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].

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.

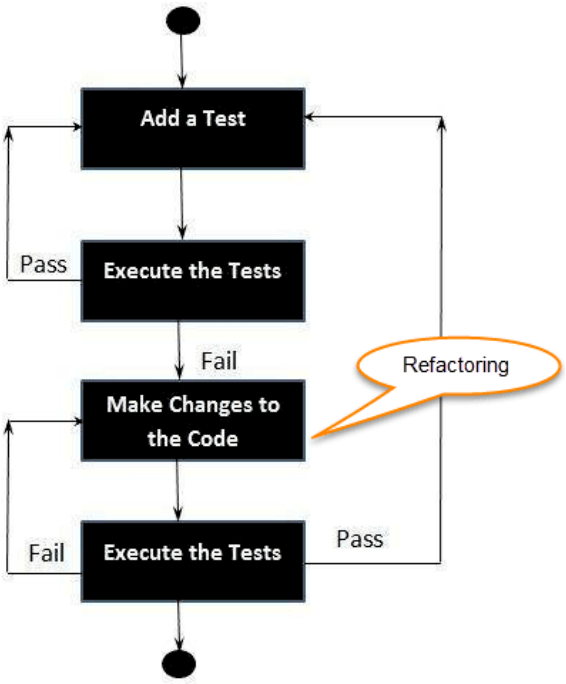

Рисунок 1.1 – Цикл керованою тестами розробки

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.

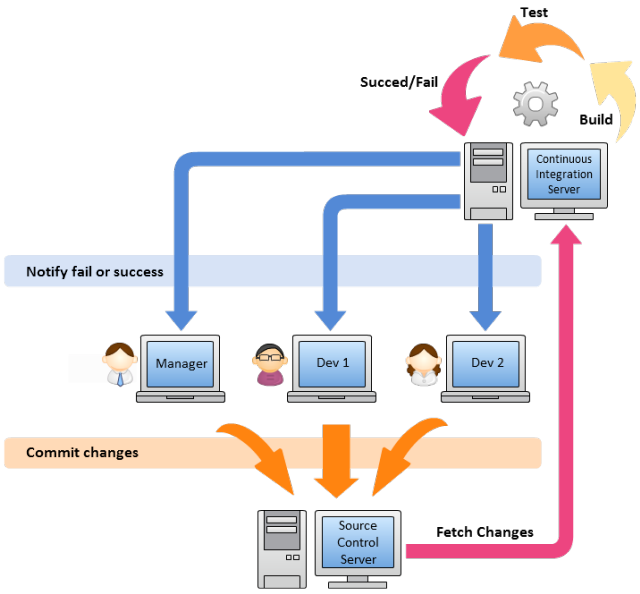

Рисунок 1.2 – Цикл роботи неперервної інтеграції

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

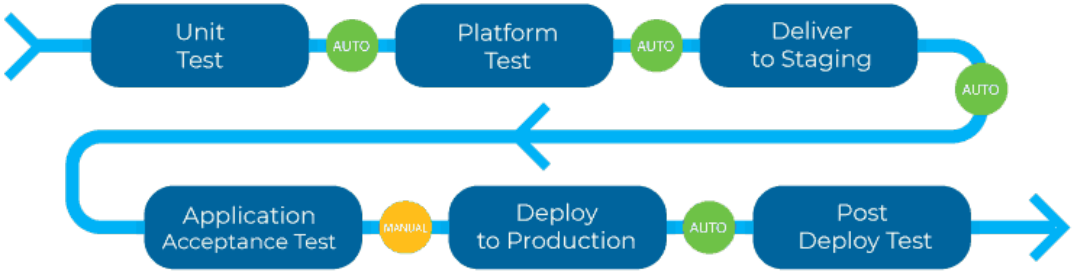

Рисунок 1.3 – Модель безперервної доставки

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 Огляд існуючих методологій проектування та розробки програмного забезпечення

1.2.1 DevOps

1.3 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, 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.

