Вінницький національний технічний університет
Факультет комп'ютерних систем і автоматики
Кафедра автоматизації та інтелектуальних інформаційних технологій

Пояснювальна записка

до дипломної роботи
бакалавр
(освітньо-кваліфікаційний рівень)

на тему: дослідження методів об'єктно-орієнтованого проектування для розробки веб-додатків

Виконав: студент IV курсу, групи 1АКІТ-17б
спеціальності 151 – Автоматизація та комп'ютерно-інтегровані технології
(шифр і назва спеціальності)

Московко С. Г.
(прізвище та ініціали)

Керівник: к.т.н., доцент кафедри АІІТ
Овчинников К. В.
(прізвище та ініціали)

Рецензент: к.т.н., доцент кафедри САІТ
Варчук І. В.
(прізвище та ініціали)

Вінниця – 2021 року

Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет комп'ютерних систем і автоматики
Кафедра автоматизації та інтелектуальних інформаційних технологій
Освітньо-кваліфікаційний рівень бакалавр
Спеціальність 151 – Автоматизація та комп'ютерно-інтегровані технології
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри АІІТ
_____ д.т.н., проф. Квєтний Р. Н.
"____" _____ 2021 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Московко_Сергію_Геннадійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи дослідження методів об'єктно-орієнтованого проектування для розробки веб-додатків
керівник роботи _Овчинников Костянтин Вячеславович, к.т.н
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "_9_" _03_2021 року №_65_
2. Строк подання студентом роботи_07.06.2021_
3. Вихідні дані до роботи спроектована архітектура додатку; модель домену предметної області; модулі-сервіси взаємодії з зовнішніми службами; діаграми відповідальності та об'єктної взаємодії

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) провести аналіз існуючих практик розробки ПЗ, розробити багаторівневу архітектуру, розробити модель домену, розробити бізнес-правила додатку, реалізувати сервіси для взаємодії з зовнішніми службами

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_ Дослідження методів об'єктно-орієнтованого проектування для розробки веб-додатків Технічне завдання на бакалаврську дипломну роботу;
Дослідження методів об'єктно-орієнтованого проектування для розробки веб-додатків Діаграми бізнес-логіки та архітектури додатку;
_ Вдосконалення системи управління якості програмного забезпечення Базові модулі системи;
_ Вдосконалення системи управління якості програмного забезпечення Модулі рівня бізнесу;
_ Вдосконалення системи управління якості програмного забезпечення Базові модулі системи Модулі рівня додатку.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Богач І. В., доц. каф. АІІТ		
2	Богач І. В., доц. каф. АІІТ		
3	Богач І. В., доц. каф. АІІТ		
4	Богач І. В., доц. каф. АІІТ		

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Провести аналіз існуючих практик проектування програмного забезпечення		
2	Розробити багаторівневу архітектуру, для підвищення надійності, полегшення розробки нових модулів, та кращої тестувальності системи.		
3	Розробити модель домену вибраної предметної області		
4	Розробити бізнес-правила додатку, що реалізують усі випадки використання системи		
5	Розробити адаптери, які перетворюють дані з формату, найбільш зручного для випадків використання та сутностей, у формат, найбільш зручний для якоїсь зовнішньої служби		
6	Реалізувати сервіси для взаємодії з зовнішніми службами		

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

Керівник роботи _____ Овчинников К. В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

У даній бакалаврській роботі проведене дослідження методів предметно-орієнтованого проектування (domain-driven design, DDD) для розробки веб-додатків. Серед практиків, DDD є загальновизнаним підходом до побудови додатків. Застосування концепцій DDD є складним завданням, оскільки йому не вистачає опису процесу розробки програмного забезпечення та класифікації в рамках існуючих підходів до розробки програмного забезпечення.

ABSTRACT

In this bachelor's thesis the research of methods is carried out domain-driven design (DDD) for web application development. Among practitioners, DDD is a generally accepted approach for building applications. Applying DDD concepts is a difficult task, because it lacks a description of the software development process and classifications within existing development approaches.

ЗМІСТ

ВСТУП

Актуальність. При розробці програмного забезпечення однією з перешкод є предметна галузь розробки. Фахівці з розробки програмного забезпечення не можуть бути одночасно фахівцями предметних галузей різних задач. Будь-яке програмне забезпечення має застосування в тій чи іншій сфері діяльності або області інтересів.

Все частіше зустрічаються команди розробників, що займаються проектуванням ПЗ по моделі предметної області. Предметно орієнтоване проектування це підхід до розробки програмного забезпечення, що зосереджує розробку на програмуванні моделі предметної області, що відображає бізнеслогіку [1–6].

Для реалізації ПЗ з предметної області, яка невідома розробнику, йому необхідно заповнити недолік, але не читаючи багато сторінкові і малозрозумілі книги, наукові статті, оскільки це дасть розпливчасте уявлення. Інструментом для подолання цих труднощів є модель, яка будується з навмисно спрощених і строго відібраних знань. Якщо модель дозволяє зосередитися на проблемі, то вона побудована правильно [1–3, 5].

Актуальним є створення вебдодатку за використанням практик предметно орієнтованого проектування, що дозволить розробникам краще розуміти бізнес модель проекту через використання єдиної мови та термінів з експертами предметної області.

Метою роботи є дослідження методик предметно орієнтованого проектування на прикладі розробки веб-додатків.

Для досягнення мети необхідно розв'язати наступні задачі:

1. Провести аналіз існуючих практик проектування програмного забезпечення.

2. Розробити багаторівневу архітектуру, для підвищення надійності, полегшення розробки нових модулів, та кращої тестувальності системи.

3. Розробити модель домену вибраної предметної області

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

5. Розробити адаптери, які перетворюють дані з формату, найбільш зручного для випадків використання та сутностей, у формат, найбільш зручний для якоїсь зовнішньої служби.

6. Реалізувати сервіси для взаємодії з зовнішніми службами.

Об'єктом дослідження є процеси проектування та розробки програмного забезпечення з використанням практик предметно орієнтованого проектування на прикладі розробки веб-додатку.

Предметом дослідження є методи та засоби проектування та розробки програмного забезпечення з використанням предметно орієнтованого проектування.

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

Науково-технічний результат роботи полягає в розробці удосконаленої методики предметно орієнтованого програмного забезпечення на прикладі розробки веб-додатків, що на відміну від існуючих дає можливість покращити продуктивність проектування та розробки програмного забезпечення за рахунок покращення взаємодії команди розробників з експертами предметної галузі.

Практична цінність роботи полягає в розробці бізнес-логіки для покращення взаємодії команди розробників з експертами предметної галузі для підвищення надійності, полегшення розробки нових модулів та кращої тестувальності системи.

Апробація результатів роботи. Результати даної роботи було представлено на L науково-технічній конференції факультету комп'ютерних систем і автоматики Вінницького національного технічного університету на кафедрі

автоматизації та інтелектуальних інформаційних технологій та опубліковані у вигляді тез доповіді [7].

1 СУЧАСНИЙ СТАН ПРОБЛЕМИ ТА ОСНОВНІ ЗАДАЧІ РОБОТИ

1.1 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 [8].

Додаток або веб-сайт також повинні додавати цінності клієнтам, працювати 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 веб-додатків з’явилися найкращі практики розробки програмного забезпечення [9].

1.1.1 Керована тестами розробка

Керована тестами розробка (Test-driven 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 [10]. Керована тестами розробка еволюціонувала як частина концепцій екстремального програмування.

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 1.1.

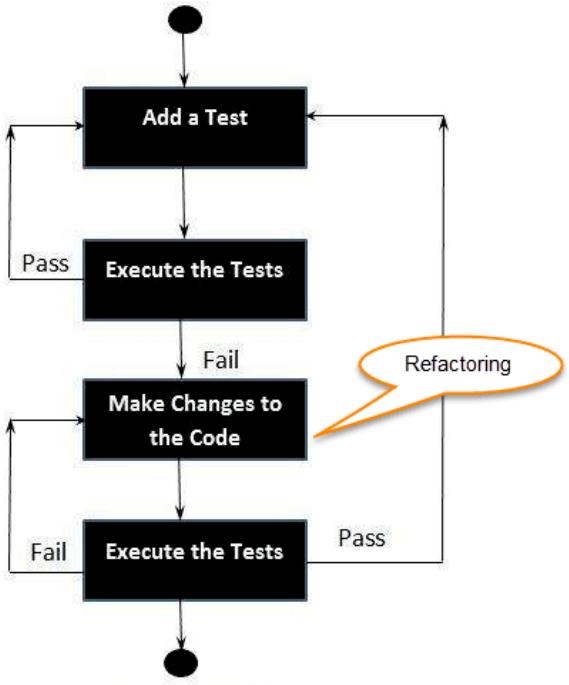

Рисунок 1.1 – Цикл керованою тестами розробки

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 [11].

1.1.2 Керована поведінкою розробка

Керована поведінкою розробка (Behavior-driven development, BDD) - це синтез та вдосконалення практик, що випливають з керованої тестами розробки (TDD) та керованою тестами розробки прийняття (Acceptance test–driven development, ATDD) [12]. 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 розглянути 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, експерт-клієнт буде

доступний у разі потреби. І якщо команди розробників працюють максимально ефективно, їх вимоги до експерта-клієнта будуть мінімізовані [13].

1.1.3 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 [14]. Принцип роботи неперервної інтеграції зображений на рисунку 1.2.

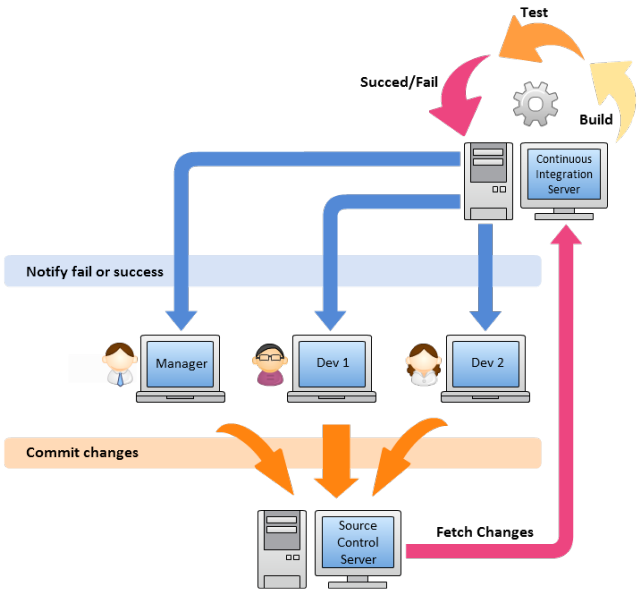

Рисунок 1.2 – Цикл роботи неперервної інтеграції

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 [15].

1.1.4 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 [16]. Модель безперервної доставки зображена на рисунку 1.3

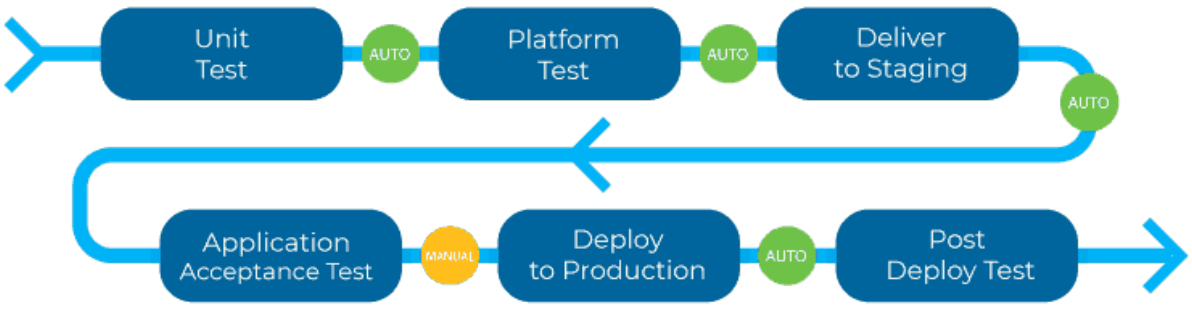

Рисунок 1.3 – Модель безперервної доставки

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 [16].

