено дублікати, та застосовано більш офіційні імена, наприклад: Товар = Продукт, Працівник = Оператор, Адреса + Примітка замовлення = Деталі доставки.

Дієслова:

– Додати товар у кошик;

– Бачити список товарів;

– Бачити загальну ціну товарів;

– Сплатити замовлення;

– Обрати спосіб оплати;

– Вказати деталі замовлення;

– Отримати підтвердження замовлення;

– Перевірити статус замовлення;

– Отримати деталі замовлення для підтвердження;

– Отримати підтвердження оплати замовлення;

– Змінити статус замовлення;

– Отримати деталі замовлення;

3.3 Моделювання бізнес логіки

Бізнес-логіка - це система зв’язків та залежностей елементів бізнес-даних та правил обробки цих даних відповідно до особливостей ведення окремої діяльності (бізнес-правил), яка встановлюється при розробці програмного забезпечення, призначеного для автоматизації цієї діяльності. Бізнес логіка описує бізнес-правила реального світу, які визначають способи створення, представлення та зміни даних. Бізнес логіка контрастує з іншими частинами програми, які мають відношення до низького рівня: управління базою даних, відображення інтерфейсу користувача, інфраструктура і.т.д.

Використовуючи вказані в розділі 3.2 іменники та дієслова, ми можемо скласти діаграму об’єктної взаємодії (рисунок В.2).

Отримавши діаграму взаємодії об’єктів, ми можемо почати думати про діаграму відповідальності об’єкта. Однією з найпоширеніших помилок є покладання відповідальності на об’єкт актора, тобто учасника. Потрібно пам’ятати, що об’єкти повинні піклуватися про себе, а також повинні бути закриті для безпосередньої комунікації.

Тож давайте слідувати вищезазначеному підходу та розподіляти обов’язки. Розроблена діаграма відповідальності об’єкта зображена на рисунку В.3.

Після створення діаграми взаємодії об’єктів та відповідальності, потрібно перейти до складання UML діаграми класів. UML діаграма класів зображена на рисунку В.4.

Використовуючи отриману діаграму класів було проведено архітектурний аналіз та виявлено список доменів системи:

– Доставка їжі (основний домен);

– Фінансовий (підтримуючий домен);

– Сповіщення (загальний домен).

Також було виділено сукупність обмежених контекстів які будуть служити лінгвістичним кордоном компонентів системи:

– Замовлення;

– Доставка;

– Оплата;

– Сповіщення.

Відобразимо виявлені нами домени разом з обмеженими контекстами на діаграму класів (рисунок В.5).

Розробивши діаграму доменів та обмежених контекстів ми виявили що існує частина коду яка використовується декількома контекстами. У таких випадках потрібно використовувати кооперaційні шаблони до яких належить шаблон спільного ядра. Особливою деталлю шаблону спільного ядра є те, що він певним чином суперечить основному принципу обмежених контекстів: лише

одна команда може володіти обмеженим контекстом, тому потрібно враховувати розділення спільного ядра у майбутньому через збільшення складності моделі.

Складемо карту контекстів для огляду обмежених контекстів та їхньої взаємодії (рисунок В.6).

3.4 Висновки

У даному розділі була розроблена модель бізнес-процесів системи щоб отримати повне їх розуміння, та була спроектована модель бізнес-логіки яка стане основою для майбутньої розробки компонентів системи.

4 РОЗРОБКА КОМПОНЕНТІВ СИСТЕМИ

4.1 Налаштування середовища розробки

Перед розробкою ПЗ потрібно обрати ОС, середовище розробки, платформу та технологічний стек, що будуть використані під час розробки.

У якості платформи для розробки програмного забезпечення було обрано операційну систему на базі Linux. Linux — це операційна система (ОС) з відкритим кодом. Операційна система — це програмне забезпечення, яке безпосередньо керує апаратним забезпеченням і ресурсами системи такими як процесор, пам’ять і сховище даних. ОС знаходиться між програмами та апаратним забезпеченням і створює зв’язки між усім вашим програмним забезпеченням і фізичними ресурсами, які виконують роботу.

Переваги використання ОС Linux:

– *Відкритий вихідний код*. Однією з головних переваг Linux є те, що ця операційна система з відкритим вихідним кодом, тобто її вихідний код легко доступний для всіх. Будь-хто, хто вміє програмувати, може вносити, змінювати, покращувати та поширювати вихідний код з будь якою метою;

– *Безпека*. Linux є більш безпечною в порівнянні з іншими операційними системами, такими як Windows. Linux не є повністю безпечним, оскільки для нього також існує деяке шкідливе програмне забезпечення, але Linux менш вразливий, ніж інші. Кожна програма в Linux, програма або вірус, потребує авторизації адміністратора у вигляді пароля. Якщо пароль не введено, вірус не запуститься;

– *Оновлення програмного забезпечення*. У Linux ви стикаєтеся з більшою кількістю оновлень програмного забезпечення. Ці оновлення програмного забезпечення виконуються набагато швидше, ніж оновлення будь-якої іншої операційної системи. Оновлення в Linux можна виконувати легко, не стикаючись із серйозними проблемами чи занепокоєннями.

В якості платформи було вирішено вибрати платформу Node.js. Node.js — платформа з відкритим кодом побудована на основі libuv, мультиплатформенної бібліотеки для асинхронного вводу-виводу, та V8, інтерпретатора мови програмування JavaScript. Серед основних переваг Node.js: висока продуктивність, мова програмування та велика кількість готових модулів, за офіційними даними понад 700000. Сукупність цих переваг дозволяє швидко розробляти високопродуктивні додатки.

За основу було вирішено вибрати технологічний стек з відкритим кодом Metarhia, що включає в себе: сервер застосунків Impress, мережевий протокол JSTP, та бібліотеку Metascheme.

Impress — це сервер застосунків на платформі Node.js, заснований на таких ідеях: монолітна архітектура, простота коду прикладного рівня, високий рівень абстракції, підтримка різних моделей роботи з клієнтами: зі станом та без, підтримка плагінів, використання асинхронного лінивого вводу\виводу та його мінімізація, підтримка декількох протоколів для клієнт-серверної та міжсерверної взаємодії, можливість конфігурації горизонтального масштабування серверної інфраструктури, реалізація вбудованих спрощених систем логування та тестування, роутинг запитів, що базується на директоріях файлової системи, кешування та автоматичне оновлення при зміні вихідного коду для програмних реалізацій обробників запиті

4.2 Розробка компонентів бізнес логіки

Для перетворення моделі бізнес логіки у реальні компоненти допоможе використання тактичних шаблонів.

Тактичні шаблони - це шаблони, які допомагають управляти складністю моделі. Роль тактичних шаблонів полягає у захопленні та зображенні об’єктів, їхньої поведінки, значення, функціонування та взаємозв’язків між ними, єдиним способом. Даний шаблон говорить про те, як об’єкт з певною функцією та

характеристиками може бути реалізований найкращим чином для забезпечення читабельності, підтримки або розширюваності всієї моделі.

4.2.1 Сутність

Сутність - це об’єкт з атрибутами та функціями, унікальна ідентичність якої є важливою. Це означає, що навіть якщо деякі атрибути об’єкта змінюються, об’єкт все одно залишає свою однакову ідентичність. Сутності часто моделюються як змінні класи з унікальним ідентифікато- ром (який є незмінним). Коли ми порівнюємо рівність сутностей, порівнюємо рівність їх ідентифікаторів.

4.2.2 Об’єкт значення

Об’єкт значення - це об’єкт, який представлений його значенням. У об’єктів значення немає особистості, і якщо вони змінюються, вони більше не відображають того ж самого значення. Об’єкти значень особливо корисні, коли клас має більше атрибутів, а деякі з них діють як група, разом вони представляють одне значення. У цьому випадку атрибути слід перемістити до власного класу, який би діяв як одна одиниця.

Ще однією важливою особливістю об’єкта значення є змінність. Не має значення, який екземпляр класу буде використовуватися, коли всі атрибути однакові. Завдяки незмінності, також має бути можливим спільний доступ до одного і того ж екземпляру в кількох місцях, де ми вимагаємо одне і те ж значення. Об’єкти значень зазвичай моделюються як незмінні класи. Якщо нам потрібно змінити значення, краще просто створити новий екземпляр. Два об’єкти значення рівні, коли їхні атрибути рівні. З об’єктами вартості легше мати справу,

оскільки нам не потрібно гарантувати унікальність ідентичності. Це одна з причин, чому їм слід надавати перевагу над сутностями, якщо це можливо.

4.2.3 Сервіс

Сервіси - це об’єкти без стану, які забезпечують функціональність домену. Вони вводяться, коли існує більш складна ділова функціональність, яка не є безпосередньою відповідальністю жодного з існуючих об’єктів (сутності або об’єкт занчення), і зазвичай вимагає співпраці більшої кількості об’єктів. Доменними службами слід користуватися з обережністю, лише коли це необхідно, оскільки надмірне використання доменних сервісів може призвести до анемічної моделі домену де логіка всього домену знаходиться в сервісах замість сутностей або об’єктів значення.

4.2.4 Домена подія

Подія домену є новим шаблоном, ніж попередні. Ерік Еванс не говорить про них у своїй книзі, але це важлива концепція домену. Вони описують виникнення чогось, що трапилось. Події мають більше ситуацій, коли вони корисні - їх можна використовувати для запису змін, внесених до сукупності, або як інструмент комунікації між сукупностями в одному або навіть різному обмеженому контексті.

Події домену повинні моделюватися як незмінні об’єкти. Правильне використання загальної мови та правильне іменування є особливо важливим. Події слід називати у минулому часі на основі дії, що сталася.

4.2.5 Фабрика

Фабрика - це шаблон який відповідає за створення складних об’єктів та сукупностей. Це корисно в основному, коли сукупність складається з багатьох сутностей та об’єктів вартості, а формування нової сукупності вимагає більшої кількості кроків, під час яких сукупність не узгоджується. Фабрика інкапсулює логіку створення та виробляє повністю послідовну сукупність. Фабрика може бути реалізована як статичний метод або як окремий клас.

4.2.6 Сукупність

Сукупність - це група сутностей та об’єктів вартості, які разом утворюють кордон узгодженості транзакцій. Сукупність в цілому повинна бути узгодженою в будь-який момент часу. Отже, створюється корінь сукупності, яка служить точкою входу до сукупності, інші сутності та об’єкт значення вважаються внутрішніми для сукупності і не можуть бути доступні безпосередньо ззовні.

Слід бути особливо обережним при проектуванні сукупностей, оскільки неправильно створені межі сукупностей можуть спричинити проблеми. Завелика сукупність, як правило, погано працює, оскільки для забезпечення узгодженості, вносячи зміни до одного об’єкта сукупності, інші агрегати потрібно блокувати. Якщо об’єкт не мають багато спільного, це зайве. Як правило, при проектуванні сукупностей необхідно знати інваріанти моделі та проектувати межі сукупностей на основі них, а не на основі логічного групування.

4.2.7 Модуль

Модулі - це контейнери доменних об’єктів, і вони допомагають їх організувати та додатково розкласти модель домену. Модулі повинні бути спро-

ектовані, маючи на увазі правило низького зчеплення - правило високої згуртованості. Об’єкти в модулі повинні бути цілісними з іншими, вони повинні створювати одну логічну одиницю. З іншого боку, між різними модулями має бути низький зв’язок, об’єкти в одному модулі повинні мати якомога менше залежностей від об’єктів в інших модулях.

4.2.8 Репозиторій

Репозиторії використовуються для збереження сукупностей. Вони інкапсулюють логіку зберігання, отримання, оновлення та видалення сукупностей із певного сховища даних. Абстрагування від технічної реалізації сховища дозволяє створити модель, не думаючи про інфраструктурні проблеми.

З точки зору використання, існує два типи сховищ: орієнтований на колекцію та орієнтований на персистентність. Орієнтовані на колекції сховища діють як колекції в пам’яті, а це означає, що вони не мають жодного методу збереження чи оновлення. Це призводить до більш якісного коду, оскільки для досягнення модифікації достатньо завантажити сукупність, а потім змінити сукупність, не викликаючи жодного іншого методу в сховищі. Недоліком цього є те, що колекційно орієнтовані сховища важче реалізувати, оскільки основний механізм збереження повинен мати можливість відстежувати зміни об’єктів і в кінці транзакції відображати ці зміни в сховищі. І навпаки, сховища, орієнтовані на персистентність, діють більше як фізичне сховище, і вони надають методи збереження (або оновлення), що полегшує їх реалізацію.

4.3 Розробка моделі данних

У цьому розділі ми опишемо використання бібліотеки Metashema в нашому проекті. Metashema — це бібліотека з відкритим вихідним кодом, яка забезпечує потужний і гнучкий спосіб визначення та перевірки структур даних. Бібліотека Metaschema використовує спеціальну мову DSL (Domain specific language) що базуються на мові JavaScript для опису структур програми та взаємодій і відношень між ними. Даний опис в подальшому використовується для створення скриптів створення та оновлення баз даних, зокрема PostgresSQL.

Також Metaschema дозволяє описувати методи які клієнт зможе використовувати у своєму інтерфейсі. Кожен такий метод пов’язаний із набором даних які клієнт має надати, не обов'язковою у формі для збору цих даних (вона може бути створена автоматично з опису методу), а безпосередньо з функцією на мові JavaScript яка буде описувати бізнес-логіку.