У книзі Domain-Driven Design [1], сформульований ряд концепцій і практик. Так, наприклад, особлива увага приділяється значенню загальної мови (Ubiquitous language). При проектуванні моделі предметної області необхідно сформувати спільну мову предметної області для опису вимог до системи, яка працює однаково добре як для бізнес-користувачів або спонсорів, так і для розробників програмного забезпечення. Ця мова означується експертами в обраній галузі. Книга зосереджена на описі доменного рівня, як одного із загальних рівнів в об’єктно-орієнтованій системі з багаторівневою архітектурою. У DDD є засоби для висловлення, створення та вилучення моделей предметної області:

– Сутність: Категорія індивідуальних об’єктів, які залишаються незмінними на різних етапах програми, для яких атрибути не грають великого значення, а послідовність та ідентичність, які поширюється в житті усієї системи називаються сутностями.

– Об’єкт значення: Об’єкт, який містить атрибути, але не має концептуальної ідентичності. Він повинен розглядатися як незмінний об’єкт.

– Сукупність: Колекція об’єктів, які пов’язані між собою завдяки головній сутності (Root Entity), інакше відомій як Aggregate root. Коренева сутність колекції об’єктів гарантує узгодженість змін, що вносяться до сукупності, забороняючи зовнішнім об’єктам посилатися на членів колекції.

– Доменна подія: Подія, яка сталася в певному домені, і фіксує пам’ять про нього. Це дає можливість повідомляти інші частини того самого домену про те, що сталося якась зміна, і ці інші частини потенційно можуть реагувати [18].

– Сервіс: Коли будь-яка операція концептуально не відноситься до будь-якого об’єкту, вона може бути реалізована в сервісі.

– Сховище: Отримання об’єктів предметної області повинно делегуватися в спеціалізовані сховища об’єктів. Це дає можливість підміняти місце збереження об’єктів.

– Фабрика: Створення об’єктів предметної області повинно бути делеговане до спеціалізованих фабрик. Це дає можливість підміняти реалізацію створення об’єктів.

Для того, щоб добре використовувати предметно-орієнтоване проектування, нам потрібно прийняти до уваги принципи SOLID [19], організувати рівень бізнес логіки в основі нашої багаторівневої архітектури, використовувати багато інверсії та впровадження залежностей, щоб підключити адаптери до рівня персистентності, веб та зовнішніх технологій.

Переваги використання предметно-орієнтованого проектування:

1. Загальнодоступна мова, полегшує спілкування між розробниками та представниками бізнесу, а також між самими розробниками.

2. Оскільки система була побудована для моделювання бізнес процесів, її, як правило, буде гнучкіше змінювати, оскільки нові функціональні вимоги цілком вписуються.

3. Завдяки способу побудови моделей домену з використанням інкапсуляції зрозумілості та хорошого розподілу моделей, доменні системи, як правило, є більш підтримуваною за своєю природою [20].

Для того, щоб допомогти зберегти модель як чисту та корисну конструкцію мови, команда, як правило, повинна здійснити велику кількість ізоляції та інкапсуляції в рамках доменної моделі. Отже, система, що базується на предметно-орієнтованому дизайні, може мати відносно високу вартість. Хоча предметно-орієнтоване проектування забезпечує багато технічних переваг, Microsoft рекомендує застосовувати його лише до складних доменів, де модель та лінгвістичні процеси дають чіткі переваги при передачі складної інформації та формулюванні загального розуміння домену.

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 - одна з архітектурних парадигм розробки ПЗ, при якій розбиття програми на самостійні складові частини відбувається по реалізованої ними функціональності [21].

Характерні особливості багаторівневої архітектури:

– необхідна функціональність реалізується в одному рівні і не дублюється в інших;

– кожен рівень повинен чітко реалізовувати ту функціональність, до області якої він відноситься, не поєднуючи код інших функціональних областей;

– організація передачі даних між рівнями через компоненти доступу до даних, далі через бізнес-логіку, з передачею через контролюючі сервіси;

– рівні слабко пов’язані між собою;

– кожен рівень агрегує залежності і абстракції рівня, розташованого безпосередньо під ним;

– фізично всі рівні можуть бути розгорнуті на одному комп’ютері або розподілені по різних комп’ютерах.