– 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 [17].

1.2 Концепція предметно-орієнтованого проектування

Предметно-орієнтоване проектування (Domain-driven design, DDD) - це підхід до розробки програмного забезпечення, який зосереджує розробку на

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 [1]. 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 [2]. 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 [2]. Основний домен є субдоменом в якому потрібно найбільше застосовувати предметно-орієнтоване 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 [18].

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

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 [1]. 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 [18].

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 [18]. 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 [1]. 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,

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 [1].

1.2.4 Обмежений контекст

Обмежений контекст - це мовна межа навколо доменної моделі [2]. 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 неприродно зміщеної контекстної межі

1.3 Тактичні шаблони

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.

1.3.1 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 ідентифікатором (який є незмінним). Коли ми порівнюємо рівність сутностей, порівнюємо рівність їх ідентифікаторів.

1.3.2 Об’єкт значення

Об’єкт значення - це об’єкт, який представлений його значенням. У об’єктів заначення немає особистості, і якщо вони змінюються, вони більше не відображають того ж самого занчення. 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 [2].

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.

1.3.3 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 вартості), 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 [2].

1.3.4 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 слу-

жить 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 [2].

1.3.5 Доменна 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.

1.3.6 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 [19].

1.3.7 Модуль

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

1.3.8 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.

1.4 Висновки

В першому розділі розглянуто основні аспекти роботи: проведено огляд та аналіз існуючих практик розробки програмного забезпечення, представлено основні поняття по темі, наведено переваги використання предметно-орієнтованого проектування.

2 ПРОЕКТУВАННЯ СТРУКТУРИ ТА КОМПОНЕНТІВ СИСТЕМИ

2.1 Проектування багаторівневої архітектури

Багаторівнева архітектура є одним із найбільш часто використовуваних архітектурних шаблонів. При його використанні програмне забезпечення логічно або фізично поділяється на рівні, де кожен рівень знаходиться на певному рівні абстракції. Кожен рівень інкапсулює свою логіку і виставляє загальнодоступний інтерфейс рівня вище. Кожен рівень може використовувати лише той рівень, який знаходиться на один рівень нижче (іноді він може використовувати будь-який рівень, що знаходиться нижче), залежність від рівнів вище не допускається. Зазвичай використовуються рівні презентації (використовуються для відображення вмісту користувачем), рівень домену (містить логіку домену) та рівень джерела даних (використовується для доступу до даних у сховищі) [24].

Структурування програмного забезпечення на рівні полегшує розуміння. Наприклад, можна зрозуміти, як реалізується конкретна ділова операція, не знаючи, як вона відображається користувачеві та як дані зберігаються. Також можна замінити весь рівень різним виконанням, якщо виконання відповідає одному контракту. Наприклад, ми можемо замінити виконання реляційної бази даних рівня джерела даних реалізацією NoSQL. Проблема багаторівневої архітектури полягає в тому, що вона не дотримується принципу інверсії залежностей. Вищі рівні зазвичай мають вищий рівень абстракції, але вони залежать від рівнів нижчого рівня, особливо це стосується рівня домену, який залежить від рівня джерела даних, що є інфраструктурним кодом низького рівня.

Ерік Еванс обрав у своїй книзі багаторівневу архітектуру як головний архітектурний зразок, на якому він зосередився найбільше. Він запропонував

чотири рівні для DDD (рисунок 2.1): рівень інтерфейсу користувача, рівень додатку, рівень домену та рівень інфраструктури.

Рівень інтерфейсу користувача відповідає за показ даних користувачеві та інтерпретацію його команд. Замість інтерфейсу користувача може бути і інша система.

Рівень додатку - це рівень із сервісами додатку, який координує операції на рівні домену. Він не містить жодної бізнес-логіки, він працює лише як посередник між рівнем інтерфейсу користувача та рівнем домену. Він організовує рівень домену на основі випадків використання, необхідних рівню користувальницького інтерфейсу, і надає їм інтерфейс API.

Доменний рівень - це серце системи, саме там знаходиться бізнес-логіка програми. Він містить інформацію про бізнес-знання, правила, концепції та робочі потоки, виражені через сутності, об’єкти значення, сукупності, події домену та сервіси.

Рівень інфраструктури містить технічні реалізації. Це може бути доступ до сховища даних через репозиторії, а також зв’язок з іншими системами або інші реалізації на низькому рівні.

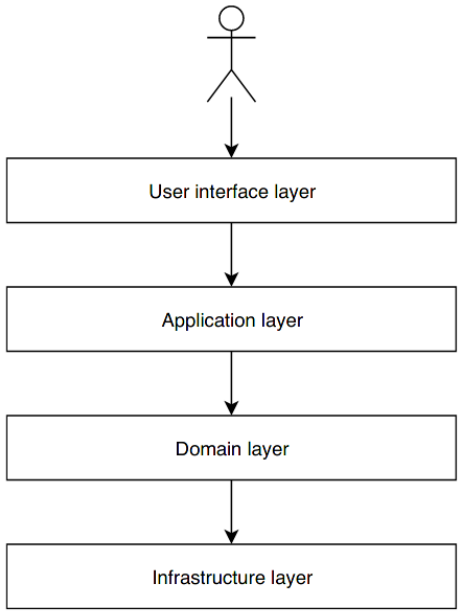

Рисунок 2.1 – Багаторівнева архітектура

Рівні можна створювати як на системному рівні, так і на рівні обмеженому контексту. Коли рівні створюються на рівні обмеженого контексту, кожен обмежений контекст є окремою одиницею і не ділить жодного рівня з іншими обмеженими контекстами, вони мають свої внутрішні рівні. Ця конструкція більше відокремлює обмежені контексти, контексти стають більш незалежними і можуть бути змінені або розгорнуті незалежно в інших контекстах, що робить контексти більш масштабованими у майбутньому. Деякі обмежені контексти навіть не повинні мати багаторівневої архітектури.

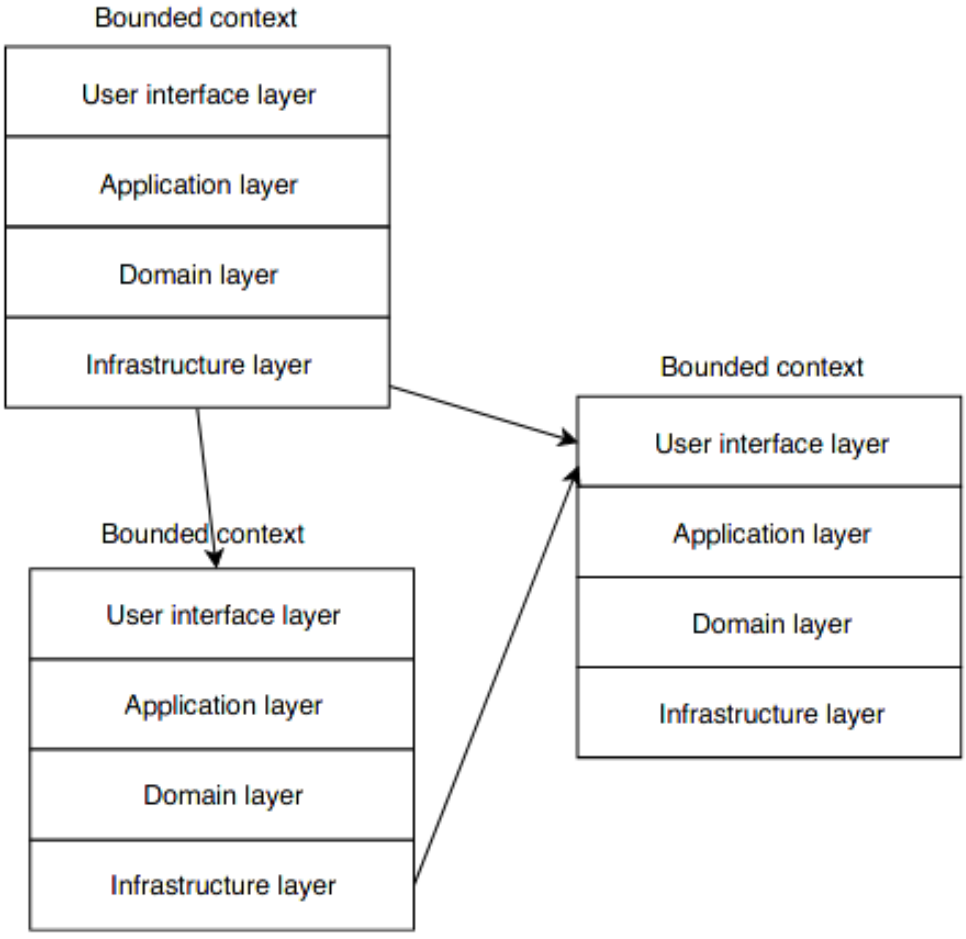

Рисунок 2.2 – Рівні, що використовуються на рівні обмеженому контексту

На основі концепції багаторівневої архітектури Еріка Еванса і за використанням принципу інверсії залежностей була спроектована архітектура додатку (рисунок Б.1).

2.2 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 [25].

Перш ніж розпочати розробку бізнес-домену, ми повинні визначити історії користувачів 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 [26].

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

Історії користувачів додатку:

– Як учасник я хочу мати можливість створювати нову публікацію.

– Як учасник, я хочу мати можливість залишити новий коментар під публікацією.

– Як учасник, я хочу мати можливість відповідати на коментарі інших учасників.

– Як учасник, я хочу мати можливість бачити кількість переглядів у публікації.

– Як учасник я хочу мати можливість бачити загальну кількість вподобань певної публікації.

– Як учасник я хочу мати можливість бачити загальну кількість вподобань первного коментара.

– Як учасник я хочу мати можливість розміщувати лайки у публікації.

– Як учасник я хочу мати можливість розміщувати дизлайки у публікації.

– Як учасник я хочу мати можливість розміщувати лайки у коментарі.

– Як учасник я хочу мати можливість розміщувати дизлайки у коментарі.

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

Іменники:

– учасник;

– публікація;

– коментар;

– лайк;

– дизлайк.

Дієслова:

– створити нову публікацію;

– залишити новий коментар;

– бачити загальну кількість лайків;

– бачити загальну кількість дизлайків;

– відповідати на коментар;

– розміщувати лайки;

– розміщувати дизлайки;

– бачити кількість переглядів публікації.

Використовуючи вказані вище іменники та дієслова, ми можемо скласти схему (рисунок Б.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 безпосереднього спілкування.

Тож давайте слідувати вищезазначеному підходу та розподіляти обов’язки. Діаграма відповідальності об’єкта зображена на рисунку Б.3.

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

2.3 Висновки

У розділі 2 була розроблена архітектура додатку та створена модель бізнесдомену, що дозволить суттєво спростити процеси проектування та розробки програмного забезпечення.

3 РОЗРОБКА КОМПОНЕНТІВ СИСТЕМИ

3.1 Налаштування середовища розробки

При розробці модулів додатку було використано мову програмування TypeScript.

TypeScript - це мова програмування, розроблена та підтримувана корпорацією Microsoft. TypeScript є суворим синтаксичним набором JavaScript, який додає до мови необов'язкове статичне введення тексту. TypeScript призначений для розробки великих додатків та перекомпіляції в JavaScript. Оскільки TypeScript - це набір JavaScript, існуючі програми JavaScript також є дійсними програмами TypeScript [27].

Типи які надає TypeScript дають спосіб описати форму об'єкта, забезпечуючи кращу документацію та дозволяючи TypeScript перевірити, чи правильно працює код програми.

Для того щоб встановити TypeScript у директорії нашої програми, використаємо пакетний менеджер npm.

npm - це менеджер пакетів для середовища виконання Node.js мови програмування JavaScript. Він складається з клієнта командного рядка, який також називається npm, та онлайнової бази даних загальнодоступних та платних приватних пакетів, яка називається реєстр npm. Доступ до реєстру здійснюється через клієнт, а доступні пакети можна переглядати та шукати через веб-сайт npm.