Metaschema повністю бере на себе відповідальність за аналіз, перевірку та серіалізцію клієнтських даних відповідно до описаних схем та перевірок. Кожен такий метод має доступ до сесії клієнта (якщо вона присутня) та GlobalStorage з можливістю виконання довільних дій. База даних на момент виклику методу уже буде містити усі вказані структури, таблиці та дані, описані з допомогою Metaschema.

Схеми типів даних які надає бібліотека Metaschema:

– Примітивні: number, biginit, string, boolean, symbol;

– Структурні: object, function;

– Ціле число: int;

– Грошова одиниця: money;

– Дата: date, time, datetime;

– Геометрія: point, path, polygon;

– Текстовий: text, buffer;

Приклад розробленої моделі даних за допомогою мови DSL:

```
({
    Entity: {},

    price: { type: 'money' },
    account: 'Account',
    address: { type: 'string', length: 500 },
    deliveryType: { type: 'number', default: 0 },
    status: {
      enum: ['pending', 'accepted', 'delivering', 'completed', 'canceled']
    },
    orderDate: { type: 'datetime' },
    acceptDate: { type: 'datetime', required: false },
    deliveryStartDate: { type: 'datetime', required: false },
    completionDate: { type: 'datetime', required: false },
  });
```

Для того щоб перетворити зазначену вище модель даних у скрипт створення та оновлення бази даних потрібно в корні репозиторію виконати наступну CLI команду:

```
metasql c && cat ./application/schemas/database.sql > ./db/structure.sql
```

В результаті виконання якої отримаємо наступний результат:

```
CREATE TABLE "Order" (
  "orderId" bigint generated always as identity,
  "price" numeric(12, 2) NOT NULL,
  "accountId" bigint NOT NULL,
  "address" varchar(500) NOT NULL,
  "deliveryType" integer NOT NULL DEFAULT 0,
  "status" varchar NOT NULL,
  "orderDate" timestamp with time zone NOT NULL,
  "acceptDate" timestamp with time zone NULL,
  "deliveryStartDate" timestamp with time zone NULL,
  "completionDate" timestamp with time zone NULL
);

ALTER TABLE "Order" ADD CONSTRAINT "pkOrder" PRIMARY KEY ("orderId");
ALTER TABLE "Order" ADD CONSTRAINT "fkOrderAccount" FOREIGN KEY ("accountId")
REFERENCES "Account" ("accountId");
```

4.4 Розробка клієнтської частини

У даній роботі було вирішено використання Telegram чат-боту у якості клієнтської частини нашого проекту. Використання чат боту у якості клієнтської частини надає декілька переваг у розробці:

– *Легкість в розробці*. Чат боти можна створювати та налаштувати за допомогою Telegram Bot API, який забезпечує простий та інтуїтивно зрозумілий інтерфейс для розробників. Це може допомогти заощадити час і зусилля під час процесу розробки, а також може полегшити розробникам створення та підтримку чат-ботів;

– *Легкість в налаштуванні*. Чат боти можна налаштовувати для виконання широкого спектру завдань і функцій, а також інтегрувати з іншими інструментами та системами. Це може допомогти розробникам створити чат-ботів, адаптованих до конкретних потреб і вимог проекту, а також покращити загальну функціональність і корисність чат-бота;

– *Легкість в розширенні.* Вони можуть обробляти велику кількість взаємодій одночасно та можуть використовуватися для обслуговування великої кількості користувачів, не вимагаючи додаткового персоналу чи ресурсів. Це може допомогти підвищити ефективність проекту та полегшити роботу з раптовими збільшенням трафіку чи використання;

– *Безпечність*. Чат боти використовують шифрування для захисту даних і зв’язку користувачів і можуть бути налаштовані відповідно до ряду правил і стандартів конфіденційності. Це може допомогти забезпечити захист і безпеку даних користувачів, а також збільшити їх довіру та впевненість у проекті.

В якості інструменту для розробки було вирішено використання фреймворку для розробки чат ботів Telegraf. Telegraf — це популярна платформа з відкритим вихідним кодом, яка дозволяє розробникам швидко й легко створювати багатофункціональні чат-боти. Він надає надійний набір

інструментів і функцій, які спрощують інтеграцію функціональності чат-бота в існуючі програми або створення нових чат-ботів з нуля.

Щоб використовувати Telegraf потрібно встановити саму бібліотеку та її залежності. Це можна зробити за допомогою наступної команди:

```
npm install telegraf
```

Після встановлення фреймворку ми можемо почати створювати наш чат-бот. Telegraf надає простий, але потужний API, який дозволяє легко визначити функціональні можливості нашого чат-бота. Наприклад, ми можемо використовувати метод hears, щоб визначити відповідь на певний вхід користувача:

```
const Telegraf = require('telegraf');
const bot = new Telegraf(process.env.BOT_TOKEN);

bot.hears('hello', (ctx) => {
  ctx.reply('Привіт!');
});
```

У цьому прикладі наш чат-бот відповість "Hello there!" кожного разу, коли він отримує повідомлення "hello" від користувача. Ми також можемо використовувати регулярні вирази, щоб відповідати введеним користувачам і викликати відповідь:

```
bot.hears(/how are you/i, (ctx) => {
    ctx.reply('У мене все добре, дякую, що запитали!');
});
```

У цьому випадку наш чат-бот відповість «У мене все добре, дякую, що запитали!» кожного разу, коли він отримує повідомлення, яке відповідає регулярному виразу “/how are you/I”, незалежно від точного формулювання чи регістру введених даних.

Окрім відповідей на сповіщення користувачів, Telegraf також дозволяє легко надсилати повідомлення проактивно. Наприклад, ми можемо використовувати метод sendMessage, щоб надіслати повідомлення певному користувачеві або групі користувачів:

```
bot.sendMessage(chatId, 'Привіт! Це автоматичне повідомлення.');
```

У цьому прикладі чат-бот надішле повідомлення «Привіт! Це автоматичне повідомлення» користувачеві або групі визначеною змінною chatId.

Приклад розробленого модуля за допомогою фреймворку Telegraf:

```
async () => {
    const { Telegraf, Scenes, session } = npm.telegraf;
    lib.bot.client = new Telegraf(config.bot.token);
    lib.bot.secretPath = lib.bot.client.secretPathComponent();
    const webhookURL = `${config.bot.webhookURL}/api/bot/${lib.bot.secretPath}`;

    if (application.worker.id === 'W1') {
      console.debug('Initialize telegraf');
      lib.bot.client.telegram.setWebhook(webhookURL).then(() => {
        console.debug('Webhook is set on', config.bot.webhookURL);
      });
    }

    lib.bot.client.telegram.setMyCommands([
      { command: 'order', description: 'Зробити замовлення' },
      { command: 'about', description: 'Про нас' },
      { command: 'support', description: 'Підтримка' },
    ]);

    const stage = new Scenes.Stage([lib.bot.scenes.shop.init()]);

    lib.bot.client.use(session());
    lib.bot.client.use(stage.middleware());

    lib.bot.client.command('order', (ctx) => ctx.scene.enter('order'));

    lib.bot.client.start((ctx) => {
      lib.bot.log(ctx, 'started bot');
      const filePath = '/content/greetings.md';
      const content = application.resources.get(filePath).toString();
      ctx.reply(content);
    });

    lib.bot.client.command('support', (ctx) => {
      lib.bot.log(ctx, 'asked for support');
      const filePath = '/content/support.md';
      const content = application.resources.get(filePath).toString();
      ctx.reply(content);
    });

    lib.bot.client.command('about', (ctx) => {
      lib.bot.log(ctx, 'requested about info');
      const filePath = '/content/about.md';
      const content = application.resources.get(filePath).toString();
      ctx.reply(content);
    });
  };
```

5 ТЕСТУВАННЯ ТА ФАКТОРИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Основні поняття тестування

Тестування програмного забезпечення представляє собою процес дослідження програмного забезпечення (ПЗ) з метою встановлення ступеню якості/готовності продукту для кінцевого замовника [11, 12, 28].

Існує велика кількість підходів вирішення задач тестування.

Тестування може виконуватися на всіх стадіях життєвого циклу розробки, на кожній з яких складають плани тестування, де описується кожний елемент програми/системи, що повинен бути протестований, а саме:

- дії, які потрібно виконати,
- тестові дані, які необхідно вводити, та
- результат, що очікується в результаті тесту.

Частіше всього на тестування недостатньо виділяється часу, щоб провести найбільш повне тестування. Тоді основна задача складається в тому, щоб вибрати відповідний набір тестів, для проведення в стислі строки. Для цього в тестуванні використовують нарощувальний підхід. Тестування поділяється на стадії, кількість яких пропорційна часу на тестування [28, 29].

Задача тестувальників впевнитися, що програма чи продукт готові до випуску. Тестувальник повинен звести до мінімуму кількість «неприємних сюрпризів», які можуть виникнути після встановлення продукту у замовника.

З ISO 9126 [30], якість програмного продукту визначається як сукупна характеристика програмного забезпечення, з урахуванням наступних факторів:

- основна функціональність;
- надійність;
- ефективність;

- практичність;

- можливість супроводжувати;

- практичність;

- мобільність;

- функціональність.

Основний список критеріїв і атрибутів було складено у стандарті ISO 9126 Міжнародної організації по стандартизації. Склад і зміст документації, що супроводжує процес тестування, визначається стандартом IEEE 829-1998 "Standard for Software Test Documentation".

5.2 Стадії тестування (Нарощувальний підхід при тестуванні)

До стадій тестування відносять наступні [28, 30]:

Стадія 1 - Вивчення – Ознайомлення з програмою/системою.

Стадія 2 - Базовий тест - Перевірка виконання основного тестового прикладу – Розробка і реалізація простого тестового прикладу, що охвачує основні можливості функціоналу, які повинна виконувати система перед тим як віддається на повне тестування.

Стадія 3 (необов’язкова, при необхідності) – Аналіз тенденцій – Визначається, чи працює система/програма, як було заплановано, коли ще неможливо попередньо оцінити реальні результати роботи.

Стадія 4 - Основна перевірка (інвентаризація) – визначення різних категорій даних, створення та тестування тестів для кожного елементу категорії.

Стадія 5 - Комбінування вхідних даних – комбінування різних вхідних даних.

Стадія 6 - Граничне оцінювання – оцінювання поведінки програми при граничних значеннях даних.

Стадія 7 - Помилкові дані – Оцінювання реакції системи на введення неправильних даних.

Стадія 8 - Створення напруження – спроба вивести програму з ладу.

5.3 Види тестування програмного забезпечення

Класифікація видів тестування виконується за різними ознаками, частіше всього виділяють:

За об'єктом тестування розрізняють:

- функціональне тестування (functional);
- тестування стабільності (stability / endurance / load);
- юзабіліті-тестування (usability);
- тестування локалізації (localization);
- тестування інтерфейсу користувача (UI);
- тестування продуктивності (performance);
- навантажувальне тестування (load);
- стрес-тестуванням (stress);
- тестування безпеки (security);
- тестування сумісності (compatibility testing).

За знанням систем – чи має розробник доступ до коду програмного забезпечення*:*

- тестування чорного ящика (black box) - немає доступу до коду, доступ тільки через інтерфейси, до яких має доступ як замовник, так й користувач;
- тестування сірого ящика (grey box) – є доступ до коду, але при безпосередньому виконанні тестів доступ до коду непотрібний;
- тестування білого ящика (white box testing) – є доступ до коду та виконувач може писати код, використовуючи розроблені бібліотеки;

За ступенем автоматизації:

- ручне тестування (manual);
- автоматизоване тестування (automated);
- напів-автоматизоване тестування (semiautomated testing).

За ступенем ізольованості компонентів:

- компонентне (модульне) тестування (component/unit);

- інтеграційне тестування (integration);

- системне тестування (system/end-to-end testing).

За часом проведення:

- альфа-тестування (alpha testing);

- регресійне тестування (regression);

- тестування при прийомці (smoke testing);

- тестування нової функціональності (new feature testing);

- тестування при здачі (acceptance);

- бета тестування (beta testing).

За ознакою позитивності сценаріїв:

- позитивне тестування (positive testing);

- негативне тестування (negative testing).

За ступенем підготовленості до тестування:

- тестування по документації (formal testing);

- тестування ad hoc або інтуїтивне тестування (ad hoc testing).

При функціональному тестуванні виконується перевірка чи реалізовані функціональні вимоги, тобто можливості програмного забезпечення, в визначених у вхідній документації умовах, вирішувати завдання, потрібні кінцевим користувачам. Функціональні вимоги визначають, що саме робить продукт, які завдання вирішує.

Функціональні вимоги включають функціональну придатність; точність; можливість до взаємодії; відповідність стандартам та правилам; безпеку.

При необхідності виконання тестування нефункціональних параметрів програми – створюються/описуються тести, необхідні для визначення характеристик ПЗ, що можуть бути виміряні різними додатковими елементами/величинами. До таких тестів відносять:

- тестування продуктивності ПЗ - перевіряється працездатність: навантажувальне тестування, стресове тестування, тестування стабільності та надійності, швидкодії, об'ємне тестування, тестування на "відмову" та відновлення, конфігураційне тестування.