В логічно розділених на рівні архітектурах інформаційних систем найбільш часто зустрічаються наступні рівні:

– рівень сутностей (даних) містить всі сутності, що використовуються в проектах програми;

– рівень бізнес-логіки реалізує функціональні можливості програми;

– сервісний рівень дозволяє використовувати додаток зовнішнім сервісам та рівню подання;

– рівень користувальницького інтерфейсу (подання) надає ергономічний інтерфейс користувачу відповідно до функціоналу, описаному в технічному завданні;

– рівень загальних компонентів містить всі бібліотеки і функціональні можливості, які можуть бути використані в будь-якому із зазначених вище рівнів.

Спроектована архітектура додатку зображена на рисунку 2.1.

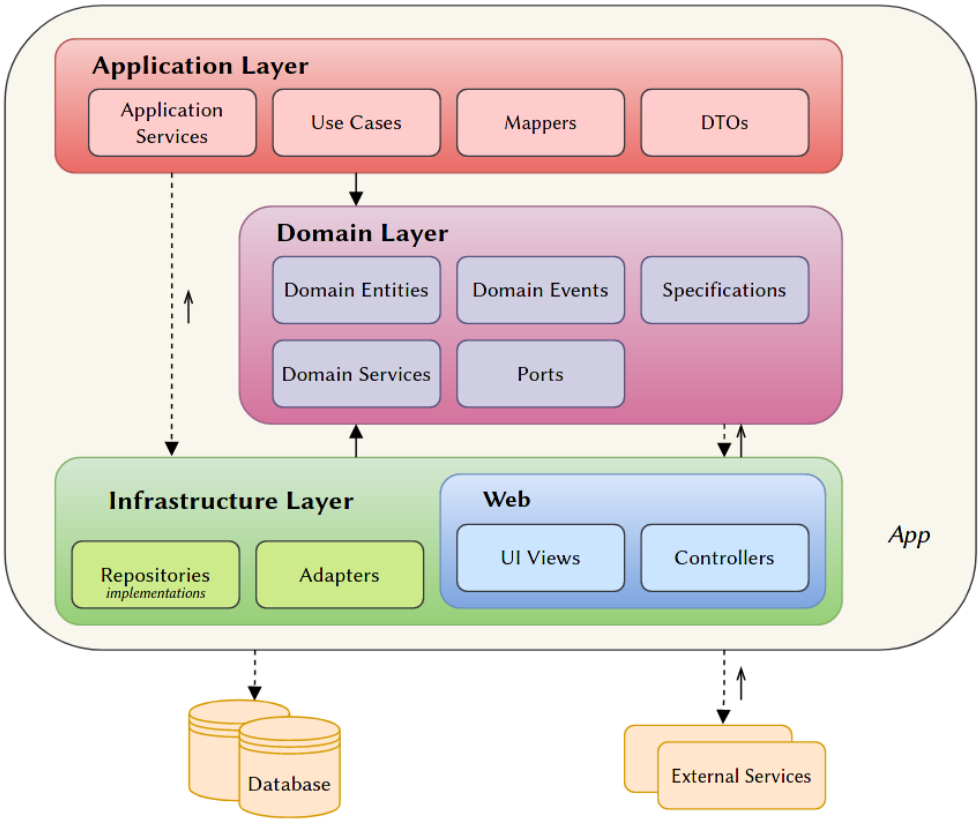

Рисунок 2.1 – Архітектура додатку

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 [22].

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 [23].