Ініціалізуємо робочу директорії, та завантажуємо необхідні модулі використовуючи наступні команди:

```
npm init
npm i -D typescript ts-node
```

```
npm i -D eslint eslint-config-prettier eslint-plugin-prettier
   eslint-plugin-jest
npm i jest mysql2 uuid ts-jest
```

Завантажуємо необхідні модулі з типами для TypeScript:

```
npm i -D @types/jest @types/node
npm i @types/uuid
```

Для того щоб скомпілювати TypeScript файли у JavaScript потрібно скласти конфігураційний файл:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "lib": ["es6"],
    "allowJs": true,
    "outDir": "build",
    "rootDir": "src",
    "strict": true,
    "noImplicitAny": true,
    "esModuleInterop": true,
    "resolveJsonModule": true,
    "types": ["node", "@types/jest"],
    "typeRoots" : ["./node_modules/@types"]
  },
  "include": [
    "src/**/*"
  ],
  "exclude" : [
    "src/**/*.spec.ts"
  ]
}
```

3.2 Реалізація доменної події

При розробці базового класу доменної події було використано клас EventEmitter модуля events. Event emitter - це об’єкт / метод, який ініціює подію, як тільки відбувається якась певна дія, щоб передати керування батьківській функції [28].

EventEmitter ініціалізується таким чином:

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();
```

Об’єкт класа EventEmitter має такі методи:

– emit, активує певну подію;

– on, додає функцію зворотного виклику, яка буде виконуватися при активації події;

– once, додає одноразового слухача;

– removeListener, видаляє слухача події певної події;

– removeAllListeners, видалити всіх слухачів події;

Методи екземпляра класа EventEmitter були інкапсульовані у методах базового класа доменної події (DomainEvents).

Основні методи класа DomainEvents:

– register, інкапсулює метод on об’єкта класа EventEmitter і реєструє обробника певної події.

– dispatchAggregateEvents інкапсулює метод emit об’єкта класа EventEmitter та виконує обробника події для кожної зареєстрованої події певної сукупності.

– markAggregateForDispatch, реєструє сукупність в якій відбулася якась подія.

Реалізація базового класу доменної події наведено у додатку А.

3.3 Розробка адаптерів перетворення даних

При розробці адаптерів претворення даних було використано архітектурний шаблон data mapper.

Data Mapper - це об’єкт який здійснює двонаправлену передачу даних між постійним сховищем даних (часто реляційною базою даних) та представленням даних у пам’яті (рівень домену). Метою шаблону є збереження представлення в пам’яті та постійного сховища даних незалежно один від одного та від самого мапера даних. Об’єкт складається з одного або декількох методів маперів, що виконують передачу даних. Реалізації мапера відрізняється за обсягом. Універсальні мапери оброблятимуть багато різних типів сутностей домену, спеціальні мапери - один або декілька.

Кожен адаптер перетворення даних має у складі два таких метода:

– toDomain, перетворює дані у формат більш зручний для бізнес логіки;

– toPersistance, перетворює дані у формат більш зручний для певних зовнішніх служб.

Реалізація адаптерів перетворення даних наведена у додатку Д.

3.4 Валідація даних

Для валідації даних на відповідність бізнес правил було використано шаблон проектування специфікація.

Специфікація - це шаблон проектування, за допомогою якого уявлення правил бізнес логіки може бути перетворено у вигляді ланцюжка об’єктів, пов’язаних операціями булевої логіки.

Цей шаблон виділяє такі специфікації (правила) в бізнес логіці, які підходять для «зчеплення» з іншими. Об’єкт бізнес логіки успадковує свою

функціональність від абстрактного сукупного класу AbstractSpecification, який містить всього один метод IsSatisfiedBy, який повертає логічне значення. Після ініціалізування, об'єкт об'єднується в ланцюжок з іншими об'єктами. В результаті, не втрачаючи гнучкості в налаштуванні бізнес логіки, ми можемо з легкістю додавати нові правила.

Реалізація базового класу специфікації наведено у додатку А.

3.5 Розробка бізнес-логіки

При розробці сутностей було використано UML діаграму класів бізнес-логіки. Для генерації ідентифікатора та підтримки унікальної ідентичності сутності було використано модуль uuid.

uuid - це модуль для генерації унікальних ідентифікаторів.

Приклад використання модуля uuid:

```
import { v4 as uuidv4 } from 'uuid';
uuidv4(); // ⇨ '9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d'
```

Методи об'єкта uuid були інкапсульовані в базовому класі UniqueID.

У кожній сутності, сукупності або об'єкта значення присутній статичний фабричний метод для створення такого об'єкта. Також у фабричному методі об'єкта значення відбувається валідація вхідних даних на відповідність бізнес правилам за допомогою шаблону специфікації.

3.6 Взаємодія з зовнішніми службами

Для реалізації доступу до бази даних було використано будівник запросів mysql2.

mysql2 - це драйвер Node.js для MySQL. mysql2 написаний на JavaScript, не вимагає компіляції. mysql2 забезпечує виконання запитів до бази баних написаних мовою SQL.

Приклад базового використання mysql2:

```
// get the client
const mysql = require('mysql2');

// create the connection to database
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  database: 'test'
});

// simple query
connection.query(
  'SELECT * FROM `table` WHERE `name` = "Page" AND `age` > 45',
  function(err, results, fields) {
    console.log(results); // results contains rows returned by
       server
    console.log(fields); // fields contains extra meta data about
       results, if available
  }
);
```

До кожної сутності системи було розроблено відповідний клас репозиторій який інкапсулює методи модуля mysql2.

3.7 Взаємодія через веб

Для реалізації взаємодії з додатком через веб було використано веб-фреймворк Express.js.

Express.js - це мінімальний та гнучкий веб-фреймворк для розробки веб-додатків, який забезпечує надійний набір функцій для веб і мобільних додатків. Завдяки безлічі методів, утиліт HTTP та проміжного програмного забезпечення, процес створення надійного API є швидким та простим.

Веб-фреймворки забезпечують такі ресурси, як HTML-сторінки, сценарії, зображення тощо за різними маршрутами. Наступна функція використовується для визначення маршрутів у Express.js:

```
app.method(path, handler);
```

Метод method можна застосувати до будь-якого з дієслів HTTP - get, set, put, delete. Також існує альтернативний метод, який виконується незалежно від типу запиту.

Path - це маршрут, за яким буде виконуватися запит.

Handler - це функція зворотного виклику, яка виконується, коли відповідний тип запиту знайдений на відповідному маршруті.

Приклад базового використання Express.js:

```
const express = require('express');
const app = express();

app.get('/hello', function(req, res){
   res.send("Hello World!");
});

app.listen(3000);
```

Для обробки запитів були розроблені відповідні контролери. Контролери у якості параметрів конструктора приймають екземпляри сервісів додатку. Контролери після оброки запиту повертають результат роботи у якості відповіді на запит. Реалізація контролерів наведено у додатку Г.

4 ТЕСТУВАННЯ ТА ФАКТОРИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Основні поняття тестування

Тестування програмного забезпечення представляє собою процес дослідження програмного забезпечення з метою встановлення ступеню якості продукту для кінцевого замовника [5, 6, 29].

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;

– результат, що очікується в результаті тесту.

Зазвичай на тестування недостатньо виділяється часу, щоб провести повноцінне тестування. Тоді основна задача складається з того, щоб вибрати відповідний набір тестів, для проведення тестування в стислі строки. Для цього в тестуванні використовують нарощувальний підхід. Тестування поділяється на певні стадії, кількість яких пропорційна часу на тестування [29, 30].

Задача тестувальників впевнитися, що програма чи продукт готові до релізу. Тестувальник повинен звести до мінімуму кількість помилок, які можуть виникнути після встановлення продукту у замовника.

З ISO 9126 [31], якість програмного продукту визначається як сукупна характеристика програмного забезпечення, з урахуванням наступних факторів

– функціональність.

– надійність;

– ефективність;

– практичність;

– можливість підтримки;

– мобільність;

Основний список критеріїв і атрибутів було складено у стандарті ISO9126 Міжнародної організації по стандартизації. Склад і зміст документації, що супроводжує процес тестування, визначається стандартом IEEE829-1998 ”Standard for Software Test Documentation”.

4.2 Стадії тестування (Нарощувальний підхід при тестуванні)

Існують такі стадії тестування [29, 31]:

Стадія 1 - Вивчення - Ознайомлення з програмою/системою.

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

Стадія 3 (необов’язкова) - 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 виконання 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.

4.3 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 вихідного коду програмного забезпечення:

– тестування чорного ящика (black box) - у розробника немає доступу до вихідного коду, доступ тільки через інтерфейси користувача;

– тестування сірого ящика (grey box) - у розробника є доступ до вихідного коду, але при безпосередньому виконанні тестування доступ до коду непотрібний;

– тестування білого ящика (white box testing) - у розробника є доступ до коду та є можливість писати код, використовуючи готові бібліотеки;

За *ступенем автоматизації*:

– ручне (manual);

– автоматизоване (automated);

– напів-автоматизоване (semiautomated testing).

За *ступенем ізольованості компонентів*:

– компонентне (component/unit);

– інтеграційне (integration);

– системне (system/end-to-endtesting).

За *часом проведення*:

– альфа-тестування;

– регресійне тестування;

– тестування при прийомці;

– тестування нової функціональності;

– тестування при здачі;

– бета тестування.

За *ознакою позитивності сценаріїв*:

– позитивне тестування;

– негативне тестування.

За *ступенем підготовленості до тестування*:

– тестування по документації;

– тестування ad hoc або інтуїтивне тестування.

При функціональному тестуванні виконується перевірка чи реалізовані функціональні вимоги, тобто можливості програмного забезпечення, вирішувати завдання, потрібні кінцевим користувачам.

При необхідності виконання тестування не функціональних параметрів програми - створюються та описуються тести, необхідні для визначення характеристик програмного забезпечення, що можуть бути виміряні різними додатковими величинами. До таких тестів відносять:

– тестування продуктивності ПЗ - перевіряється працездатність: навантажувальне тестування, стресове тестування, тестування стабільності та надій-

ності, швидкодії, об’ємне тестування, тестування на ”відмову” та відновлення, конфігураційне тестування.

– тестування зручності - виконується з метою визначення зручності використання ПЗ для його подальшого використання. Цей метод оцінки полягає у залученні кінцевих користувачів як тестувальників-випробувачів і у отриманні від них відгуків.

– 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, Chrome, Safari, Mozila, Microsoft Edge); операційні системи (GNU/Linux, Windows, MacOS); системне програмне забезпечення (веб-сервер, фаєрвол, антивірус); бази даних (MongoDB, MS SQL, MySQL).

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 використовуватисяпрограмне забезпечення, що тестується.

Бета-тестування - частіше всього, це приймальнетестування 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 відгук про продукт від його майбутніх користувачів.

4.4 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 [5, 29, 30].

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.

Окрім технічного погляду на якістьпрограмного забезпечення, існує і оцінка якості зі сторони користувача [5, 29]. 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?

4.5 Тестування бізнес логіки

Для тестування бізнес логіки програми було використано середовище тестування Jest.

Jest - це середовище тестування мови JavaScript, розроблена для забезпечення правильності будь-якої кодової бази JavaScript. Jest дозволяє писати тести за допомогою доступного, звичного та багатофункціонального API, який швидко дає результати.