- тестування зручності використання виконується з метою визначення зручності використання програмного забезпечення для його подальшого застосування. Цей метод оцінки полягає у залученні користувачів як тестувальників-випробувачів і підсумовуванні отриманих від них висновків.

- тестування безпеки програм – перевірка конфіденційності даних для запобігання злому програми/продукту.

- тестування сумісності, де основною метою є перевірка коректної роботи продукту в певному середовищі. Середовище може включати в себе наступні елементи: різні браузери (Firefox, Opera, Chrome, Safari, Mozila, Internet Explorer); операційна система (Unix, Windows, MacOS); системне програмне забезпечення (веб-сервер, фаєрвол, антивірус); бази даних (Oracle, MS SQL, MySQL); периферія (принтери, CD/DVD-приводи, веб-камери).

Альфа-тестування – імітація реальної роботи програми/продукту Найчастіше альфа-тестування проводиться на ранніх стадіях розробки продуктів, але іноді може застосовуватися як внутрішнє приймальне тестування. Виявлені помилки можуть бути передані для додаткового дослідження у середовищі, подібному тому, в якому буде використовуватися програмне забезпечення, що тестується.

Бета-тестування – частіше всього, це приймальне тестування вже на території замовника з метою виявлення помилок на реальному середовищі, на проміжку між впровадженням та введенням в промислову експлуатацію. Бета тестування частіше всього проводиться на етапі дослідної експлуатації на території замовника та іноді для того, щоб отримати зворотній зв'язок про продукт від його майбутніх користувачів.

Часто стадія альфа-тестування характеризує функціональне наповнення коду, а бета-тестування – стадію виправлення помилок. При цьому, як правило, на кожному етапі розробки проміжні результати роботи доступні кінцевим користувачам [28, 30].

5.4 Фактори якості програмного забезпечення

Фактор якості програмного забезпечення — це нефункціональна вимога до програми, яка зазвичай не описується в договорі з замовником, але тим не менше є бажаною вимогою, що підвищує якість програмного продукту [11, 28, 29].

Розглянемо основні фактори якості:

- зрозумілість - призначення програмного забезпечення повинно бути зрозумілим із самої програми, та документації;

- повнота - всі необхідні частини програми повинні бути представлені і повністю реалізовані;

- стислість - відсутність зайвої інформації, та інформації, що дублюється;

- можливість перенесення - легкість адаптації програми до іншого середовища: іншої архітектури, платформи, операційної системи, або її версії;

- узгодженість - у всій програмі і в документації повинні використовуватись одні й ті самі узгодження, формати і визначення;

- можливість підтримувати супроводження - показник того, наскільки важко змінити програму для задоволення нових вимог. Ця вимога також показує, що програма повинна бути добре задокументована, не занадто заплутана, і мати резерв для росту при використанні ресурсів (пам’ять, процесор).

- тестованість — здатність програми здійснити перевірку приймальних характеристик: чи підтримується можливість виміру продуктивності?

- зручність використання — простота і зручність використання програмного продукту. Ця вимога в першу чергу відноситься до інтерфейсу користувача;

- надійність — відсутність відмов і збоїв у роботі програми, а також простота виправлення дефектів і помилок;

- ефективність — показник того, наскільки раціонально програма відноситься до ресурсів (пам’ять, процесор) при виконанні своїх задач [30].

Окрім технічного погляду на якість програмного забезпечення, існує і оцінка якості зі сторони користувача [11, 28]. Для цього аспекту якості іноді

використовують термін "юзабіліті". Доволі складно отримати оцінку "юзабіліті" для заданого програмного продукту. Найбільш важливими питаннями, що впливають на оцінку:

– Чи являється інтерфейс користувача інтуїтивно зрозумілим?

– Наскільки просто виконувати прості, часті операції?

– Чи видає програма зрозумілі повідомлення про помилки?

– Чи завжди програма поводить себе так, як очікується?

– Чи є документація, і наскільки вона повна?

– Чи є інтерфейс користувача само-описуючим?

– Чи завжди затримки реакції програми є прийнятними?

5.5 Тестування компонентів системи

Для тестування компонентів системи було вирішено використати вбудований модуль платформи Node.js - модуль assert.

Модуль assert є найпростішим способом для написання тестів. Він не надає ніяких вихідних даних під час виконання тесту, якщо тільки один з випадків виконання не зазнає невдачі. Модуль assert надає простий набір перевірок тверджень, які можна використовувати для перевірки інваріантів. Модуль призначений для внутрішнього використання Node.js, але його можна використовувати в коді програми виконавши імпорт модуля.

Список основних методів модуля assert:

– assert() - Перевіряє, чи значення істинне;

– deepEqual() - Перевіряє, чи рівні два значення;

– deepStrictEqual() - Перевіряє рівність двох значень за допомогою оператора суворої рівності;

– equal() - Перевіряє рівність двох значень за допомогою оператора рівності;

– fail() - Викидає помилку твердження;

– ifError() - Викидає вказану помилку, якщо вказана помилка оцінюється як істина;

– notDeepEqual() - Перевіряє, чи є протилежними два значення;

– notDeepStrictEqual() - Перевіряє, чи два значення є протилежними, використовуючи суворий оператор нерівності;

– notEqual() - Перевіряє, чи два значення не є рівними, за допомогою оператора нерівності;

– notStrictEqual() - Перевіряє, чи є два значення протилежними, використовуючи суворий оператор нерівності;

– strictEqual() - Перевіряє рівність двох значень за допомогою оператора суворої рівності.

Приклад тестування компонентів системи за допомогою модуля assert:

```
'use strict';

const http = require('http');
const assert = require('assert').strict;

require('impress');

const HOST = '127.0.0.1';
const PORT = 8000;
const START_TIMEOUT = 1000;
const TEST_TIMEOUT = 3000;

let callId = 0;

console.log('System test started');

setTimeout(async () => {
  console.log('System test finished');
  process.exit(0);
}, TEST_TIMEOUT);

const tasks = [
  { get: '/', status: 302 },
  { get: '/console.js' },
  {
    post: '/api',
    method: 'signIn',
    args: { login: 'serhii', password: '12345' },
  },
];
```

```
const getRequest = (task) => {
  const request = {
    host: HOST,
    port: PORT,
    agent: false,
  };
  if (task.get) {
    request.method = 'GET';
    request.path = task.get;
  } else if (task.post) {
    request.method = 'POST';
    request.path = task.post;
  }
  if (task.args) {
    const packet = { call: ++callId, [task.method]: task.args };
    task.data = JSON.stringify(packet);
    request.headers = {
      'Content-Type': 'application/json',
      'Content-Length': task.data.length,
    };
  }
  return request;
};

setTimeout(() => {
  tasks.forEach((task) => {
    const name = task.get || task.post;
    console.log('HTTP request ' + name);
    const request = getRequest(task);
    const req = http.request(request);
    req.on('response', (res) => {
      const expectedStatus = task.status || 200;
      setTimeout(() => {
        assert.equal(res.statusCode, expectedStatus);
      }, TEST_TIMEOUT);
    });
    req.on('error', (err) => {
      console.log(err.stack);
    });
    if (task.data) req.write(task.data);
    req.end();
  });
}, START_TIMEOUT);
```

6 ПІДХІД ДО РОЗГОРТАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У цьому розділі описано процес розгортання компонентів системи, у якому хмарний архітектор може вибрати доступні служби, які краще відповідають архітектурним вимогам програмного забезпечення.

6.1 Назва екземпляру EC2

Розробник повинен визначити відповідне ім’я EC2 для його ідентифікації, щоб інший розробник у команді міг ідентифікувати екземпляр для подальшого використання. У AWS (Amazon Web Service) цей процес іменування називається додаванням тегів, і він є абсолютно необов’язковим для запуску екземпляра EC2. Називаючи екземпляр, AWS створює «пару ключ-значення». Де значення представляє ім’я (тег), введене для EC2.

6.2 Вибір відповідного AMI (Amazon Machine Image) і типу екземпляра

AMI включає в себе операційну систему та web сервер, та потребує конфігурації, яка необхідна для запуску екземпляру. AWS має сім різних типів образів ОС. Це Ubuntu, Amazon Linux, macOS, Microsoft Windows, Red Hat, SUSE Linux і Debian. Серед них нещодавно додалася macOS. Amazon EC2 пропонує різноманітний набір типів екземплярів, адаптованих до конкретних випадків використання. Типи екземплярів – це комбінації процесора, пам’яті, сховища та мережевих можливостей. Після ретельного огляду всіх можливих варіантів було обрано Amazon Linux 2 AMI та t1.micro для демонстрації в цій дипломній роботі.

6.3 Налаштування ключів автентифікації

Ключі автентифікації — це облікові дані безпеки, що складаються з відкритого та закритого ключів. Це набір облікових даних безпеки, які використовуються для автентифікації під час підключення до екземпляру Amazon EC2. Ключі автентифікації дозволяють розробнику безпечно підключатися до серверу за допомогою SSH (Secure Shell). Після натискання кнопки «Створити пару ключів» він автоматично завантажить назву закритого ключа у форматі .pem «server-key-pair.pem» на локальну машину, а відкритий ключ буде збережено у екземплярі EC2.

6.4 Запуск і моніторинг екземпляру EC2

Після налаштування екземпляра EC2 розробник може запустити екземпляр, натиснувши кнопку «Запустити екземпляр» в інтерфейсі користувача. Моніторинг є ключовим аспектом забезпечення стабільності, доступності та продуктивності екземплярів Amazon EC2. Вкладка «Перевірки стану», показана на рисунку 5.1, надає чітке уявлення про те, чи виявив EC2 будь-які проблеми, які можуть перешкоджати запуску програм.

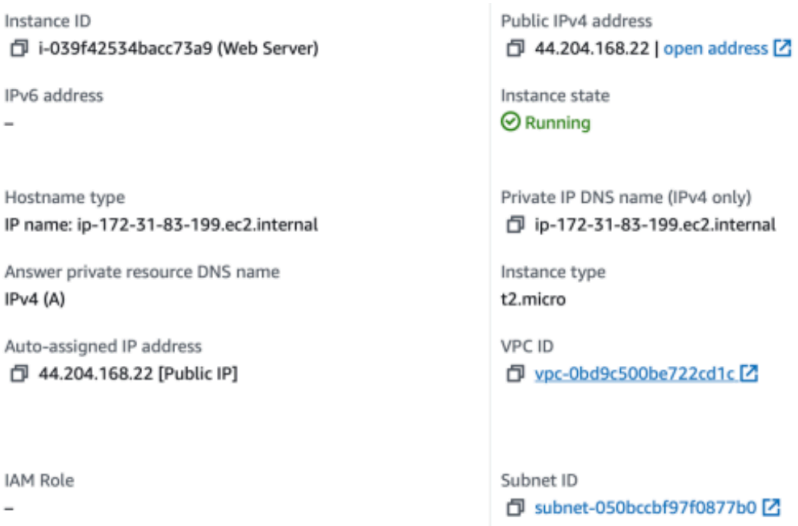

Рисунок 5.1 - Статус роботи екземпляра веб сервісу

Варто зазначити, що AWS виконує автоматичні перевірки кожного запущеного екземпляра EC2, щоб виявити проблеми з апаратним і програмним забезпеченням. Під час запуску екземпляр EC2 буде автоматично пов’язаний із групою безпеки за замовчуванням для VPC.

6.5 Налаштування мережевого доступу до екземпляру

Для керування вхідним і вихідним трафіком група безпеки функціонує як віртуальний брандмауер для екземпляру EC2. Вхідні правила контролюють вхідний трафік до екземпляра, а вихідні – вихідний трафік з екземпляра. Під час запуску екземпляра можна вказати одну або кілька груп безпеки. Без визначення групи безпеки Amazon EC2 використовує групу безпеки за замовчуванням. Правила групи безпеки можна змінювати. Усі екземпляри, підключені до групи безпеки, отримують автоматичні оновлення для нових і змінених правил.

Кожне вхідне правило складається з трьох ключових елементів:

– *Протокол* - Тип протоколу, наприклад TCP або UDP. Надає додаткову опцію ICMP;

– *Діапазон портів* - Дозволяє трафік через певний порт або набір портів;

– *Джерело* - Контролює трафік, який може досягти екземпляра. Це може бути або один IP адреса або діапазон IP-адрес у нотації CIDR. (Наприклад, 10.10.1.0/28)

Для цього демонстраційного розгортання програми був використаний налаштований порт TCP 3000, оскільки програма працює з використанням цього порта. Обидва порта HTTP (ПОРТ 80), і HTTPS (ПОРТ 443) були ввімкнені, як показано в таблиці 5.1. Після дозволу передачі трафіку з порту 3000, вихідний трафік екземпляру EC2 публічного IPv4 DNS може спостерігатись. У цьому випадку URL-адреса буде такою:

http://ec2-44-204-168-22.compute-1.amazonaws.com:3000

Тип	Протокол	Діапазон портів	Джерело
SSH	TCP	22	0.0.0.0/0
Custom TCP	TCP	3000	0.0.0.0/0
HTTPS	TCP	443	0.0.0.0/0
Custom TCP	TCP	8000	0.0.0.0/0
HTTP	TCP	80	0.0.0.0/0

Таблиця 5.1 - Налаштування правил групи безпеки для вхідного трафіку

Вихідна група безпеки буде використовувати параметри за замовчуванням, оскільки цьому екземпляру не потрібна зовнішня комунікація.