Перед тим як виконувати тестування потрібно скласти відповідний конфігураційний файл:

```
module.exports = {
  transform: {
    '^.+\\.ts?$': 'ts-jest'
  },
  testEnvironment: 'node',
  testRegex: './src/.*\\.(test|spec)?\\.(ts|ts)$',
```

```
  moduleFileExtensions: ['ts', 'tsx', 'js', 'jsx', 'json', 'node'],
  "roots": [
    "<rootDir>/src"
  ]
};
```

Приклад написання тестів для доменого сервісу "BlogService":

```
describe('BlogService', () => {
  it('should like post', () => {
    expect(true).toEqual(true);
    const member = Member.create({
      username: UserName.create('Mosk6565').value as UserName,
      firstName: FirstName.create('Serhii').value as FirstName,
      lastName: LastName.create('Moskovko').value as LastName,
      email: Email.create('serhii@moskovko.xyz').value as Email,
    }).value as Member;

    const post = Post.create({
      memberId: member.id,
      postTitle: PostTitle.create('Post title').value as PostTitle,
      postText: PostText.create('Post text').value as PostText,
    }).value as Post;

    const likePostResult = new BlogService().likePost(post, member,
        []);

    expect(likePostResult.isSuccess).toEqual(true);
    expect(post.likes.getItems()[0].memberId.toString()).toEqual(
      member.id.toString(),
    );
    expect(post.likes.getItems()[0].isLike()).toEqual(true);
    expect(post.likes.getItems().length).toEqual(1);
  });

  it('should remove like on post', () => {
```

```
    expect(true).toEqual(true);
    const member = Member.create({
      username: UserName.create('Mosk6565').value as UserName,
      firstName: FirstName.create('Serhii').value as FirstName,
      lastName: LastName.create('Moskovko').value as LastName,
      email: Email.create('serhii@moskovko.xyz').value as Email,
    }).value as Member;

    const postId = UniqueID.create();

    const existingVote = PostVote.create({
      memberId: member.id,
      voteType: 'LIKE',
      postId,
    }).value as PostVote;

    const post = Post.create({
      memberId: member.id,
      postTitle: PostTitle.create('Post title').value as PostTitle,
      postText: PostText.create('Post text').value as PostText,
      postLikes: PostVotes.create([existingVote]),
      postId,
    }).value as Post;

    const likePostResult = new BlogService().likePost(post, member,
        [
      existingVote,
    ]);

    expect(likePostResult.isSuccess).toEqual(true);
    expect(post.likes.getItems().length).toEqual(0);
  });
});
```

ПЕРЕЛІК ПОСИЛАНЬ

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

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

3. 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 05.05.2021).

4. Швабер, К. Сазерленд Д. Руководство по скраму. Исчерпывающее руководство по скраму: правила игры. - URL: https://brainrain.com.ua/scrum-guide/ (дата звернення 05.05.2021).

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

6. Рубин Кеннет С. Основы Scrum: практическое руководство по гибкой разработке ПО. М.: ООО «И.Д.Вильямс», 2016. 544 с.

7. Московко С.Г., Богач І.В. Аналіз архітектурних шаблонів програмного забезпечення. *L Науково-технічна конференція факультету комп'ютерних систем і автоматики* : 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 0.5.05.2021).

8. How to build quality software solutions using TDD and BDD? - URL: https://y-sbm.com/blog/difference-between-tdd-and-bdd (дата звернення 0.5.05.2021).

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

10. What is Test Driven Development (TDD)? - URL: https://www.guru99.com/test-driven-development.html (дата звернення 0.5.05.2021).

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

12. Behavior Driven Development (BDD). - URL: https://www.agilealliance.org/glossary/bdd/ (дата звернення 0.5.05.2021).

13. Behavior Driven Development: Alternative to the Waterfall Approach. - URL: https://www.agilest.org/devops/behavior-driven-development/ (дата звернення 0.5.05.2021).

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

15. Continuous Integration. - URL: https://www.agilest.org/devops/continuous-integration/ (дата звернення 0.5.05.2021).

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

17. 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 0.5.05.2021).

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

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

20. Schwaber K. Scrum Development Process. *OOPSLA Business Object Design and Implementation Workshop* : матеріали конференції ACM, електронні наукові видання., London : Springer, 1997. - URL:

http://damiantgordon.com/Methodologies/Papers/Business Object Design and Implementation.pdf (дата звернення 0.5.05.2021).

21. How to publish and handle Domain Events. - URL: http://www.kamilgrzybek.com/design/how-to-publish-and-handle-domain-events/ (дата звернення 0.5.05.2021).

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

23. What is Domain-Driven Design (DDD) | Pros & Cons. - URL: https://codezup.com/what-is-domain-driven-design-ddd-pros-cons/ (дата звернення 0.5.05.2021).

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

25. Business Logic Definition. - URL: http://wiki.c2.com/?BusinessLogicDefinition (дата звернення 0.5.05.2021).

26. User Stories with Examples and Template. - URL: https://www.atlassian.com/agile/project-management/user-stories (дата звернення 0.5.05.2021).

27. 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 0.5.05.2021).

28. The Node.js Event emitter. - URL: https://nodejs.dev/learn/the-nodejs-event-emitter (дата звернення 0.5.05.2021).

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

30. Инструменты автоматизации тестирования. - URL: http://ru.qatestlab.com/technologies/software-infrastructure/test-automation-tools/ (дата звернення: 05.05.2021).

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

32. Автоматизация тестирования REST API при помощи Postman и JavaScript. - URL: http://quality-lab.ru/test-automation-rest-api-using-postman-and-javascript/ (дата звернення: 05.05.2021).

33. Асанов П. Автоматизация функционального тестирования REST API: секреты, тонкости и подводные камни. - URL: https://sqadays.com/ru/talk/34635 (дата звернення: 05.05.2021).

Додатки

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

З А Т В Е Р Д Ж У Ю

Завідувач кафедри АІІТ

д.т.н., професор Квєтний Р. Н.

_____
(підпис)

«____» _____ 2021 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на бакалаврську дипломну роботу

Вдосконалення системи управління якості програмного забезпечення

08-02.БДР.000.09.000 ТЗ

Керівник бакалаврської дипломної роботи

к.т.н., Овчинников К. В.

«____» _____ 2021 р.

Розробив студент гр. 1АКІТ-17б

Московко С. Г.

«____» _____ 2021 р.

Вінниця ВНТУ 2021

Додаток А
(обов'язковий)
Технічне завдання на бакалаврську дипломну роботу

1 Підстава для проведення робіт

Підставою для виконання бакалаврської дипломної роботи на тему: «Вдосконалення системи управління якості програмного забезпечення» є наказ № 65 від 09.03.2021 р.

Термін виконання робіт:

початок 01.02.2021 р.
кінець 07.06.2021 р.

2 Мета та вихідні дані для проведення робіт

Метою бакалаврської дипломної роботи є дослідження методик предметно орієнтованого проектування на прикладі розробки веб-додатків.

Вихідними даними для проведення робіт є індивідуальне завдання на бакалаврську дипломну роботу від 01.02.2021 р.

3 Етапи виконання робіт

Виконавцем всіх перерахованих в даному розділі етапів є: студент групи **АКІТ-17б Московко Сергій Геннадійович** факультету комп'ютерних систем та автоматики Вінницького національного технічного університету, а замовником є: кафедра автоматизації та інтелектуальних інформаційних технологій.

№ Етапу	Зміст етапу	Строки виконання
Е1	Аналіз існуючих практик проектування програмного забезпечення	
Е2	Розробити багаторівневу архітектуру	
Е3	Розробити модель домену вибраної предметної області	
Е4	Розробити бізнес-правила додатку, що реалізують усі випадки використання системи	
Е5	Розробити адаптери, які перетворюють дані з формату, найбільш зручного для випадків використання та сутностей, у формат, найбільш зручного для якоїсь зовнішньої служби	
Е6	Реалізувати сервіси для взаємодії з зовнішніми службами	

4 Призначення і галузь застосування

Практики предметно-орієнтованого проектування застосовуються при проектуванні та розробки програмного забезпечення. Предметно-орієнтоване проектування базується на наступних цілях:

- основна увага проекту спрямована на основний домен та логіку домену;
- розташування складних конструкцій в моделі домену;
- започаткування творчої співпраці між технічними експертами та експертами предметної області з метою ітеративного вдосконалення концептуальної моделі, що розглядає певні проблеми домену.

5 Технічні дані

5.1 спроектована архітектура додатку;
5.2 модель домену предметної області;
5.3 модулі-сервіси взаємодії з зовнішніми службами;
5.4 діаграми відповідальності та об’єктної взаємодії;

6 Джерела розробки

6.1 Положення про бакалаврську 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-17б _____ Московко С. Г.

Додаток Б

(обов’язковий)

Діаграми бізнес-логіки та архітектури додатку

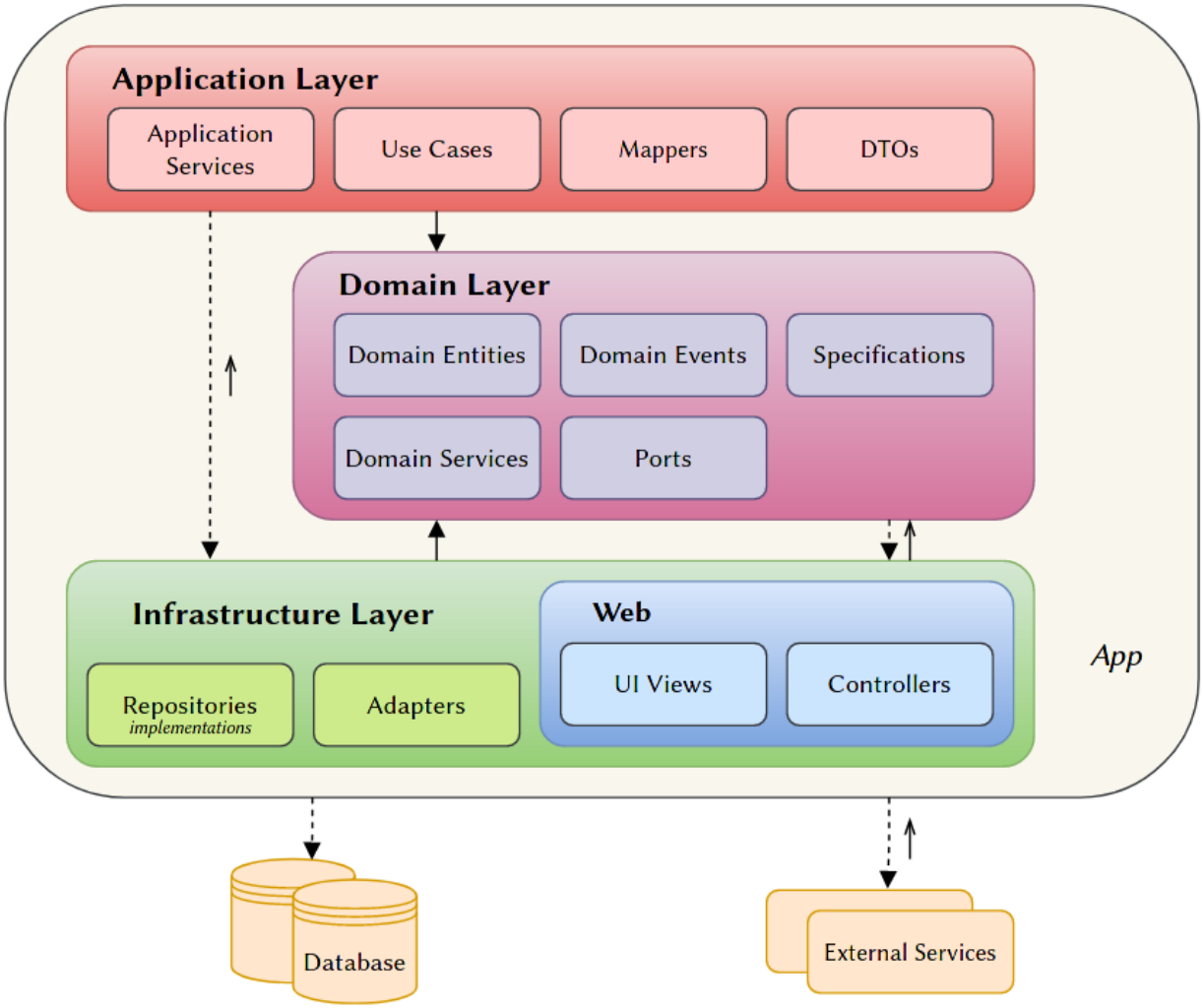

Рисунок Б.1 – Архітектура додатку

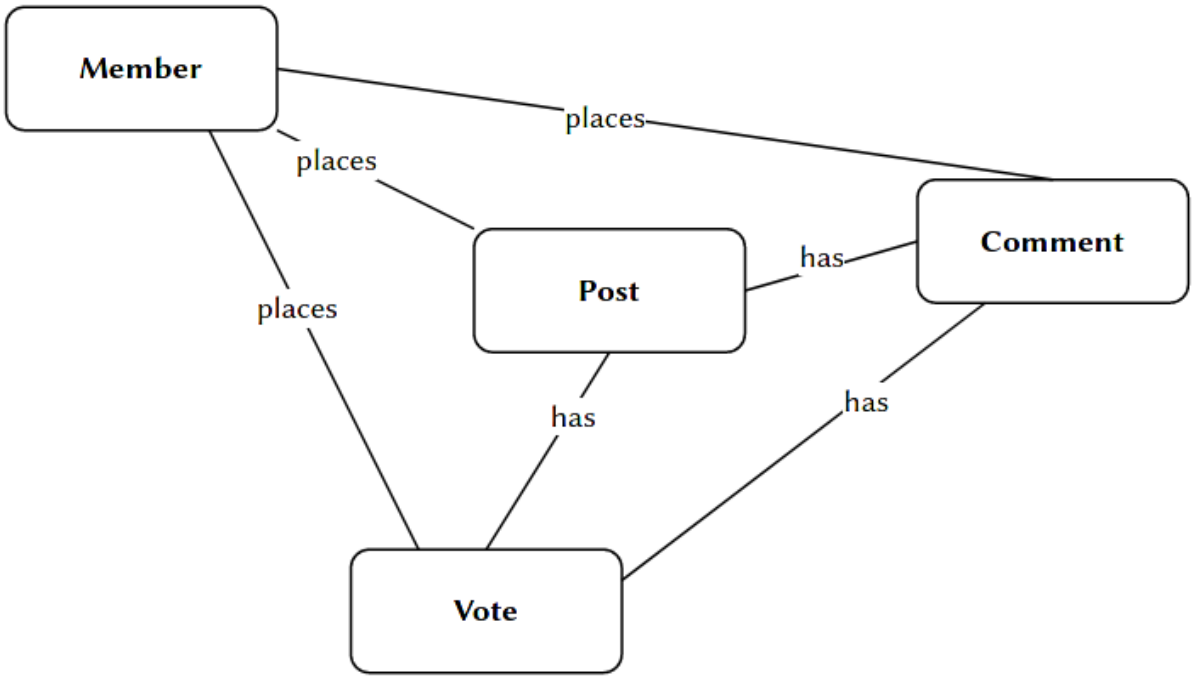

Рисунок Б.2 – Діаграма об’єктної взаємодії

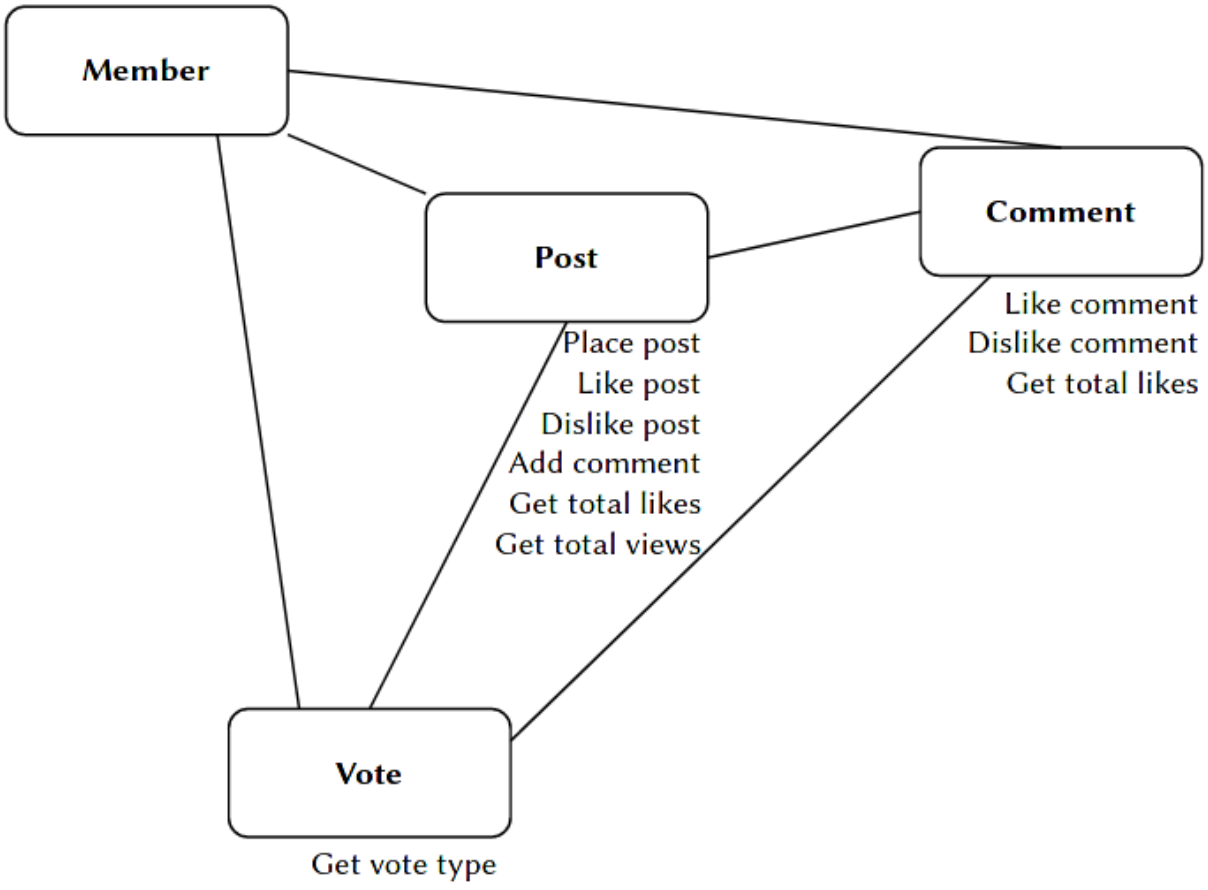

Рисунок Б.3 – Діаграма відповідальності об’єкта

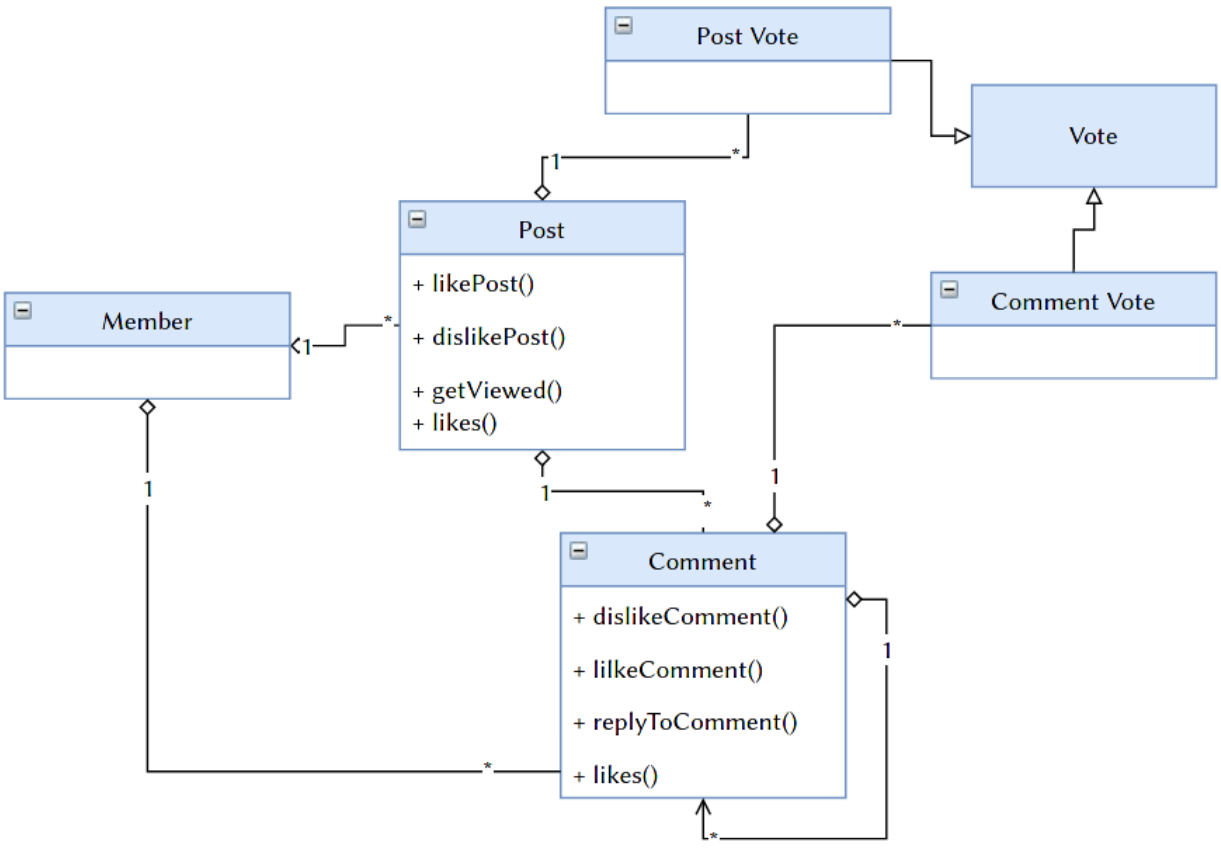

Рисунок Б.4 – UML діаграма класів бізнес-логіки

Додаток В

(обов'язковий)

Базові модулі системи

Базовий клас DomainEvents

```
export class DomainEvents {
  private static _eventEmitter: EventEmitter = new EventEmitter();
  private static _markedAggregates: AggregateRoot[] = [];

  public static register(
    callback: (event: DomainEvent) => void,
    eventClassName: string,
  ): void {
    this._eventEmitter.on(eventClassName, callback);
  }

  public static dispatch(event: DomainEvent): void {
    this._eventEmitter.emit(event.constructor.name, event);
  }

  public static dispatchAggregateEvents(aggregate: AggregateRoot):
     void {
    aggregate.domainEvents.forEach((event: DomainEvent) =>
      this.dispatch(event),
    );
  }

  public static dispatchEventsForAggregate(id: UniqueID): void {
    const aggregate = this.findMarkedAggregateByID(id);

    if (aggregate) {
      this.dispatchAggregateEvents(aggregate);
      aggregate.clearEvents();
```

```
      this.removeAggregateFromMarkedDispatchList(aggregate);
    }
  }

  public static findMarkedAggregateByID(id: UniqueID):
     AggregateRoot | null {
    return (
      this._markedAggregates.find(aggregateRoot =>
        aggregateRoot.id.equals(id),
      ) || null
    );
  }

  public static removeAggregateFromMarkedDispatchList(
    aggregate: AggregateRoot,
  ): void {
    const index = this._markedAggregates.findIndex(a => a.equals(
       aggregate));
    this._markedAggregates.splice(index, 1);
  }

  public static markAggregateForDispatch(aggregate: AggregateRoot):
      void {
    const foundAggregate = !!this.findMarkedAggregateByID(aggregate
       .id);

    if (!foundAggregate) {
      this._markedAggregates.push(aggregate);
    }
  }
}
```

Базовий клас UniqueID

```
export class UniqueID extends Identifier<string> {
  constructor(id: string) {
```

```
    super(id);
  }

  static create(id?: string): UniqueID {
    if (id) return new UniqueID(id);
    return new UniqueID(uuidv4());
  }
}
```