6.6 Налаштування серверу NGNIX

Сервер NGINX — це безкоштовне програмне забезпечення з відкритим вихідним кодом. NGNIX є другим за популярністю веб-сервером після Apache. Він найбільш відомий своєю оптимальною продуктивністю, стабільністю та простою конфігурацією.

Він має кілька можливих варіантів використання, таких як зворотній проксі-сервер, кешування, балансування навантаження, потокове передавання медіа та багато іншого. Окрім HTTP-сервера, NGINX може працювати як проксі-сервер електронної пошти та балансувальник навантаження.

У цій дипломній роботі NGINX буде служити протоколом зворотного проксі. Зворотний проксі-сервер — це протокол, який часто знаходиться за брандмауером приватної мережі та надсилає клієнтські запити на належний внутрішній сервер. Іншими словами, зворотний проксі працює для сервера, тоді як проксі працює для клієнта. Зворотний проксі додає ще один рівень абстракції та керування потоком мережевого трафіку між клієнтами та серверами. Nginx

виконуватиметься на порту 80 між сервером нашого застосунку, щоб перехопити весь інтернет-трафік і направити його на порт 3000.

Розробник повинен налаштувати Nginx, щоб полегшити розробку програми і сприяти досягненню її функціональних вимог. З міркувань безпеки не рекомендується розкривати номери портів у загальнодоступних URL-адресах. Щоб вирішити проблему з видимістю номера порту, потрібно додати проксі-сервер у файл конфігурації «nginx.conf», розташований у каталозі /etc, як показано на рисунку 5.2 із підкресленим зеленим полем. Варто зазначити, що проксі-сервер складається з загальнодоступної IPv4-адреси EC2 і перенаправленого порту (у цьому випадку 3000). Після успішного впровадження коду сервер Nginx потрібно перезапустити, щоб він почав діяти.

```
    keepalive_timeout   65;
    types_hash_max_size 4096;

    include             /etc/nginx/mime.types;
    default_type        application/octet-stream;

    # Load modular configuration files from the /etc/nginx/conf.d directory.
    # See http://nginx.org/en/docs/ngx_core_module.html#include
    # for more information.
    include /etc/nginx/conf.d/*.conf;

    server {
        listen       80;
        listen       [::]:80;
        server_name  _;
        root         /usr/share/nginx/html;

        # Load configuration files for the default server block.
        include /etc/nginx/default.d/*.conf;

        error_page 404 /404.html;
        location = /404.html {
        }

        error_page 500 502 503 504 /50x.html;
        location = /50x.html {
        }

        location / {
                proxy_pass http://44.204.168.22:3000;
                }
    }
# Settings for a TLS enabled server.
#
#    server {
#        listen       443 ssl http2;
#        listen       [::]:443 ssl http2;
-- INSERT --                                                    60,1          52%
```

Рисунок 5.2 – Налаштування файлу nginx.conf

6.7 Менеджер процесів PM2

PM2 — це менеджер процесів Node.js із вбудованим балансувальником навантаження. PM2 (Менеджер процесів) — це програма, яка гарантує, що додатки залишаться в робочому стані після запуску. Це також дозволяє контролювати програму. Через PM2 можна отримати доступ до логів додатка та інших життєво важливих показників, таких як навантаження процесора, стан додатку і використання пам’яті системою, як показано на рисунку 5.3.

Рисунок 5.3 – Показники менеджера процесів PM2

Одним із найважливіших і визначальних аспектів розробки програмного забезпечення є розгортання. Встановлювати PM2 на сервер Linux EC2 має бути частиною процесу розгортання системи. PM2 доступний через пакет NPM. Якщо ввести команду «npm install -g pm2», буде завантажено останню версію PM2 на сервер EC2. Команда “pm2 -v” може перевірити версію в CLI.

Застосунок PM2 використовується для запуску програми у фоновому режимі під час її розгортання у робочому стані. Він створює демон, який спостерігає за нашим додатком і підтримує його у робочому стані.

Команда запуску застосунку:

```
pm2 start “npm start” ––name "job Tracker"
```

7 ЕКОНОМІЧНИЙ РОЗДІЛ

7.1 Технологічний аудит розробленого сервісу доставки їжі з використанням методів предметно-орієнтованого проектування

Як було зазначено раніше, стрімке економічне зростання та розвиток широкосмугового інтернету стали рушійною силою глобальної експансії електронної комерції. Споживачі все частіше почали використовувати онлайн-послуги, оскільки реальні доходи споживачів зростають, електронні платежі стають більш надійними, а асортимент постачальників послуг і розмір мереж їх доставки розширюються.

Тому створення сервісу доставки їжі з використанням практик предметно орієнтованого проектування та відкритим вихідним кодом є актуальною задачею, що дозволить малому бізнесу зменшити витрати на опанування нового комерційного сектору та спонукати його розвиток.

У зв’язку з цим, метою нашої роботи було розроблення сервісу доставки їжі з використанням практик предметно орієнтованого проектування. Для нами було розв’язано такі задачі: проведено огляд сектору онлайн-доставки їжі; розроблено багаторівневу архітектуру, яка підвищує надійність, полегшує розроблення нових модулів та покращує тестувальність системи; розроблено модель домену предметної області; розробити бізнес-правила додатку, що реалізують усі випадки використання системи; реалізовано сервіси для взаємодії з зовнішніми службами; розроблено тести для тестування бізнес логіки додатку тощо.

Результатом виконаної магістерської кваліфікаційної роботи стало розроблення нових методів і засобів надання послуг онлайн у секторі доставки їжі клієнтам.

Для встановлення комерційного потенціалу розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування,

було запрошено 3-х відомих і визнаних експертів – кандидатів технічних наук, доцентів Гармаша В.В., Маслія Р.В. та Кривогубченка С.Г.

Встановлення комерційного потенціалу розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування було здійснено за критеріями, наведеними в таблиці 7.1.

Таблиця 7.1 – Рекомендовані критерії оцінювання технічного рівня та комерційного потенціалу будь-якої розробки і їх бальна оцінка

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
	0	1	2	3	4
Технічна здійсненність концепції:					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздатність продукту в реальних умовах
Ринкові переваги (недоліки):					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту	Технічні та споживчі властивості продукту	Технічні та споживчі властивості продукту на	Технічні та споживчі властивості продукту	Технічні та споживчі властивості продукту значно

	значно гірші, ніж в аналогів	трохи гірші, ніж в аналогів	рівні аналогів	трохи кращі, ніж в аналогів	кращі, ніж в аналогів
Ринкові перспективи					
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає

Продовження таблиці 7.1

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
	0	1	2	3	4
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово-промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві
11	Термін реалізації ідеї	Термін реалізації ідеї	Термін реалізації ідеї	Термін реалізації ідеї менше	Термін реалізації ідеї менше

	більший за 10 років	більший за 5 років. Термін окупності інвестицій більше 10-ти років	від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	3-х років. Термін окупності інвестицій від 3-х до 5-ти років	3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Запрошені експерти оцінили розроблений нами сервіс доставки їжі з використанням методів предметно-орієнтованого проектування таким чином (табл. 7.2):

Таблиця 7.2 – Результати технологічного аудиту розробленого мобільного додатка (за шкалою оцінювання 0-1-2-3-4)

Критерії	Прізвище, ініціали експертів		
	Гармаш В.В.	Маслій Р.В.	Кривогубченко С.Г.
	Бали, що їх виставили експерти:		
1	4	3	4
2	3	4	4

3	4	3	4
4	3	4	4
5	4	3	4
6	4	3	4
7	3	4	3
8	4	4	3
9	4	4	4
10	4	4	3
11	4	3	4
12	4	4	4
Сума балів	$СБ_1 = 45$	$СБ_2 = 43$	$СБ_3 = 45$
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_{1}^{3} СБ_i}{3} = \frac{45+43+45}{3} = \frac{133}{3} = 44,33$		

Встановлення комерційного потенціалу розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування будемо здійснювати на основі рекомендацій, наведених в таблиці 7.3 [34].

Таблиця 7.3 – Рівні комерційного потенціалу будь-якої наукової розробки

Середньоарифметична сума балів $\overline{СБ}$, розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0 – 10	Низький
11 – 20	Нижче середнього
21 – 30	Середній
31 – 40	Вище середнього
41 – 48	Високий

Оскільки середньоарифметична сума балів, що їх виставили експерти, складає 44,33 бали, то це свідчить, що розроблений нами сервіс доставки їжі з використанням методів предметно-орієнтованого проектування має рівень комерційного потенціалу, який вважається «високим».

Це пояснюється тим, що нами розроблено удосконалену методику предметно орієнтованого програмного забезпечення, яка дає можливість покращити продуктивність проектування та розроблення програмного забезпечення за рахунок покращення взаємодії команди розробників з експертами предметної галузі.

7.2 Розрахунок витрат на розроблення сервісу доставки їжі з використанням методів предметно-орієнтованого проектування

При розробленні сервісу доставки їжі були зроблені певні витрати. Зокрема:

– Основна заробітна плата Зо розробників, яка визначається за формулою:

$$З_о = \frac{М}{Т_р} \cdot t \text{ грн}, \qquad (7.1)$$

де М – місячний посадовий оклад розробника (дослідника), грн; приймемо, що

М = (6700…25000) грн/місяць;

$Т_р$ – число робочих днів в місяці; приймемо $Т_р$ = 22 день;

t – число днів роботи розробників.

Зроблені розрахунки зведемо до таблиці 7.4:

Таблиця 7.4 – Основна заробітна плата розробників

Найменування посади виконавця	Місячний посадовий оклад, грн	Оплата за робочий день, грн	Число днів роботи	Витрати на оплату праці, грн
1. Науковий керівник магістерської роботи	25000	1136,36	20 годин	≈ 3788 (при 6-годинному робочому дні)
2. Магістрант-студент-виконавець	2000 (беремо 6700)	304,54	84	≈ 25582
3. Консультант з економічної частини	16000	727,27	1,5 години	≈ 182
Загалом				$З_о$ = 29552 грн

– Додаткова заробітна плата $З_д$ розробників розраховується як (10…12)% від величини їх основної заробітної плати, тобто:

$$З_д = \alpha \cdot З_о = (0{,}1 \ldots 0{,}12) \cdot З_о. \qquad (5.2)$$

Приймемо, що α = 0,12. Тоді для нашого випадку отримаємо:

$$З_д = 0{,}13 \times 29552 = 3841{,}76 \approx 3842 \text{ грн.}$$

– Нарахування на заробітну плату $НЗП_{зп}$ розробників (дослідників) розраховуються за формулою:

$$НЗП_{зп} = (З_о + З_д) \cdot \frac{\beta}{100}, \qquad (5.3)$$

де β – ставка обов'язкового єдиного внеску на державне соціальне страхування, %. β = 22%. Тоді:

$$НЗН_{зп} = (29552 + 3842) \times 0{,}22 = 7346{,}68 \approx 7347 \text{ грн.}$$

– Амортизація основних засобів А, які використовувались під час виконання цієї роботи:

$$А = \frac{Ц \cdot Н_а}{100} \cdot \frac{Т}{12} \text{ грн,} \qquad (5.4)$$

де Ц – загальна балансова вартість основних засобів, грн;

$Н_а$ – річна норма амортизаційних відрахувань. Для нашого випадку можна прийняти, що $Н_а$ = (2,5...25)%;

Т – термін використання основних засобів, місяці.

Зроблені розрахунки зведено в таблицю 7.5.

Таблиця 7.5 – Розрахунок амортизаційних відрахувань

Найменування обладнання, приміщень тощо	Балансова вартість, грн.	Норма амортизації, %	Термін використання, міс.	Величина амортизаційних відрахувань, грн
1. Комп'ютерна техніка, обладнання тощо	60000	25	3,0 (при 85% використанні	3187,5 ≈ 3188
2. Приміщення університету, кафедри	26000	3,5	3,0 при 60% використанні	136,5 × 137
Всього				А = 3325 грн

– Витрати на матеріали М розраховуються за формулою:

$$М = \sum_{1}^{n} Н_i \cdot Ц_i \cdot К_i - \sum_{1}^{n} В_i \cdot Ц_в \text{ грн.}, \quad (5.5)$$

де $Н_i$ – витрати матеріалу *і*-го найменування, кг; $Ц_i$ – вартість матеріалу *і*-го най-менування; $К_i$ – коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15); $В_i$ – маса відходів матеріалу *і*-го найменування; $Ц_в$ – ціна відходів матеріалу *і*-го найменування; n – кількість видів матеріалів.

– Витрати на комплектуючі К розраховуються за формулою:

$$К = \sum_{1}^{n} Н_i \cdot Ц_i \cdot К_i \text{ грн}, \quad (5.6)$$

де $Н_i$ – кількість комплектуючих *і*-го виду, шт.; $Ц_i$ – ціна комплектуючих *і*-го виду; $К_i$ – коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15); n – кількість видів комплектуючих.

Під час виконання роботи загальні витрати на матеріали та комплектуючі склали приблизно 1400 грн.

– Витрати на силову електроенергію $В_е$ розраховуються за формулою:

$$В_е = \frac{В \cdot П \cdot Ф \cdot К_п}{К_д}, \quad (5.7)$$

де В – вартість 1 кВт-год. електроенергії, в 2022 р. В ≈ 3,0 грн/кВт;

П – установлена потужність обладнання, кВт; П = 1,3 кВт;

Ф – фактична кількість годин роботи обладнання, годин.