Базовий клас Identifier

```
export class Identifier<T> {
  constructor(private readonly _value: T) {}

  equals(id: Identifier<T>): boolean {
    return id.value === this.value;
  }

  toString(): string {
    return String(this.value);
  }

  get value(): T {
    return this._value;
  }
}
```

Базовий клас Entity

```
export abstract class Entity {
  constructor(private readonly _id: UniqueID) {}

  get id(): UniqueID {
    return this._id;
  }

  public equals(object: Entity): boolean {
```

```
    return object.id === this.id;
  }
}
```

Базовий клас AggregateRoot

```
export class AggregateRoot extends Entity {
  private readonly _domainEvents: DomainEvent[] = [];

  get domainEvents(): DomainEvent[] {
    return this._domainEvents;
  }

  protected addDomainEvent(domainEvent: DomainEvent): void {
    this.domainEvents.push(domainEvent);

    DomainEvents.markAggregateForDispatch(this);
  }

  public clearEvents(): void {
    this.domainEvents.splice(0, this.domainEvents.length);
  }
}
```

Базовий клас Result

```
export class Result<T> {
  public constructor(
    private readonly _isSuccess: boolean,
    private readonly _error: Error | null,
    private readonly _value?: T,
  ) {}

  get errorValue(): Error | null {
    return this._error;
  }
```

```
  get isSuccess(): boolean {
    return this._isSuccess;
  }

  get isFailure(): boolean {
    return !this.isSuccess;
  }

  get value(): T | undefined {
    if (this.isFailure) {
      throw new Error("Can't get the value of an error result.");
    }
    return this._value;
  }

  public static ok<U>(value?: U): Result<U> {
    const result = new Result<U>(true, null, value);
    Object.freeze(result);
    return result;
  }

  public static fail<U>(error: Error): Result<U> {
    const result = new Result<U>(false, error);
    Object.freeze(result);
    return result;
  }
}

export type Either<L, A> = Left<L, A> | Right<L, A>;

export class Left<L, A> {
  readonly value: L;

  constructor(value: L) {
    this.value = value;
```

```
  }

  isLeft(): this is Left<L, A> {
    return true;
  }

  isRight(): this is Right<L, A> {
    return false;
  }
}

export class Right<L, A> {
  readonly value: A;

  constructor(value: A) {
    this.value = value;
  }

  isLeft(): this is Left<L, A> {
    return false;
  }

  isRight(): this is Right<L, A> {
    return true;
  }
}

export const left = <L, A>(l: L): Either<L, A> => {
  return new Left(l);
};

export const right = <L, A>(a: A): Either<L, A> => {
  return new Right<L, A>(a);
};
```

Базовий клас AbstractSpecification

```
export interface Specification<T> {
  isSatisfiedBy(candidate: T): boolean;
  and(other: Specification<T>): Specification<T>;
  or(other: Specification<T>): Specification<T>;
  not(): Specification<T>;
}

export abstract class AbstractSpecification<T> implements
   Specification<T> {
  public abstract isSatisfiedBy(hand: T): boolean;

  public and(other: Specification<T>): Specification<T> {
    return new AndSpecification(this, other);
  }

  public or(other: Specification<T>): Specification<T> {
    return new OrSpecification(this, other);
  }

  public not(): Specification<T> {
    return new NotSpecification(this);
  }
}
```

Додаток Г

(обов'язковий)

Модулі бізнес рівня

Сутність Member

```
interface MemberProps {
  username: UserName;
  firstName: FirstName;
  lastName: LastName;
  email: Email;
  memberId?: UniqueID;
}

export class Member extends AggregateRoot {
  constructor(
    _id: UniqueID,
    private readonly _username: UserName,
    private readonly _email: Email,
    private readonly _firstName: FirstName,
    private readonly _lastName: LastName,
  ) {
    super(_id);
  }

  get username(): UserName {
    return this._username;
  }

  get email(): Email {
    return this._email;
  }

  get firstName(): FirstName {
```

```
    return this._firstName;
  }

  get lastName(): LastName {
    return this._lastName;
  }

  get fullName(): string {
    return `${this.firstName.value} ${this.lastName.value}`;
  }

  static create(props: MemberProps): Result<Member> {
    const propsGuardResult = Guard.againstNullOrUndefinedBulk([
      { argument: props.username, argumentName: 'username' },
      { argument: props.firstName, argumentName: 'firstName' },
      { argument: props.lastName, argumentName: 'lastName' },
      { argument: props.email, argumentName: 'email' },
    ]);

    if (!propsGuardResult.succeeded) {
      return Result.fail<Member>(new Error(propsGuardResult.message
          ));
    }

    const userIdResult = Guard.againstNullOrUndefined(props.
       memberId, 'userId');

    const member = new Member(
      userIdResult.succeeded ? (props.memberId as UniqueID) :
         UniqueID.create(),
      props.username,
      props.email,
      props.firstName,
      props.lastName,
    );
```

```
    if (!userIdResult.succeeded) {
      member.addDomainEvent(new MemberCreated(member));
    }

    return Result.ok<Member>(member);
  }
}
```

Сутність Post

```
export interface PostProps {
  postId?: UniqueID;
  memberId: UniqueID;
  postTitle: PostTitle;
  postText: PostText;
  postLikes?: PostVotes;
  comments?: Comments;
  hashtags?: Hashtags;
  datePosted?: Date;
  dateModified?: Date;
}

export interface EditPostProps {
  postTitle: PostTitle;
  postText: PostText;
  hashtags: Hashtags;
}

export class Post extends AggregateRoot {
  constructor(
    _id: UniqueID,
    private _postTitle: PostTitle,
    private _postText: PostText,
    private _hashtags: Hashtags,
    private readonly _memberId: UniqueID,
```

```
    private readonly _postLikes: PostVotes,
    private readonly _comments: Comments,
    private readonly _datePosted: Date,
    private _dateModified: Date,
  ) {
    super(_id);
  }

  get memberId(): UniqueID {
    return this._memberId;
  }

  get title(): PostTitle {
    return this._postTitle;
  }

  private updateTitle(title: PostTitle): Result<void> {
    this._postTitle = title;
    return Result.ok<void>();
  }

  get text(): PostText {
    return this._postText;
  }

  private updateText(text: PostText): Result<void> {
    this._postText = text;
    return Result.ok<void>();
  }

  get likes(): PostVotes {
    return this._postLikes;
  }

  get comments(): Comments {
```

```
    return this._comments;
  }

  get hashtags(): Hashtags {
    return this._hashtags;
  }

  private updateHashtags(hashtags: Hashtags): Result<void> {
    this._hashtags = hashtags;
    return Result.ok<void>();
  }

  private postModified(): Result<void> {
    this._dateModified = new Date();
    return Result.ok<void>();
  }

  get datePosted(): Date {
    return this._datePosted;
  }

  get dateModified(): Date {
    return this._dateModified;
  }

  public getViewed(): Result<void> {
    this.addDomainEvent(new PostViewed(this));
    return Result.ok<void>();
  }

  public getComment(comment: Comment): Result<void> {
    this.comments.add(comment);
    this.addDomainEvent(new CommentPosted(comment));
    return Result.ok<void>();
  }
```

```
public removeVote(vote: PostVote): Result<void> {
  this.likes.remove(vote);
  return Result.ok<void>();
}

public addVote(vote: PostVote): Result<void> {
  this.likes.add(vote);
  return Result.ok<void>();
}

public updateInfo(props: EditPostProps): Result<void> {
  const propsResult = Guard.againstNullOrUndefinedBulk([
    { argument: props.postTitle, argumentName: 'postTitle' },
    { argument: props.postText, argumentName: 'postText' },
    { argument: props.hashtags, argumentName: 'hashtags' },
  ]);

  if (!propsResult.succeeded) {
    return Result.fail<void>(new Error(propsResult.message));
  }

  const previousPostState = Object.assign({}, this);

  this.updateTitle(props.postTitle);
  this.updateText(props.postText);
  this.updateHashtags(props.hashtags);

  this.postModified();

  this.addDomainEvent(new PostEdited(previousPostState, this));

  return Result.ok<void>();
}
```

```
private removeCommentIfExists(comment: Comment): Result<void> {
  if (this.comments.exists(comment)) {
    this.comments.remove(comment);
  }

  return Result.ok<void>();
}

public updateComment(comment: Comment): Result<void> {
  this.removeCommentIfExists(comment);
  this.comments.add(comment);
  return Result.ok<void>();
}

static create(props: PostProps): Result<Post> {
  const propsResult = Guard.againstNullOrUndefinedBulk([
    { argument: props.memberId, argumentName: 'memberId' },
    { argument: props.postTitle, argumentName: 'postTitle' },
    { argument: props.postText, argumentName: 'postText' },
  ]);

  if (!propsResult.succeeded) {
    return Result.fail<Post>(new Error(propsResult.message));
  }

  const postIdResult = Guard.againstNullOrUndefined(props.postId,
      'postId');
  const postLikesResult = Guard.againstNullOrUndefined(
    props.postLikes,
    'postLikes',
  );
  const commentsResult = Guard.againstNullOrUndefined(
    props.comments,
    'comments',
  );
```

```
const hashtagsResult = Guard.againstNullOrUndefined(
  props.hashtags,
  'hashtags',
);
const datePostedResult = Guard.againstNullOrUndefined(
  props.datePosted,
  'datePosted',
);
const dateModifiedResult = Guard.againstNullOrUndefined(
  props.dateModified,
  'dateModified',
);

const post = new Post(
  postIdResult.succeeded ? (props.postId as UniqueID) :
     UniqueID.create(),
  props.postTitle,
  props.postText,
  hashtagsResult.succeeded
    ? (props.hashtags as Hashtags)
    : Hashtags.create([]),
  props.memberId,
  postLikesResult.succeeded
    ? (props.postLikes as PostVotes)
    : PostVotes.create([]),
  commentsResult.succeeded
    ? (props.comments as Comments)
    : Comments.create([]),
  datePostedResult.succeeded ? (props.datePosted as Date) : new
      Date(),
  dateModifiedResult.succeeded ? (props.dateModified as Date) :
      new Date(),
);

if (!postIdResult.succeeded) {
```

```
      post.addDomainEvent(new PostCreated(post));
    }

    return Result.ok<Post>(post);
  }
}
```

Сутність Comment

```
interface CommentProps {
  commentId?: UniqueID;
  memberId: UniqueID;
  postId: UniqueID;
  text: CommentText;
  likes?: CommentVotes;
  parentComentId?: UniqueID;
}

export class Comment extends Entity {
  constructor(
    _id: UniqueID,
    private readonly _memberId: UniqueID,
    private readonly _postId: UniqueID,
    private readonly _text: CommentText,
    private readonly _likes: CommentVotes,
    private readonly _parentComentId?: UniqueID,
  ) {
    super(_id);
  }

  get memberId(): UniqueID {
    return this._memberId;
  }

  get postId(): UniqueID {
    return this._postId;
```

```
}

get parentComentId(): UniqueID | undefined {
  return this._parentComentId;
}

get text(): CommentText {
  return this._text;
}

get likes(): CommentVotes {
  return this._likes;
}

public removeVote(vote: CommentVote): Result<void> {
  this.likes.remove(vote);
  return Result.ok<void>();
}

public addVote(vote: CommentVote): Result<void> {
  this.likes.add(vote);
  return Result.ok<void>();
}

static create(props: CommentProps): Result<Comment> {
  const guardPropsResult = Guard.againstNullOrUndefinedBulk([
    { argument: props.text, argumentName: 'text' },
    { argument: props.postId, argumentName: 'postId' },
    { argument: props.memberId, argumentName: 'memberId' },
  ]);

  if (!guardPropsResult.succeeded) {
    return Result.fail<Comment>(new Error(guardPropsResult.
      message));
  }
```

```
    const guardLikesResult = Guard.againstNullOrUndefined(props.
       likes, 'likes');
    const guardIdResult = Guard.againstNullOrUndefined(
      props.commentId,
      'commentId',
    );

    return Result.ok<Comment>(
      new Comment(
        guardIdResult.succeeded
          ? (props.commentId as UniqueID)
          : UniqueID.create(),
        props.memberId,
        props.postId,
        props.text,
        guardLikesResult.succeeded
          ? (props.likes as CommentVotes)
          : CommentVotes.create([]),
        props.parentComentId,
      ),
    );
  }
}
```