Приймемо, що Ф = 210 годин;

$К_п$ – коефіцієнт використання потужності; $К_п$ < 1 = 0,84.

$К_д$ – коефіцієнт корисної дії, $К_д$ = 0,73.

Тоді витрати на силову електроенергію будуть дорівнювати:

$$В_е = \frac{В \cdot П \cdot Ф \cdot К_п}{К_д} = \frac{3 \cdot 1,3 \cdot 210 \cdot 0,84}{0,73} = 942,41 \approx 943 \text{ грн.}$$

– Інші витрати $В_{інш}$ можна прийняти як (50…300)% від основної заробітної плати розробників, тобто:

$$В_{інш} = (0,5....3) \times З_о. \quad (5.8)$$

Для нашого випадку отримаємо:

$$В_{інш} = 1,5 \times 29552 = 44328 \text{ грн.}$$

– Сума всіх попередніх статей витрат складає витрати на виконання нашої роботи (безпосередньо розробником-магістрантом) – В.

В = 29552+ 3842 + 7347+ 3325 + 1400 + 943 + 44328 = 90737 грн.

– Загальні витрати на розроблення та можливе впровадження розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування $В_{заг}$ розраховуються за формулою:

$$В_{заг} = \frac{В}{\beta}, \quad (5.9)$$

де β – коефіцієнт, який характеризує етап (стадію) виконання цієї роботи. Можна прийняти, що, $\beta \approx 0,92$ [34], оскільки робота майже практично завершена.

Тоді: $В_{заг} = \frac{90737}{0,92} = 98627,17$ грн або приблизно 99 тисяч грн.

Тобто прогнозовані загальні витрати на розробку та можливе впровадження (комерціалізацію) розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування становлять приблизно 99 тисяч грн.

7.3 Розрахунок економічного ефекту від можливої комерціалізації нашої розробки

Економічний ефект від впровадження та можливої комерціалізації розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування пояснюється його значно кращими функціональними можливостями. Тому нашу розробку можна реалізовувати на ринку дещо дорожче, ніж аналогічні за функціями розробки. Тобто, якщо одна ліцензія на рік для суб'єктів бізнесу (які займаються виготовленням та доставкою їжі) становить приблизно $499 (або 19960 грн при курсі НБУ в грудні 2022 року 1$ ≈40 грн), а кількість таких замовників становить 100 шт., то ми можемо реалізовувати нашу розробку на ринку за ціною приблизно 25000 грн за одну ліцензію, або на 25000 – 19960 = 5040 грн або приблизно на 5000 грн дорожче.

Аналіз ринку також показав, що потенційна кількість замовників (які займаються виготовленням та доставкою їжі) розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування також буде зростати (що обумовлено економічною та політичною ситуацією в країні), тобто можна очікувати зростання попиту на нашу розробку принаймні протягом 3-х років після її впровадження. Після цього фахівці кафедри АІІТ ВНТУ обов'язково запропонують ще більш ефективний сервіс доставки їжі.

Тобто, якщо наша розробка буде впроваджена з 1 січня 2024 року, то її результати будуть виявлятися протягом 2024-го, 2025-го та 2026-го років.

Прогноз зростання попиту на нашу розробку може складати по роках:

а) 2024 р. – приблизно + 50 шт. до базового року;

б) 2025 р. – +100 шт. до базового року;

в) 2026 р. – +200 шт. до базового року.

Можливе збільшення чистого прибутку $\Delta П_i$, що його може отримати потенційний інвестор від комерціалізації, тобто виведення нашої розробки на ринок, становитиме:

$$\Delta П_i = \sum_{1}^{n} (\Delta Ц_o \cdot N + Ц_o \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot (1 - \frac{\upsilon}{100}), \qquad (5.10)$$

де $\Delta Ц_o$ – покращення основного якісного показника від впровадження результатів нашої розробки у цьому році. Для нашого випадку це є збільшення ціни реалізації нашої розробки $\Delta Ц_o$ = 25000 - 19960 ≈ 5000 грн (5 тисяч грн);

N – основний кількісний показник, який визначає обсяг діяльності у році до впровадження результатів розробки; N = 100 шт.;

ΔN – покращення основного кількісного показника від впровадження результатів розробки. Таке покращення становитиме по роках: у 2024 році – + 50 шт., у 2025 році +100 шт. та у 2026 році + 200 шт. (відносно базового 2022 року);

$Ц_o$ – основний якісний показник (тобто ціна), який визначає обсяг діяльності у році після впровадження результатів розробки, грн; $Ц_o$= 25000 грн (25 тисяч грн);

n – кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки; для нашого випадку n = 3;

λ – коефіцієнт, який враховує сплату податку на додану вартість; $\lambda = 0{,}8333$;

ρ – коефіцієнт, який враховує рентабельність продукту. Рекомендується приймати ρ= (0,2...0,5); візьмемо ρ= 0,5;

υ – ставка податку на прибуток. У 2022-23 роках υ = 18%. У 2024 році також очікуємо на 18%.

Тоді можливе зростання чистого прибутку $\Delta П_1$ для потенційного інвестора протягом першого року від можливого впровадження нашої розробки (2024 р.) складе:

$$\Delta П_1 = [5 \cdot 100 + 25 \cdot 50] \cdot 0{,}8333 \cdot 0{,}5 \cdot (1 - \frac{18}{100}) \approx 598 \text{ тис. грн.}$$

Можливе зростання чистого прибутку $\Delta П_2$ для потенційного інвестора від можливого впровадження нашої розробки протягом другого (2025 р.) року складе:

$$\Delta П_2 = [5 \cdot 100 + 25 \cdot 100] \cdot 0,8333 \cdot 0,5 \cdot (1 - \frac{18}{100}) \approx 1025 \text{ тис. грн.}$$

Можливе зростання чистого прибутку $\Delta П_3$ для потенційного інвестора від можливого впровадження нашої розробки протягом третього (2026 р.) року складе:

$$\Delta П_3 = [5 \cdot 100 + 25 \cdot 200] \cdot 0,8333 \cdot 0,5 \cdot (1 - \frac{18}{100}) \approx 1879 \text{ тис. грн.}$$

Приведена вартість зростання всіх чистих прибутків від можливого впровадження нашої розробки становитиме:

$$ПП = \sum_{1}^{т} \frac{\Delta П_i}{(1+\tau)^t}, \qquad (5.11)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої роботи, грн;

т – період часу, протягом якого виявляються результати впровадженої роботи, роки. Для нашого випадку т = 3 роки;

τ – ставка дисконтування. Приймемо τ = 0,10 (10%);

t – період часу від моменту початку розроблення сервісу доставки їжі з використанням методів предметно-орієнтованого проектування до моменту отримання можливих чистих прибутків потенційним інвестором.

Тоді приведена вартість зростання всіх можливих чистих прибутків ПП, що їх може отримати потенційний інвестор від комерціалізації нашої розробки, складе:

$$ПП = \frac{598}{(1+0,1)^2} + \frac{1025}{(1+0,1)^3} + \frac{1879}{(1+0,1)^4} \approx 494 + 770 + 1283 = 2547 \text{ тисяч грн.}$$

Теперішня вартість інвестицій PV, що повинні бути вкладені інвестором для реалізації і комерціалізації нашої розробки: $PV = (1,0…5) \times В_{заг}$.

Для нашого випадку $PV = (1,0…5) \times 99 = 3 \times 99 = 297$ тисяч грн.

Розраховуємо абсолютний ефект від можливих вкладених інвестицій $Е_{абс}$.

$$Е_{абс} = ПП – PV, \quad (5.12)$$

де ПП – приведена вартість збільшення всіх чистих прибутків для інвестора від можливого впровадження нашої розробки, грн;

PV – теперішня вартість інвестицій PV = 297 тисяч грн.

Абсолютний ефект від можливого впровадження нашої розробки (при прогнозованому ринку збуту) за три роки складе:

$$Е_{абс} = 2547 – 297 = 2250 \text{ тисяч грн.}$$

Оскільки $Е_{абс} > 0$, то комерціалізація нашої розробки може бути доцільною.

Далі розрахуємо внутрішню дохідність $Е_в$ вкладених інвестицій:

$$Е_в = \sqrt[Т_ж]{1 + \frac{Е_{абс}}{PV}} - 1, \quad (5.13)$$

де $Е_{абс}$ – абсолютний ефект вкладених інвестицій; $Е_{абс} = 2250$ тис. грн;

PV –теперішня вартість початкових інвестицій PV = 297 тис. грн;

$Т_ж$ – життєвий цикл розробки, роки.

$Т_ж = 4$ років (2023-й, 2024-й, 2025-й, 2026-й роки)

Для нашого випадку отримаємо:

$$Е_в = \sqrt[4]{1 + \frac{2250}{297}} - 1 = \sqrt[4]{1 + 7,5757} - 1 = \sqrt[4]{8,5757} - 1 = 1,711 - 1 = 0,711 = 71,1\%.$$

Далі визначимо ту мінімальну дохідність, нижче за яку потенційному інвестору не вигідно буде займатися комерціалізацією нашої розробки.

Мінімальна дохідність або мінімальна (бар'єрна) ставка дисконтування $\tau_{мін}$ визначається за формулою:

$$\tau_{мін} = d + f, \quad (5.14)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2022 році в Україні d = (0,10...0,12);

f – показник, що характеризує ризикованість вкладень; f = (0,1...0,30).

Приймемо f = 0,30.

Для нашого випадку отримаємо:

$$\tau_{мін} = 0,12 + 0,30 = 0,42 \text{ або } \tau_{мін} = 42\%.$$

Оскільки величина $E_в = 71,1\% > \tau_{мін} = 42\%$, то потенційний інвестор у принципі може бути зацікавлений у фінансуванні та комерціалізації нашої розробки.

Далі розраховуємо термін окупності коштів, вкладених у можливу комерціалізацію розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування.

Термін окупності $T_{ок}$ розраховується за формулою:

$$T_{ок} = \frac{1}{E_в}. \quad (5.15)$$

Для нашого випадку термін окупності $T_{ок}$ коштів становитиме:

$$T_{ок} = \frac{1}{0,711} = 1,4 \text{ років} < 3 \text{ років},$$

що свідчить про потенційну доцільність комерціалізації розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування.

Далі проведено моделювання залежності величини внутрішньої дохідності вкладених потенційних інвестицій від рівня інфляції в країні. Як відомо, на наступні роки, на жаль, прогнозується високий рівень інфляції (в межах 30% і вище), що обумовлюється військовою агресію росії проти України.

Прийнявши прогнозований рівень інфляції на наступні роки у 30% отримаємо:

$$ПП = \frac{598}{(1+0,3)^2} + \frac{1025}{(1+0,3)^3} + \frac{1879}{(1+0,3)^4} \approx 354 + 467 + 658 = 1479 \text{ тисяч грн.}$$

Тоді абсолютний ефект від можливого впровадження нашої розробки за три роки складе:

$$Е_{абс} = 1479 – 297 = 1182 \text{ тисячі грн.}$$

Внутрішня дохідність $Е_в$ вкладених інвестицій становитиме:

$$Е_в = \sqrt[Т_ж]{1 + \frac{Е_{абс}}{PV}} - 1,$$

де $Е_{абс}$ – абсолютний ефект вкладених інвестицій; $Е_{абс}$ = 1182 тисячі грн;

PV –теперішня вартість початкових інвестицій PV = 297 тисяч грн.

Для нашого випадку отримаємо:

$$Е_в = \sqrt[4]{1 + \frac{1182}{297}} - 1 = \sqrt[4]{1 + 3,9797} - 1 = \sqrt[4]{4,9797} - 1 = 1,494 - 1 = 0,494 = 49,4\%.$$

Прийнявши прогнозований рівень інфляції на наступні роки у 50% отримаємо:

$$ПП = \frac{598}{(1+0,5)^2} + \frac{1025}{(1+0,5)^3} + \frac{1879}{(1+0,5)^4} \approx 266 + 304 + 371 = 941 \text{ тисяч грн.}$$

Тоді абсолютний ефект від можливого впровадження нашої розробки за три роки складе:

$$Е_{абс} = 941 – 297 = 644 \text{ тисячі грн.}$$

Внутрішня дохідність $Е_в$ вкладених інвестицій становитиме:

$$Е_в = \sqrt[Т_ж]{1 + \frac{Е_{абс}}{PV}} - 1,$$

де $Е_{абс}$ – абсолютний ефект вкладених інвестицій; $Е_{абс}$ = 644 тисячі грн;

PV –теперішня вартість початкових інвестицій PV = 297 тисяч грн.

Для нашого випадку отримаємо:

$$E_в = \sqrt[4]{1+\frac{644}{297}} - 1 = \sqrt[4]{1+2,1683} - 1 = \sqrt[4]{3,1683} - 1 = 1,334 - 1 = 0,334 = 33,4\%.$$

Зроблені розрахунки у вигляді графіків наведено на рис. 5.1.

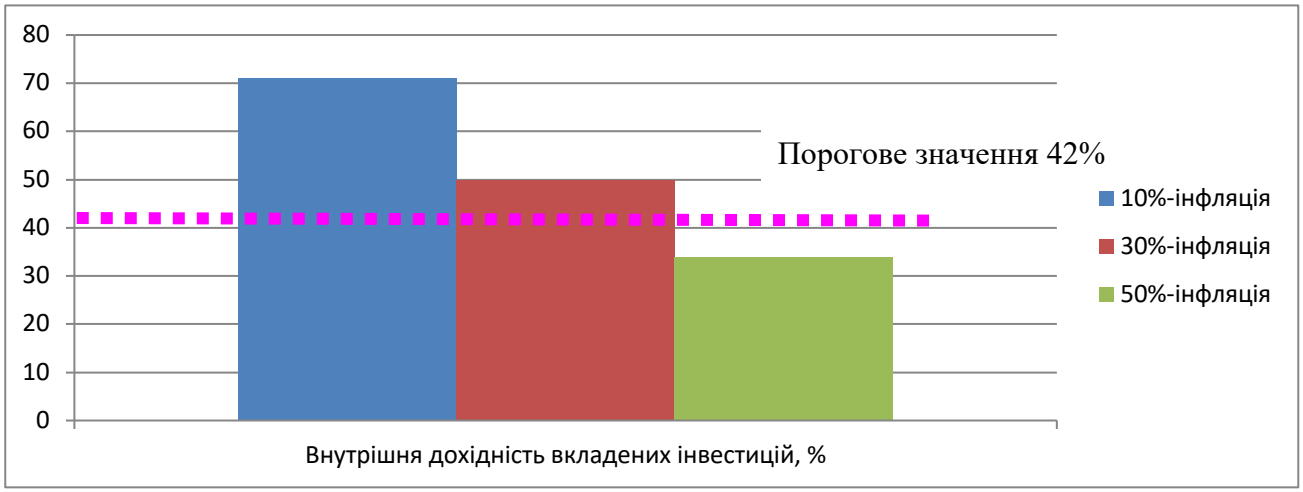

Рисунок 5.1 – Моделювання залежності величини внутрішньої дохідності потенційних інвестицій від рівня інфляції в країні

Аналіз графіка на рис 5.1 показує, що при рівні інфляції в 10% величина внутрішньої дохідності інвестицій становить $E_в$ = 71,1%, що значно більше порогового значення $\tau_{мін}$ = 42%, і тому комерціалізація нашої розробки є доцільною. При рівні інфляції в 30% величина внутрішньої дохідності інвестицій, вкладених в комерціалізацію нашої розробки, становить $E_в$ = 49,4%, що також значно вище порогового значення $\tau_{мін}$ = 42%, і тому комерціалізація нашої розробки потенційним інвестором також є доцільно. При рівні інфляції а країні у 50% величина внутрішньої дохідності інвестицій, вкладених в комерціалізацію нашої розробки, становить $E_в$ = 33,4%, тобто практично наближається до її порогового значення $\tau_{мін}$ = 42%. Тобто виробництво і доставка їжі є досить вигідною справою для бізнесу. Да хто колись в цьому сумнівався!?

Результати виконаної економічної частини магістерської кваліфікаційної роботи зведено у таблицю:

Показники	Задані у ТЗ	Досягнуті у магістерській кваліфікаційній роботі	Висновок
1. Витрати на розробку	Не більше 100 тис. грн	99 тис. грн.	Досягнуто
2. Абсолютний ефект від впровадження розробки (в майбутній вартості грошей), тисяч грн	Не менше 2000 тисяч грн (за три роки)	2250 тисяч грн	Виконано
3. Внутрішня дохідність інвестицій, %	не менше 42%	71,1%	Досягнуто
4. Термін окупності інвестицій, роки	до 3-ти років	1,4 років	Виконано

Таким чином, основні техніко-економічні показники розробленого нами сервісу доставки їжі з використанням методів предметно-орієнтованого проектування, визначені у технічному завданні, виконані.

ВИСНОВКИ

У даній магістерській кваліфікаційній роботі було розроблено та інтегровано високоефективний сервіс доставки їжі з використанням практик предметно орієнтованого проектування з метою підвищення продуктивності проектування та розробки програмного забезпечення за рахунок кращого розуміння бізнес процесів предметної галузі.

В першому розділі роботи було проаналізовано сучасний стан проблеми, проведений огляд сектору онлайн доставки їжі та основних концепцій предметно-орієнтованого проектування.

В другому розділі роботи було проведено огляд існуючих практик розробки та проектування програмного забезпечення, та проведений аналіз архітектурних шаблонів вищого рівня.

В третьому розділі було розроблено модель бізнес-процесів для отримання повного їх розуміння, та була спроектована модель бізнес-логіки системи.

В четвертому розділі було описано розробку компонентів системи, проведене налаштування середовища розробки, реалізовано клієнтську частину, розроблено компоненти бізнес-логіки та розроблена модель даних системи.

В п’ятому розділі описано процеси тестування та визначені основні фактори якості розробленого програмного забезпечення.

В шостому розділі був реалізований підхід до розгортання програмного забезпечення.

Серед практиків, методи предметно-орієнтованого проектування є загальновизнаним підходом до побудови додатків. Застосування концепцій методів предметно-орієнтованого програмування є складним завданням, оскільки йому не вистачає опису процесу розробки програмного забезпечення та класифікації в рамках існуючих підходів до розробки програмного забезпечення.

В даній роботі було застосуванно та удосконаленно методики проектування предметно орієнтованого програмного забезпечення на прикладі розробки онлайн сервісу доставки їжі, що надає можливість підвищити

продуктивність проектування та розробки програмного забезпечення за рахунок покращення взаємодії команди розробників з експертами предметної області.

ПЕРЕЛІК ПОСИЛАНЬ

1. Nanehkaran Y. An Introduction To Electronic Commerce. *International Journal of Scientific & Technology Research*, 2013. Vol. 2, Issue 4. P. 190–193.

2. Report From The Commission To The Council And The European Parliament, *Final report on the E-commerce Sector Inquiry*. Brussels, 2017. - URL: http://ec.europa.eu/competition/antitrust/sector_inquiry_final_report_en.pdf (дата звернення 01.11.2022)

3. Gallaugher J. E-Commerce and the Undulating Distribution Channel. Communications of the ACM. 2002. Vol. 45, Issue 7. P. 89–95. - URL: http://doi.org/10.1145/514236.514240 (дата звернення 01.11.2022)

4. Operkent A. The Law Problems of Electronic Economy. *Journal of Monetary Economics*, 2001. Issue 12. P. 89–90.

5. Sidorova O. V. Regulirovanie elektronnoy ekonomicheskoy deyatel'nosti v zarubezhnykh stranakh. Problemy sovremennoy ekonomiki, 2011. Issue 2. P. 97–100.

6. Melnychuk O. S. Hlobalni tendentsii rozvytku elektronnoi komertsii. Naukovi pratsi NDFI, 2014. Issue 1 (66). P. 58–69.

7. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003. 320 с. ISBN 9780321125217.

8. Vernon V. Implementing Domain-Driven Design : Addison-Wesley Professional, 2013. 656 с. ISBN 978-0321834577.

9. Meyer Bertrand. Agile! The Good, the Hype and the Ugly – URL: http://ndl.ethernet.edu.et/bitstream/123456789/67154/1/149%20%282%29.pdf (дата звернення 01.11.2022).

10. Швабер, К. Сазерленд Д. Руководство по скраму. Исчерпывающее руководство по скраму: правила игры. – URL: https://brainrain.com.ua/scrum-guide/ - (дата звернення 01.11.2021).

11. Пилон Д. Управление разработкой ПО. СпБ. : Издательство «Питер», 2018. 464 с. ISBN 978-5-459-00522-6.

12. Рубин Кеннет С. Основы Scrum: практическое руководство по гибкой разработке ПО. М.: ООО «И.Д.Вильямс», 2016. 544 с. ISBN 978-5-8459-2052-2.

13. Московко С.Г., Богач І.В. Аналіз архітектурних шаблонів програмного забезпечення. *L Науково-технічна конференція факультету інтелектуальних інформаційних технологій та автоматизації*: матеріали конференції ВНТУ, електронні наукові видання (м. Вінниця, 2021). - URL: https://conferences.vntu.edu.ua/index.php/all-fksa/all-fksa-2021/paper/view/11903/10500 (дата звернення 01.11.2022).

14. How to build quality software solutions using TDD and BDD? - URL: https://y-sbm.com/blog/difference-between- tdd-and-bdd (дата звернення 01.11.2022).

15. Рыбаков М.Ю. Бизнес-процессы: как их описать, отладить и внедрить. Практикум. Издательство Михаила Рыбникова, 2016. 392 с.

16. What is Test Driven Development (TDD)? - URL: https://www.guru99.com/test-driven-development.html (дата звернення 01.11.2021).

17. Khanam Z. Evaluating the Effectiveness of Test Driven Development: Advantages and Pitfalls. *International Journal of Applied Engineering Research*, 2017. С. 7705–7716.

18. Behavior Driven Development (BDD) - URL: https://www.agilealliance.org/glossary/bdd/ (дата звернення 01.11.2022).

19. Behavior Driven Development: Alternative to the Waterfall Approach. - URL: https://www.agilest.org/devops/behavior-driven-development/ (дата звернення 01.11.2022).

20. Duvall P.M. Continuous Integration: Improving Software Quality and Reducing Risk, 2007. 336 с. - ISBN 9780321336385.

21. Continuous Integration. - URL: https://www.agilest.org/devops/continuous-integration/ (дата звернення 01.11.2022).

22. Chen L. Continuous Delivery: Overcoming adoption challenges. Journal of Systems and Software, 2017. Vol. 128. P. 72-86.

23. The Advantages and Disadvantages of CI/CD for FinTech. - URL: https://www.intellias.com/the-pros-and-cons-of-ci-cd-for-fintech/ (дата звернення 01.11.2022).

24. Millett S., Tune N. Patterns, Principles, and Practices of Domain-Driven Design : Wrox, 2015. 790 с. ISBN 978-1118714706.

25. Gamma E., Helm R. Design Patterns: Elements of Reusable Object- Oriented Software : Addison-Wesley Professional, 1994. 416 с. ISBN 978- 0201633610.

26. Schwaber K. Scrum Development Process. OOPSLA Business Object Design and Implementation Workshop : матеріали конференції ACM, електронні наукові видання., London : Springer, 1997. - URL: 48 http://damiantgordon.com/Methodologies/Papers/Business Object Design and Implementation.pdf (дата звернення 01.11.2021).

27. How to publish and handle Domain Events. - URL: http://www.kamilgrzybek.com/design/how-to-publish-and-handle-domain-events/ (дата звернення 01.11.2022).

28. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017. 432 с. ISBN 9780134494166.

29. What is Domain-Driven Design (DDD) | Pros & Cons. - URL: https://codezup.com/what-is-domain-driven- design-ddd-pros-cons/ (дата звернення 01.11.2022).

30. Кислач А.А. Многоуровневая архитектура, Мінск: БНТУ, 2016. С. 102–103.

31. Business Logic Definition. - URL: http://wiki.c2.com/?BusinessLogicDefinition (дата звернення 05.05.2021).\

32. User Stories with Examples and Template. - URL: https://www.atlassian.com/agile/project-management/user- stories (дата звернення 05.05.2021).

33. Microsoft TypeScript: the JavaScript we need, or a solution looking for a problem? - URL: https://arstechnica.com/information-

technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/ (дата звернення 01.11.2022).

34. The Node.js Event emitter. - URL: https://nodejs.dev/learn/the-nodejs-event-emitter (дата звернення 01.11.2022).

35. Блэк Р. Ключевые процессы тестирования. Планирование, подготовка, проведение, совершенствование. М. :Лори, 2011. 544 с. ISBN 5-85582-239-7.

36. Инструменты автоматизации тестирования. - URL: http://ru.qatestlab.com/technologies/software-infrastructure/test-automation-tools/ (дата звернення: 01.11.2022).

37. ISO/IEC 9126. 2001. Software engineering. – Software product quality. – Part 1: Qaulity model. Part 2: External metrics. Party 3: Internal metrics. Part 4: Quality In use metrics – Geneva, Switzerland: International Organization for Standartization.

38. Автоматизация тестирования REST API при помощи Postman и JavaScript - URL: http://quality-lab.ru/test-automation-rest-api-using-postman-and-javascript/ (дата звернення: 01.11.2022).

39. Асанов П. Автоматизация функционального тестирования REST API: секреты, тонкости и подводные камни – URL: https://sqadays.com/ru/talk/34635 (дата звернення: 01.11.2022).

40. Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт. / Укладачі В.О. Козловський, О.Й. Лесько, В.В.Кавецький. – Вінниця : ВНТУ, 2021. – 42 с.

Додатки

Вінницький національний технічний університет
(повне найменування вищого навчального закладу)
Кафедра автоматизації та інтелектуальних інформаційних технологій
(повна назва кафедри)

ЗАТВЕРДЖУЮ

Завідувач кафедри АІІТ

д.т.н., професор Бісікало О. В.

_____
(підпис)

«____» _____ 2022 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на магістерську кваліфікаційну роботу

Розробка та інтеграція сервісу доставки їжі з використанням методів предметно-орієнтованого проектування

08-02.МКР.011.00.000 ТЗ

Керівник магістерської кваліфікаційної роботи

д.т.н., проф. Квєтний Р. Н.

«____» _____ 2022 р.

Розробив студент гр. 1АКІТ-21м

Московко С. Г.