Сервіс BlogService

```
export class BlogService {
  public dislikeComment(
    post: Post,
    member: Member,
    comment: Comment,
    commentVotesByMember: CommentVote[],
  ) {
    const existingDislike: CommentVote | undefined =
       commentVotesByMember.find(
```

```
    (vote: CommentVote) => {
      return vote.isDislike();
    },
  );

  const dislikeAlreadyExists = !!existingDislike;
  if (dislikeAlreadyExists) {
    comment.removeVote(existingDislike as CommentVote);
    post.updateComment(comment);
    return Result.ok<void>();
  }

  const existingLike: CommentVote | undefined =
     commentVotesByMember.find(
    (vote: CommentVote) => {
      return vote.isLike();
    },
  );

  const likeAlreadyExists = !!existingLike;
  if (likeAlreadyExists) {
    comment.removeVote(existingLike as CommentVote);
    post.updateComment(comment);
  }

  const dislikeOrError: Result<CommentVote> = CommentVote.create
     ({
    commentId: comment.id,
    memberId: member.id,
    voteType: CommentVote.DISLIKE,
  });

  if (dislikeOrError.isFailure) {
    return Result.fail<void>(dislikeOrError.errorValue as Error);
  }
```

```
  const dislike: CommentVote = dislikeOrError.value as
     CommentVote;
  comment.addVote(dislike);
  post.updateComment(comment);
  return Result.ok<void>();
}

public likeComment(
  post: Post,
  member: Member,
  comment: Comment,
  commentVotesByMember: CommentVote[],
) {
  const existingLike: CommentVote | undefined =
     commentVotesByMember.find(
    (vote: CommentVote) => {
      return vote.isLike();
    },
  );

  const likeAlreadyExists = !!existingLike;
  if (likeAlreadyExists) {
    comment.removeVote(existingLike as CommentVote);
    post.updateComment(comment);
    return Result.ok<void>();
  }

  const existingDislike: CommentVote | undefined =
     commentVotesByMember.find(
    (vote: CommentVote) => {
      return vote.isDislike();
    },
  );
```

```
  const dislikeAlreadyExists = !!existingDislike;
  if (dislikeAlreadyExists) {
    comment.removeVote(existingDislike as CommentVote);
    post.updateComment(comment);
  }

  const likeOrError: Result<CommentVote> = CommentVote.create({
    commentId: comment.id,
    memberId: member.id,
    voteType: CommentVote.LIKE,
  });

  if (likeOrError.isFailure) {
    return Result.fail<void>(likeOrError.errorValue as Error);
  }

  const like: CommentVote = likeOrError.value as CommentVote;
  comment.addVote(like);
  post.updateComment(comment);
  return Result.ok<void>();
}

public dislikePost(
  post: Post,
  member: Member,
  postVotesByMember: PostVote[],
): Result<void> {
  const existingDislike: PostVote | undefined = postVotesByMember
     .find(
    (vote: PostVote) => {
      return vote.isDislike();
    },
  );

  const dislikeAlreadyExists = !!existingDislike;
```

```
  if (dislikeAlreadyExists) {
    post.removeVote(existingDislike as PostVote);
    return Result.ok<void>();
  }

  const existingLike: PostVote | undefined = postVotesByMember.
     find(
    (vote: PostVote) => {
      return vote.isLike();
    },
  );

  const likeAlreadyExists = !!existingLike;

  if (likeAlreadyExists) {
    post.removeVote(existingLike as PostVote);
  }

  const dislikeOrError: Result<PostVote> = PostVote.create({
    postId: post.id,
    memberId: member.id,
    voteType: PostVote.DISLIKE,
  });

  if (dislikeOrError.isFailure) {
    return Result.fail<void>(dislikeOrError.errorValue as Error);
  }

  const dislike: PostVote = dislikeOrError.value as PostVote;

  post.addVote(dislike);

  return Result.ok<void>();
}
```

```
public likePost(
  post: Post,
  member: Member,
  postVotesByMember: PostVote[],
): Result<void> {
  const existingLike: PostVote | undefined = postVotesByMember.
     find(
    (vote: PostVote) => {
      return vote.isLike();
    },
  );

  const likeAlreadyExists = !!existingLike;

  if (likeAlreadyExists) {
    post.removeVote(existingLike as PostVote);
    return Result.ok<void>();
  }

  const existingDislike: PostVote | undefined = postVotesByMember
     .find(
    (vote: PostVote) => {
      return vote.isDislike();
    },
  );

  const dislikeAlreadyExists = !!existingDislike;

  if (dislikeAlreadyExists) {
    post.removeVote(existingDislike as PostVote);
  }

  const likeOrError: Result<PostVote> = PostVote.create({
    postId: post.id,
```

```
      memberId: member.id,
      voteType: PostVote.LIKE,
    });

    if (likeOrError.isFailure) {
      return Result.fail<void>(likeOrError.errorValue as Error);
    }

    const like: PostVote = likeOrError.value as PostVote;

    post.addVote(like);

    return Result.ok<void>();
  }
}
```

Валідатор PostTitleValidator

```
export class PostTitleValidator implements Validator<string> {
  public readonly errorMessage = 'Post title is not valid';
  private readonly rules: Specification<string>[] = [
    new ShorterThanSpec(PostTitle.maxLength),
    new LongerThanSpec(PostTitle.minLength),
  ];

  public isValid(postTitle: string): boolean {
    return this.brokenRules(postTitle).length === 0;
  }

  public brokenRules(str: string): Array<string> {
    return this.rules
      .filter(rule => !rule.isSatisfiedBy(str))
      .map(rule => rule.constructor.name);
  }
}
```

Подія PostCreated

```
export class PostCreated implements DomainEvent {
  private readonly _dateOccured: Date = new Date();

  constructor(private readonly _post: Post) {}

  get dateTimeOccurred(): Date {
    return this._dateOccured;
  }

  get post(): Post {
    return this._post;
  }

  public getAggregateId(): UniqueID {
    return this.post.id;
  }
}
```

Подія PostRated

```
export class PostRated implements DomainEvent {
  private readonly _dateOccured: Date = new Date();

  constructor(private readonly _vote: PostVote) {}

  get dateTimeOccurred(): Date {
    return this._dateOccured;
  }

  get postVote(): PostVote {
    return this._vote;
  }

  public getAggregateId(): UniqueID {
    return this.postVote.id;
  }
```

```
}
```

Обробник події PostCreatedHandler

```
export class PostCreatedHandler implements Handler {
  constructor(private readonly _loggers: Logger[]) {
    this.setupSubscriptions();
  }

  setupSubscriptions(): void {
    DomainEvents.register(
      this.handle.bind(this) as (event: DomainEvent) => Promise<
         void>,
      PostCreated.name,
    );
  }

  async handle(event: PostCreated): Promise<void> {
    this._loggers.forEach(async logger => {
      logger.log(
        `[${event.dateTimeOccurred}][${PostCreated.name}] New post
           created.`,
      );
    });
  }
}
```

Обробник події PostRatedHandler

```
export class PostRatedHandler implements Handler {
  constructor(
    private readonly _emailService: EmailServicePort,
    private readonly _memberRepo: MemberRepoPort,
  ) {
    this.setupSubscriptions();
  }
```

```
  setupSubscriptions(): void {
    DomainEvents.register(
      this.handle.bind(this) as (event: DomainEvent) => Promise<
        void>,
      PostRated.name,
    );
  }

  async handle(event: PostRated): Promise<void> {
    const member: Member = await this._memberRepo.getMemberById(
      event.postVote.memberId,
    );

    this._emailService.sendEmail(
      member.email.value,
      'Member ${member.fullName} rated your post.',
    );
  }
}
```

Додаток Д

(обов'язковий)

Модулі рівня додатку

Сервіс CreatePostService

```
export class CreatePostService implements CreatePostUseCase {
  constructor(
    private readonly _postRepo: PostRepoPort,
    private readonly _memberRepo: MemberRepoPort,
  ) {}

  async createPost(dto: CreatePostDTO): Promise<CreatePostResponse>
      {
    const postProps = {} as PostProps;

    try {
      const member = await this._memberRepo.getMemberById(
        UniqueID.create(dto.memberId),
      );
      postProps.memberId = member.id;
    } catch (error) {
      return Result.fail<Post>(error);
    }

    const titleResult = PostTitle.create(dto.title);
    if (titleResult.isFailure) {
      return Result.fail<Post>(titleResult.errorValue as Error);
    }
    postProps.postTitle = titleResult.value as PostTitle;

    const textResult = PostText.create(dto.text);
    if (textResult.isFailure) {
      return Result.fail<Post>(textResult.errorValue as Error);
```

```
    }
    postProps.postText = textResult.value as PostText;

    const postResult = Post.create(postProps);
    if (postResult.isFailure) {
      return Result.fail<Post>(postResult.errorValue as Error);
    }

    const post = postResult.value as Post;

    try {
      await this._postRepo.savePost(post);
    } catch (error) {
      return Result.fail<Post>(error);
    }

    return Result.ok<Post>(post);
  }
}
```

Сервіс DeletePostService

```
export class DeletePostService implements DeletePostUseCase {
  constructor(private readonly _postRepo: PostRepoPort) {}

  async deletePost(dto: DeletePostDTO): Promise<DeletePostResponse>
      {
    try {
      await this._postRepo.deletePost(UniqueID.create(dto.postId));
      return Result.ok<void>();
    } catch (error) {
      return Result.fail<void>(error);
    }
  }
}
```

Сервіс EditPostService

```
export class EditPostService implements EditPostUseCase {
  constructor(private readonly _postRepo: PostRepoPort) {}

  async editPost(dto: EditPostDTO): Promise<EditPostResponse> {
    const editPostProps = {} as EditPostProps;

    const titleResult = PostTitle.create(dto.title);
    if (titleResult.isFailure) {
      return Result.fail<Post>(titleResult.errorValue as Error);
    }

    editPostProps.postTitle = titleResult.value as PostTitle;

    const textResult = PostText.create(dto.text);
    if (textResult.isFailure) {
      return Result.fail<Post>(textResult.errorValue as Error);
    }

    editPostProps.postText = textResult.value as PostText;

    editPostProps.hashtags = Hashtags.create([]);

    try {
      const post = await this._postRepo.getPostById(
        UniqueID.create(dto.postId),
      );

      const updateInfoResult = post.updateInfo(editPostProps);

      if (updateInfoResult.isFailure) {
        return Result.fail<Post>(updateInfoResult.errorValue as
          Error);
      }

      await this._postRepo.savePost(post);
```

```
      return Result.ok<Post>(post);
    } catch (error) {
      return Result.fail<Post>(error);
    }
  }
}
```

Сервіс GetPostService

```
export class GetPostService implements GetPostUseCase {
  constructor(private readonly _postRepo: PostRepoPort) {}

  async getPost(dto: GetPostDTO): Promise<GetPostResponse> {
    try {
      const post = await this._postRepo.getPostById(
        UniqueID.create(dto.postId),
      );
      return Result.ok<Post>(post);
    } catch (error) {
      return Result.fail<Post>(error);
    }
  }
}
```

Сервіс GetAllPostsService

```
export class GetAllPostsService implements GetAllPostsUseCase {
  constructor(private readonly _postRepo: PostRepoPort) {}

  async getAllPosts(): Promise<GetAllPostsResponse> {
    try {
      const posts = await this._postRepo.getAllPosts();
      return Result.ok<Post[]>(posts as Post[]);
    } catch (error) {
      return Result.fail<Post[]>(error);
    }
```

```
  }
}
```