«____» _____ 2022 р.

Вінниця ВНТУ 2022

Додаток А
(обов'язковий)
Технічне завдання на магістерську кваліфікаційну роботу

1 Підстава для проведення робіт

Підставою для виконання магістерської кваліфікаційної роботи на тему: «Розробка та інтеграція сервісу доставки їжі з використанням методів предметно-орієнтованого проектування» є наказ № ___ від _____ р.

Термін виконання робіт:

початок _____ р.
кінець _____ р.

2 Мета та вихідні дані для проведення робіт

Метою магістерської кваліфікаційної роботи є розробка та інтеграція високоефективного сервісу доставки їжі з використанням практик предметно орієнтованого проектування.

Вихідними даними для проведення робіт є індивідуальне завдання на магістерську кваліфікаційну роботу від _____ р.

3 Етапи виконання робіт

Виконавцем всіх перерахованих в даному розділі етапів є: студент групи **АКІТ-21м Московко Сергій Геннадійович** факультету інтелектуальних інформаційних технологій та автоматизації Вінницького національного технічного університету, а замовником є: кафедра автоматизації та інтелектуальних інформаційних технологій.

№ Етапу	Зміст етапу	Строки виконання
Е1	Дослідження методологій розробки програмного забезпечення	
Е2	Проектування моделі предметно-орієнтованої системи	
Е3	Розробка компонентів системи	
Е4	Реалізація підходу до розгортки програмного забезпечення	

4 Призначення і галузь застосування

Розроблений сервіс може бути використаний у сфері харчування для автоматизації процесу доставки їжі.

5 Технічні дані

5.1 BPMN діаграма;
5.2 діаграми відповідальності об’єкта;
5.3 діаграма об’єктної взаємодії
5.4 UML діаграма класів;
5.5 діаграма карти контексту
5.6 модель домену предметної області;

6 Джерела розробки

6.1 Положення про магістерську роботу.
6.2 ISO/IEC 2382-1:1993, Information technology – Vocabulary – Part 1: Fundamental terms.
6.3 ISO/IEC 9126-1:2001, Software engineering – Product quality – Part 1: Quality model.
6.4 ISO/IEC TR 9126-2:2003, Software engineering – Product quality – Part 2: External metrics.
6.5 ISO/IEC TR 9126-3:2003, Software engineering – Product quality – Part 3: Internal metrics.
6.6 ISO/IEC TR 9126-4:2004, Software engineering – Product quality – Part 4: Quality in use metrics.

Дипломник ст. гр. 1АКІТ-21м _____ Московко С. Г.

Додаток Б

(обов’язковий)

Графічна частина

Зав. кафедри АІІТ	_____	професор О.В. Бісікало
	(підпис)	(прізвище та ініціали)
Науковий керівник	_____	професор Р.Н. Квєтний
	(підпис)	(прізвище та ініціали)
Тех.контроль	_____	професор Р.Н. Квєтний
	(підпис)	(прізвище та ініціали)
Норм.контроль	_____	професор Р.Н. Квєтний
	(підпис)	(прізвище та ініціали)
Рецензент	_____	професор В.М. Дубовой
	(підпис)	(прізвище та ініціали)
Ст. групи 1АКІТ-21м	_____	С. Г. Московко
	(підпис)	(прізвище та ініціали)

Додаток В

(обов’язковий)

Графічні матеріали

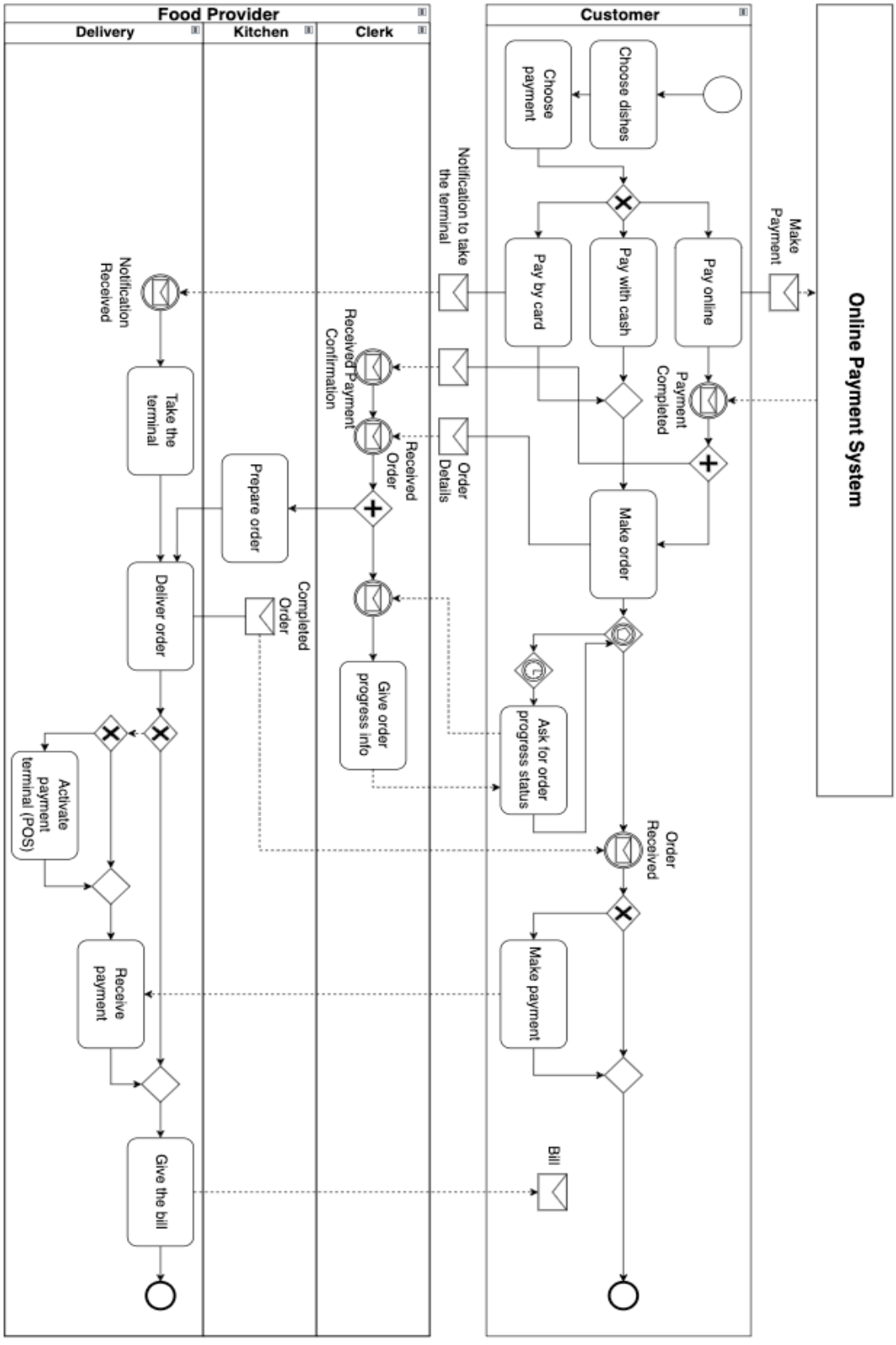

Рисунок В.1 - BPMN діаграма бізнесу у сфері доставки їжі

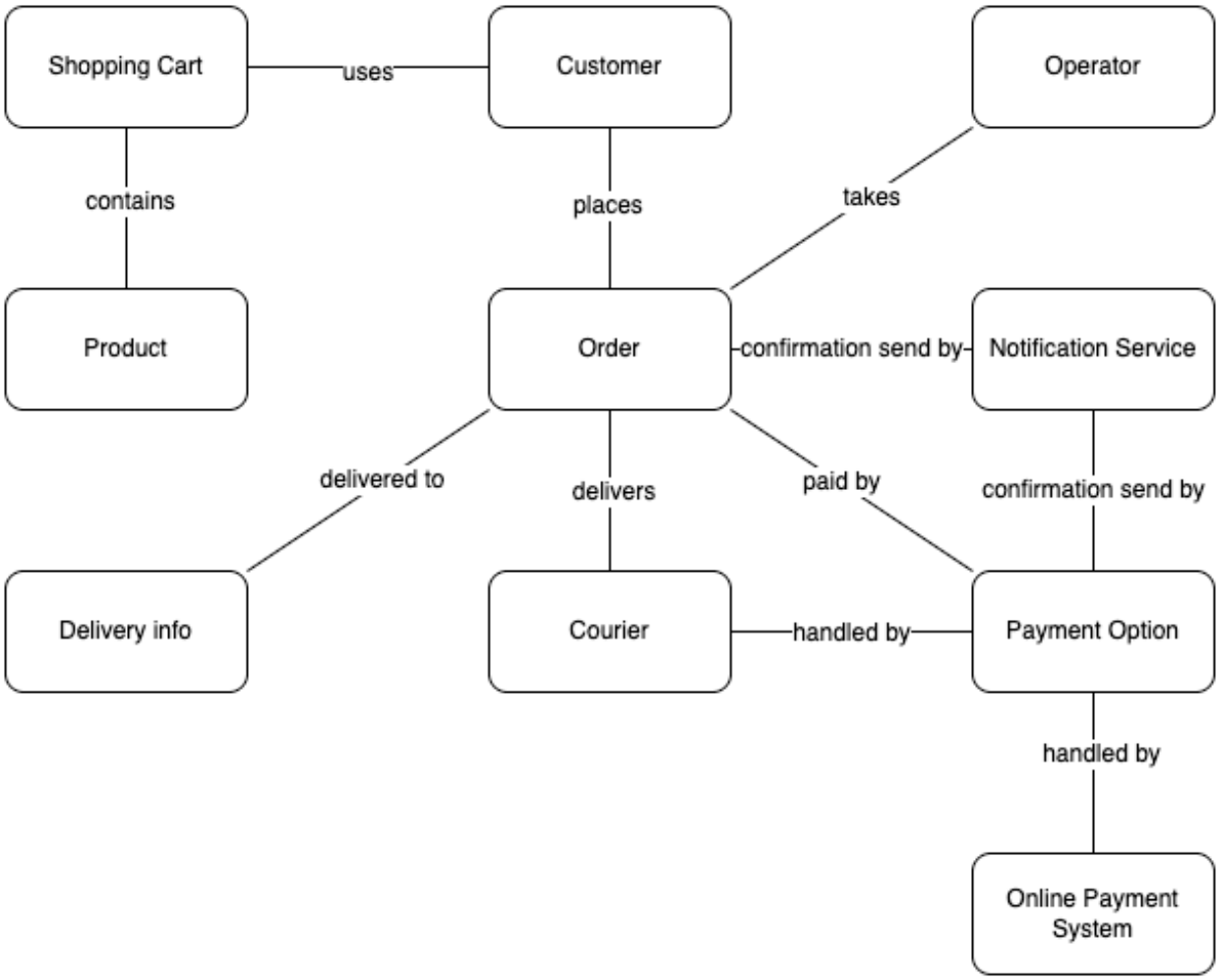

Рисунок В.2 - Діаграма об’єктної взаємодії

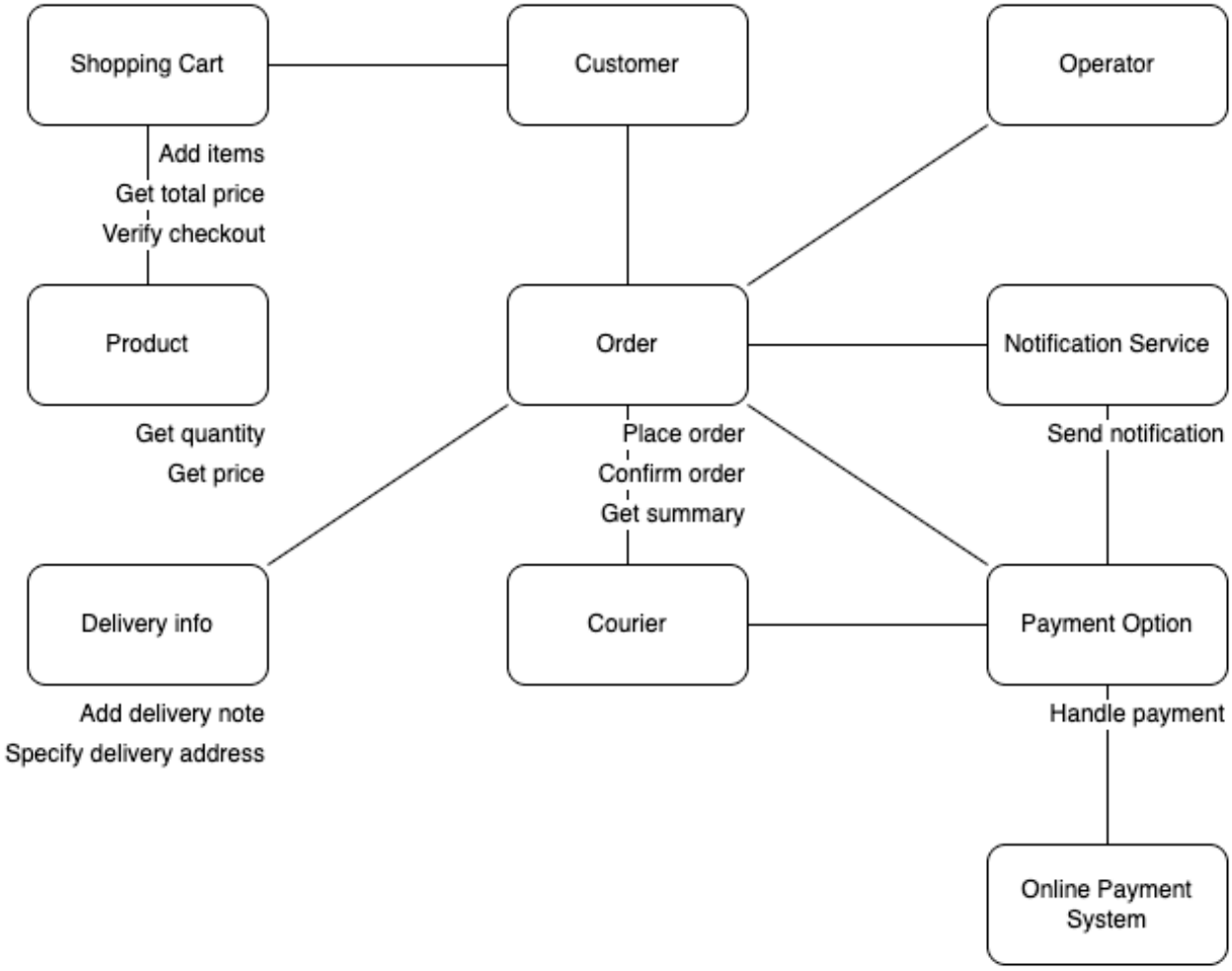

Рисунок В.3 - Діаграма відповідальності об’єкта

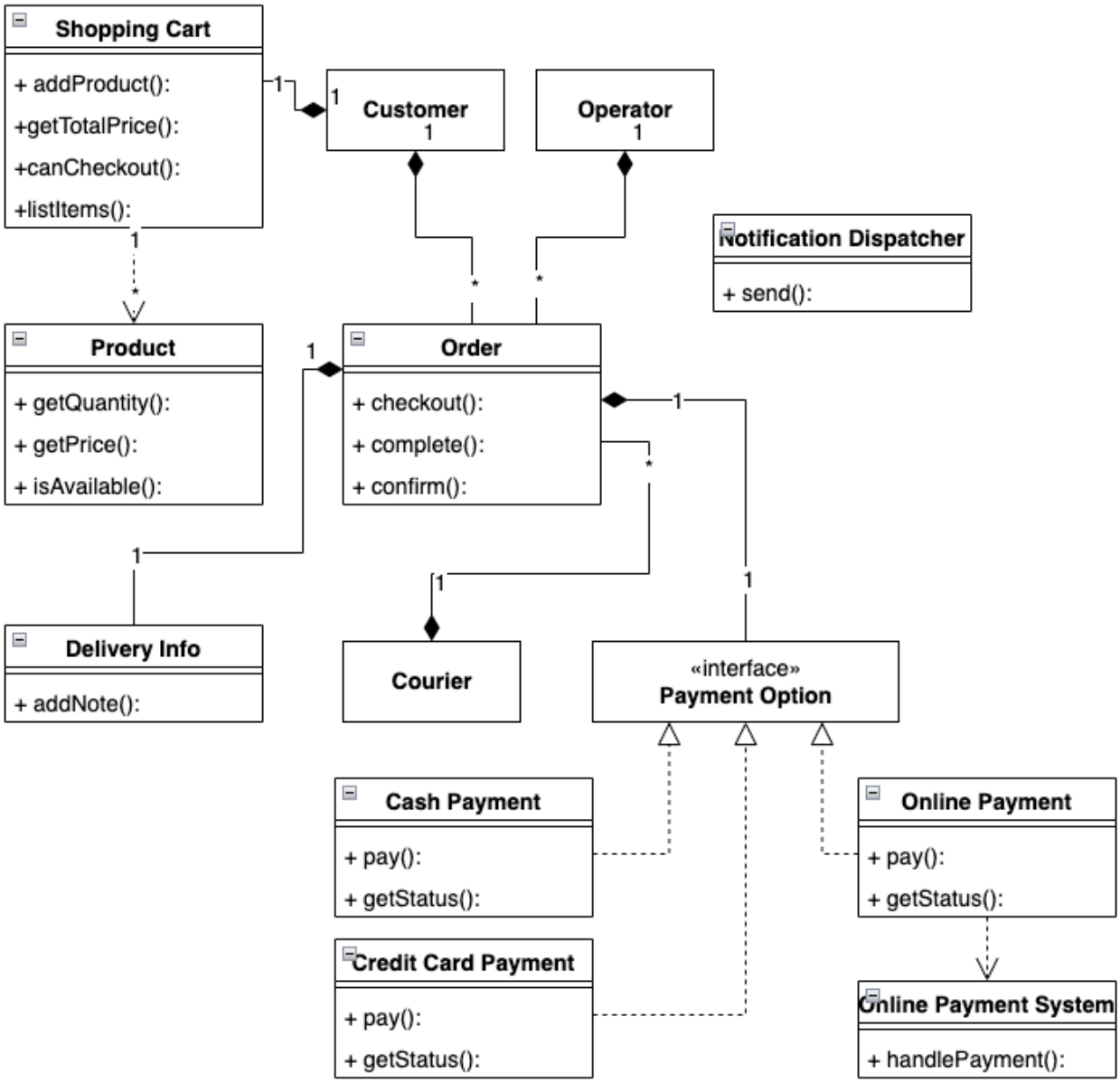

Рисунок В.4 - UML діаграма класів

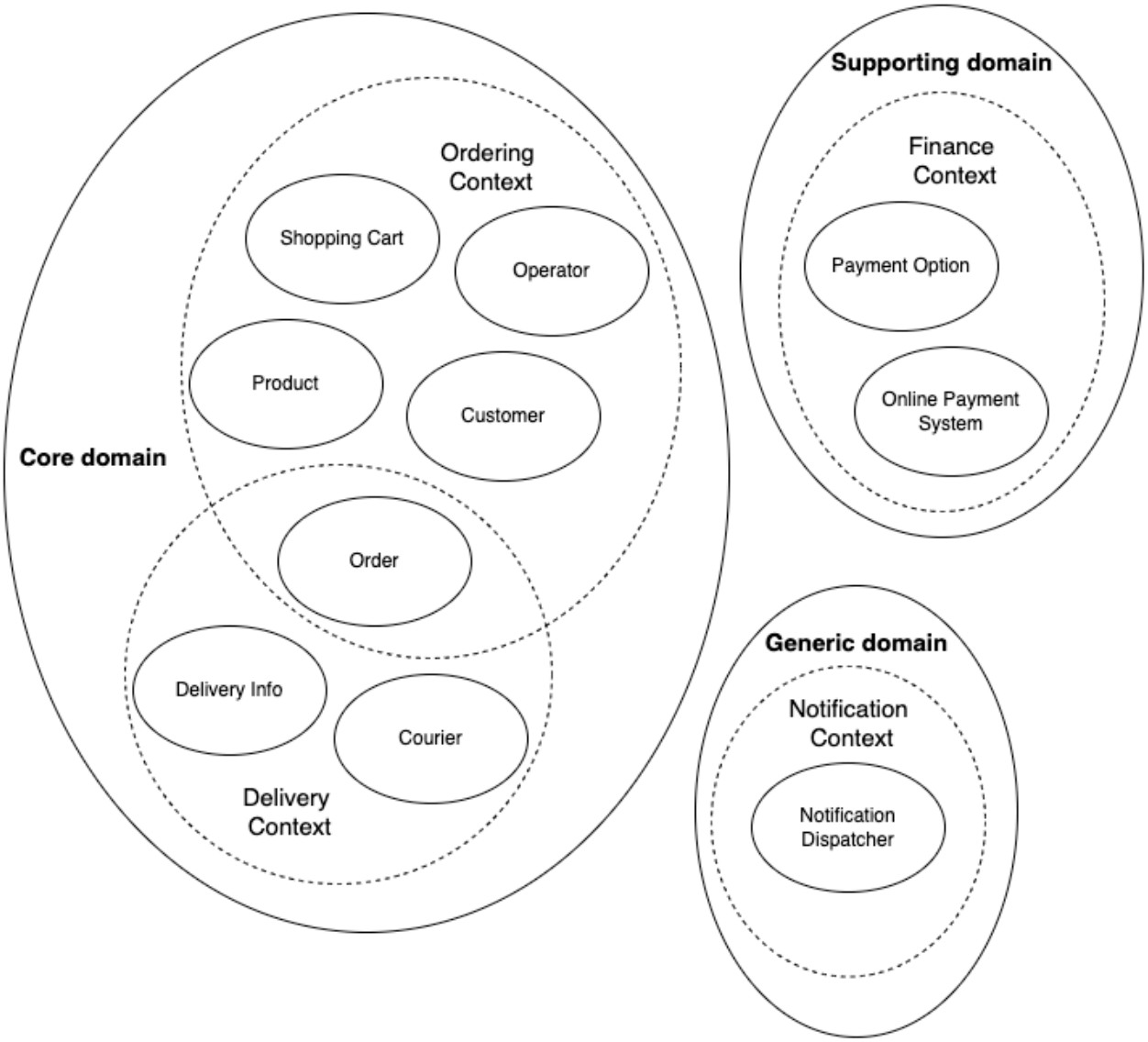

Рисунок В.5 - Діаграма доменів та обмежених контекстів

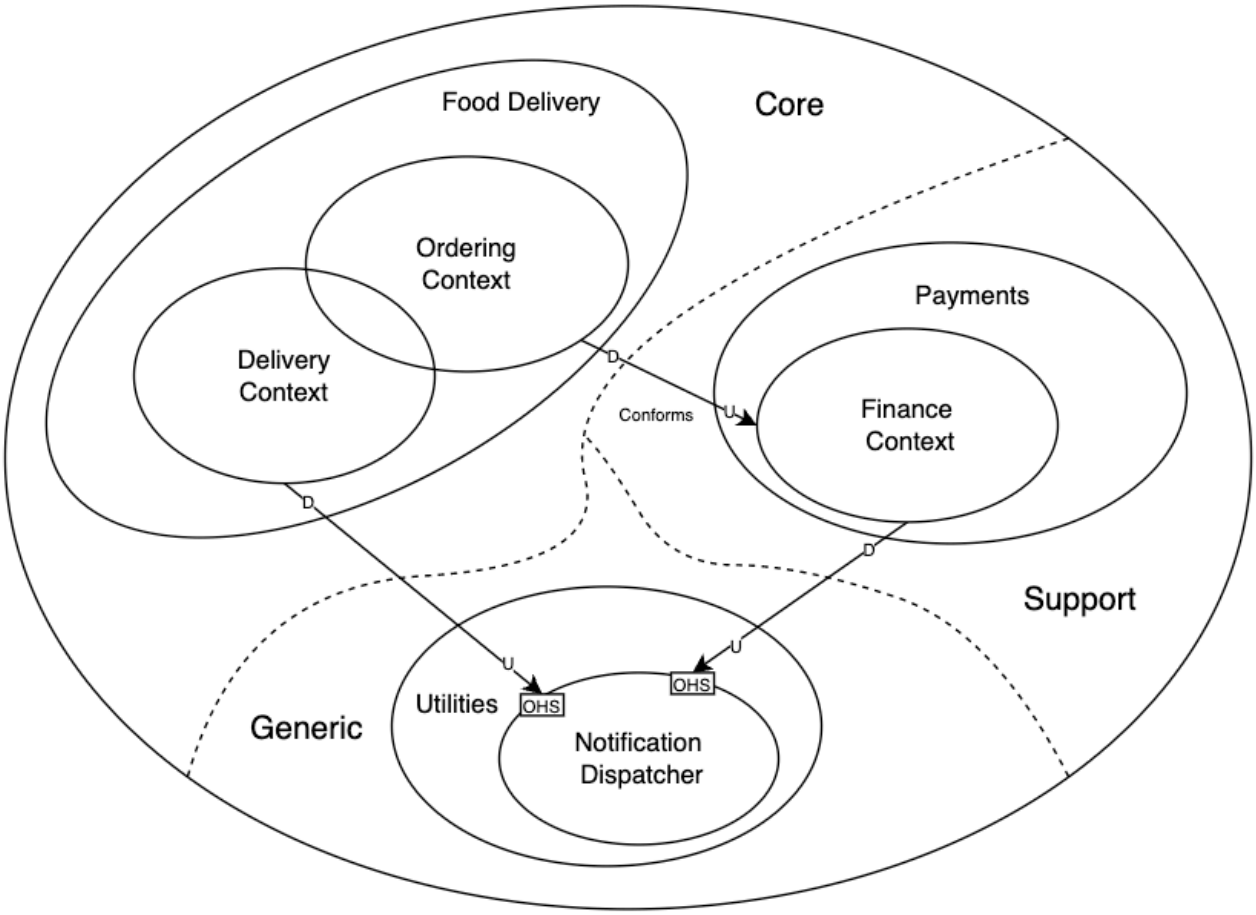

Рисунок В.6 - Діаграма карти контексту

Додаток Г

(обов’язковий)

Акт впровадження результатів магістерської кваліфікаційної роботи в ТОВ «ІКРОК»

Додаток Д

(обов’язковий)

Протокол перевірки навчальної (кваліфікаційної) роботи