Сервіс LikePostService

```
export class LikePostService implements LikePostUseCase {
  constructor(
    private readonly _postRepo: PostRepoPort,
    private readonly _memberRepo: MemberRepoPort,
    private readonly _postVoteRepo: PostVoteRepo,
    private readonly _blogService: BlogService,
  ) {}

  async likePost(dto: LikePostDTO): Promise<LikePostResponse> {
    try {
      const member = await this._memberRepo.getMemberById(
        UniqueID.create(dto.memberId),
      );

      const post = await this._postRepo.getPostById(
        UniqueID.create(dto.postId),
      );

      const postVotesByMember = await this._postVoteRepo.
         getPostVotesByMemberId(
        post.id,
        member.id,
      );

      const likePostResult = this._blogService.likePost(
        post,
        member,
        postVotesByMember,
      );

      if (likePostResult.isFailure) {
```

```
        return Result.fail<void>(likePostResult.errorValue as Error
          );
      }

      await this._postRepo.savePost(post);

      return Result.ok<void>();
    } catch (error) {
      return Result.fail<void>(error);
    }
  }
}
```

Сервіс DislikePostService

```
export class DislikePostService implements DislikePostUseCase {
  constructor(
    private readonly _postRepo: PostRepoPort,
    private readonly _memberRepo: MemberRepoPort,
    private readonly _postVoteRepo: PostVoteRepo,
    private readonly _blogService: BlogService,
  ) {}

  async dislikePost(dto: DislikePostDTO): Promise<
     DislikePostResponse> {
    try {
      const member = await this._memberRepo.getMemberById(
        UniqueID.create(dto.memberId),
      );

      const post = await this._postRepo.getPostById(
        UniqueID.create(dto.postId),
      );

      const postVotesByMember = await this._postVoteRepo.
         getPostVotesByMemberId(
```

```
        post.id,
        member.id,
      );

      const dislikePostResult = this._blogService.dislikePost(
        post,
        member,
        postVotesByMember,
      );

      if (dislikePostResult.isFailure) {
        return Result.fail<void>(dislikePostResult.errorValue as
          Error);
      }

      await this._postRepo.savePost(post);

      return Result.ok<void>();
    } catch (error) {
      return Result.fail<void>(error);
    }
  }
}
```

Мапер PostMapper

```
export class PostMapper implements Mapper<PostDTO, Post> {
  public toDomain(dto: PostDTO): Post {
    return Post.create({
      postId: UniqueID.create(dto.postId),
      memberId: UniqueID.create(dto.memberId),
      postTitle: PostTitle.create(dto.postTitle).value as PostTitle
        ,
      postText: PostText.create(dto.postText).value as PostText,
      postLikes: PostVotes.create(
        dto.postLikes.map(new PostVoteMapper().toDomain),
```

```
      ),
      comments: Comments.create(dto.comments.map(new CommentMapper
        ().toDomain)),
      datePosted: dto.datePosted,
      dateModified: dto.dateModified,
    }).value as Post;
  }

  public toPersistence(domain: Post): PostDTO {
    return {
      postId: domain.id.value,
      memberId: domain.memberId.value,
      postTitle: domain.title.value,
      postText: domain.text.value,
      postLikes: domain.likes
        .getItems()
        .map(new PostVoteMapper().toPersistence),
      comments: domain.comments
        .getItems()
        .map(new CommentMapper().toPersistence),
      datePosted: domain.datePosted,
      dateModified: domain.dateModified,
    };
  }
}
```

Маπер MemberMapper

```
export class MemberMapper implements Mapper<MemberDTO, Member> {
  public toDomain(dto: MemberDTO): Member {
    return Member.create({
      memberId: UniqueID.create(dto.id.toString()),
      username: UserName.create(dto.username).value as UserName,
      firstName: FirstName.create(dto.firstName).value as FirstName
        ,
      lastName: LastName.create(dto.lastName).value as LastName,
```

```
      email: Email.create(dto.email).value as Email,
    }).value as Member;
  }

  public toPersistence(domain: Member): MemberDTO {
    return {
      id: parseInt(domain.id.value, 10),
      username: domain.username.value,
      firstName: domain.firstName.value,
      lastName: domain.lastName.value,
      email: domain.email.value,
    };
  }
}
```

Маπер CommentMapper

```
export class CommentMapper implements Mapper<CommentDTO, Comment> {
  toDomain(dto: CommentDTO): Comment {
    const commentProps = {
      commentId: UniqueID.create(dto.commentId),
      memberId: UniqueID.create(dto.memberId),
      postId: UniqueID.create(dto.postId),
      text: CommentText.create(dto.text).value as CommentText,
      likes: CommentVotes.create(
        dto.commentVotes.map(new CommentVoteMapper().toDomain),
      ),
    };

    if (dto.parentComentId) {
      Object.assign(commentProps, {
        parentComentId: UniqueID.create(dto.parentComentId),
      });
    }

    return Comment.create(commentProps).value as Comment;
```

```
  }

  toPersistence(domain: Comment): CommentDTO {
    const dto = {
      commentId: domain.id.value,
      memberId: domain.memberId.value,
      postId: domain.postId.value,
      text: domain.text.value,
      commentVotes: domain.likes
        .getItems()
        .map(new CommentVoteMapper().toPersistence),
    };

    if (domain.parentComentId) {
      Object.assign(dto, { parentComentId: domain.parentComentId.
         value });
    }

    return dto;
  }
}
```

Додаток Е

(обов’язковий)

Модулі рівня інфраструктури

Репозиторій MemberRepo

```
interface MemberData extends MemberDTO, RowDataPacket {}

export class MemberRepo implements MemberRepoPort {
  constructor(private readonly _connection: Connection) {}

  public async getAllMembers(): Promise<Member[]> {
    const [rows]: [
      Array<MemberData>,
      Array<FieldPacket>,
    ] = await this._connection.promise().execute('SELECT * FROM `
       Members``, []);

    return rows.map(new MemberMapper().toDomain);
  }

  public async getMemberById(id: UniqueID): Promise<Member> {
    const [rows]: [
      Array<MemberData>,
      Array<FieldPacket>,
    ] = await this._connection
      .promise()
      .execute('SELECT * FROM `Members` WHERE `id` = ?', [id.
         toString()]);

    if (rows.length === 0) {
      throw new NotFoundError('Member');
    }
```

```
    return rows.map(new MemberMapper().toDomain)[0];
  }

  public async deleteMember(id: UniqueID): Promise<void> {
    this._connection
      .promise()
      .execute('DELETE FROM `Members` WHERE `id` = ?', [id.toString
        ()]);
  }

  public async exists(id: UniqueID): Promise<boolean> {
    try {
      await this.getMemberById(id);
    } catch (error) {
      if (error instanceof NotFoundError) {
        return false;
      }
      throw error;
    }

    return true;
  }

  public async saveMember(member: Member): Promise<void> {
    const exists = await this.exists(member.id);

    if (exists) {
      await this._connection
        .promise()
        .execute(
          'UPDATE `Members` SET username = ?, firstName = ?,
            lastName = ?, email = ? WHERE id = ?',
          [
            member.username.value,
            member.firstName.value,
```

```
            member.lastName.value,
            member.email.value,
            member.id.toString(),
          ],
        );

      DomainEvents.dispatchEventsForAggregate(member.id);

      return;
    }

    await this._connection
      .promise()
      .execute(
        'INSERT INTO 'Members' (username,firstNmae,lastName,email)
           VALUES (?,?,?,?)',
        [
          member.username.value,
          member.firstName.value,
          member.lastName.value,
          member.email.value,
        ],
      );

    DomainEvents.dispatchEventsForAggregate(member.id);
  }
}
```

Репозиторій PostRepo

```
interface PostData extends PostDTO, RowDataPacket {}

export class PostRepo implements PostRepoPort {
  constructor(private readonly _connection: Connection) {}

  public async getAllPosts(): Promise<Post[]> {
```

```
    const [rows]: [
      Array<PostData>,
      Array<FieldPacket>,
    ] = await this._connection.promise().execute('SELECT * FROM `
       Posts``, []);

    return rows.map(new PostMapper().toDomain);
  }

  public async getPostById(id: UniqueID): Promise<Post> {
    const [rows]: [
      Array<PostData>,
      Array<FieldPacket>,
    ] = await this._connection
      .promise()
      .execute('SELECT * FROM `Posts` WHERE `id` = ?', [id.toString
         ()]);

    if (rows.length === 0) {
      throw new NotFoundError('Post');
    }

    return rows.map(new PostMapper().toDomain)[0];
  }

  public async deletePost(id: UniqueID): Promise<void> {
    await this._connection
      .promise()
      .execute('DELETE FROM `Posts` WHERE `id` = ?', [id.toString()
         ]);
  }

  public async exists(id: UniqueID): Promise<boolean> {
    try {
      await this.getPostById(id);
```

```
    } catch (error) {
      if (error instanceof NotFoundError) {
        return false;
      }
      throw error;
    }

    return true;
  }

  public async savePost(post: Post): Promise<void> {
    const exists = await this.exists(post.id);

    if (exists) {
      await this._connection
        .promise()
        .execute('UPDATE `Posts` SET title = ?, text = ?,
           dateModified = ?', [
          post.title.value,
          post.text.value,
          `${
            post.dateModified.getFullYear
          }-${post.dateModified.getMonth()}-${post.dateModified.
             getDay()}`,
        ]);

      DomainEvents.dispatchEventsForAggregate(post.id);

      return;
    }

    await this._connection
      .promise()
      .execute(
```

```
        'INSERT INTO 'Posts' (memberId,title,text,datePosted,
           dateModified) VALUES (?,?,?,?)',
        [
          post.memberId.toString(),
          post.title.value,
          post.text.value,
          '${
            post.datePosted.getFullYear
          }-${post.datePosted.getMonth()}-${post.datePosted.getDay
             ()}',
          '${
            post.dateModified.getFullYear
          }-${post.dateModified.getMonth()}-${post.dateModified.
             getDay()}',
        ],
      );

    DomainEvents.dispatchEventsForAggregate(post.id);
  }
}
```

Контролер CreatePostController

```
export class CreatePostController {
  constructor(private readonly _createPostService:
     CreatePostUseCase) {}

  async createPost(req: Request, res: Response): Promise<void> {
    const dto: CreatePostDTO = {
      memberId: req.body.memberId,
      title: req.body.title,
      text: req.body.text,
    };

    const response = await this._createPostService.createPost(dto);
```

```
    if (response.isFailure) {
      throw response.errorValue;
    }

    res.status(200);
  }
}
```

Контролер DeletePostController

```
export class DeletePostController {
  constructor(private readonly _deletePostService:
     DeletePostUseCase) {}

  async deletePost(req: Request, res: Response): Promise<void> {
    const dto: DeletePostDTO = {
      postId: req.body.postId,
    };

    const response = await this._deletePostService.deletePost(dto);

    if (response.isFailure) {
      throw response.errorValue;
    }

    res.status(200);
  }
}
```

Модуль server.ts

```
import express from 'express';
import bodyParser from 'body-parser';
import helmet from 'helmet';
import mysql from 'mysql2';
```

```
import { CreatePostController } from './controllers/create-post.
   controller';
import { PostRepo } from '../data/mysql/post.repo';
import { MemberRepo } from '../data/mysql/member.repo';
import { CreatePostService } from '../../application/services/
   create-post.service';
import { DeletePostService } from '../../application/services/
   delete-post.service';
import { DeletePostController } from './controllers/delete-post.
   controller';

const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(helmet());

const database = mysql.createConnection({
  host: 'localhost',
  user: 'serhii',
  password: '123',
  database: 'forum',
});

app.post(
  '/post',
  new CreatePostController(
    new CreatePostService(new PostRepo(database), new MemberRepo(
       database)),
  ).createPost,
);

app.delete(
  '/post',
```

```
  new DeletePostController(new DeletePostService(new PostRepo(
     database)))
    .deletePost,
);

const port = process.env.PORT || 3000;

app.listen(port, () => {
  console.log(‘Server started on port ${port}‘);
});
```