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 2.2)

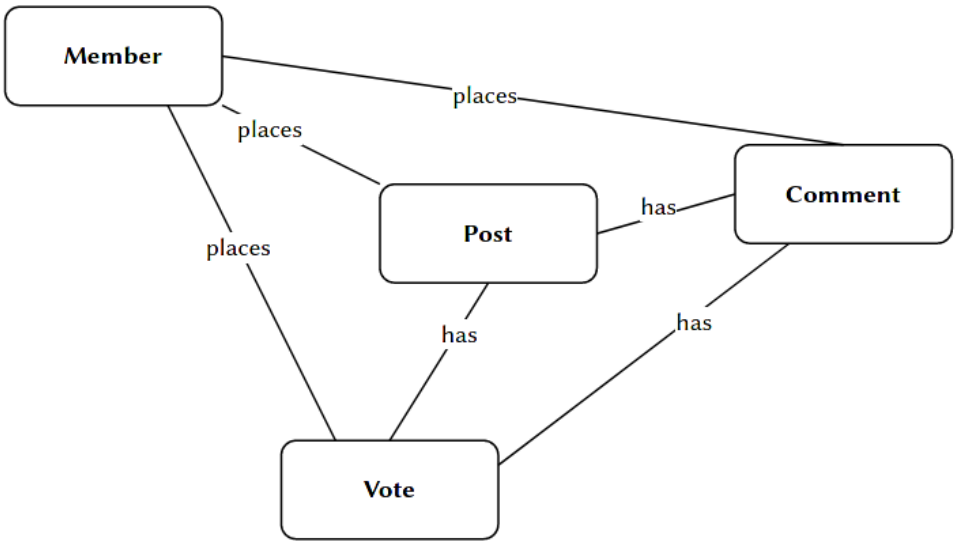

Рисунок 2.2 – Діаграма об’єктної взаємодії

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 2.3.

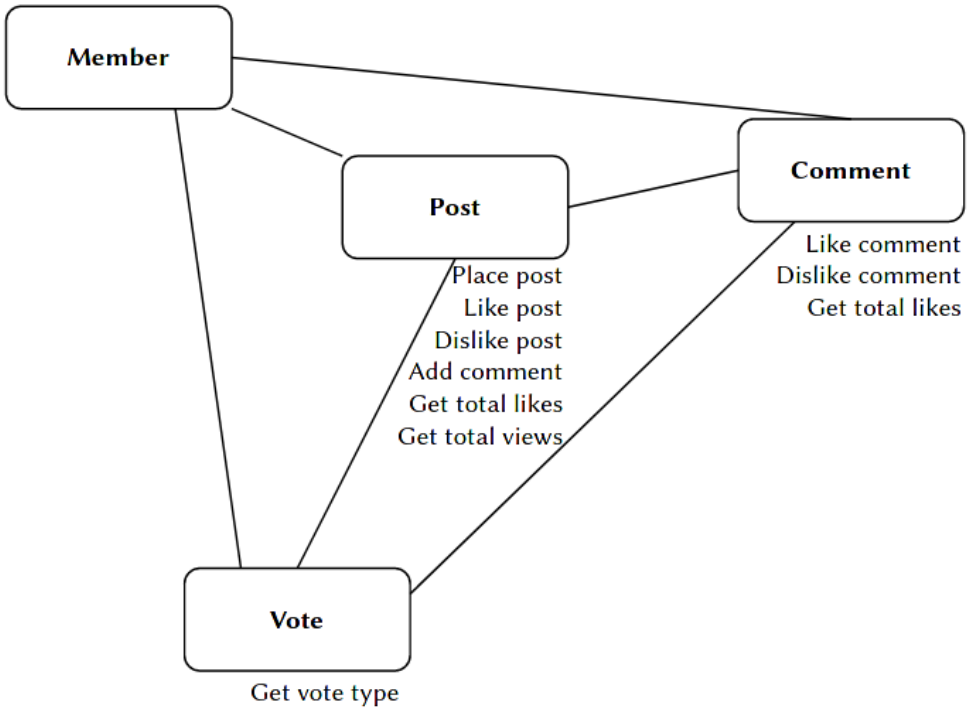

Рисунок 2.3 – Діаграма відповідальності об’єкта

Після створення діаграми взаємодії об’єктів та відповідальності, потрібно перейти до складання UML діаграми класів. UML діаграма класів зображена на рисунку 2.4.

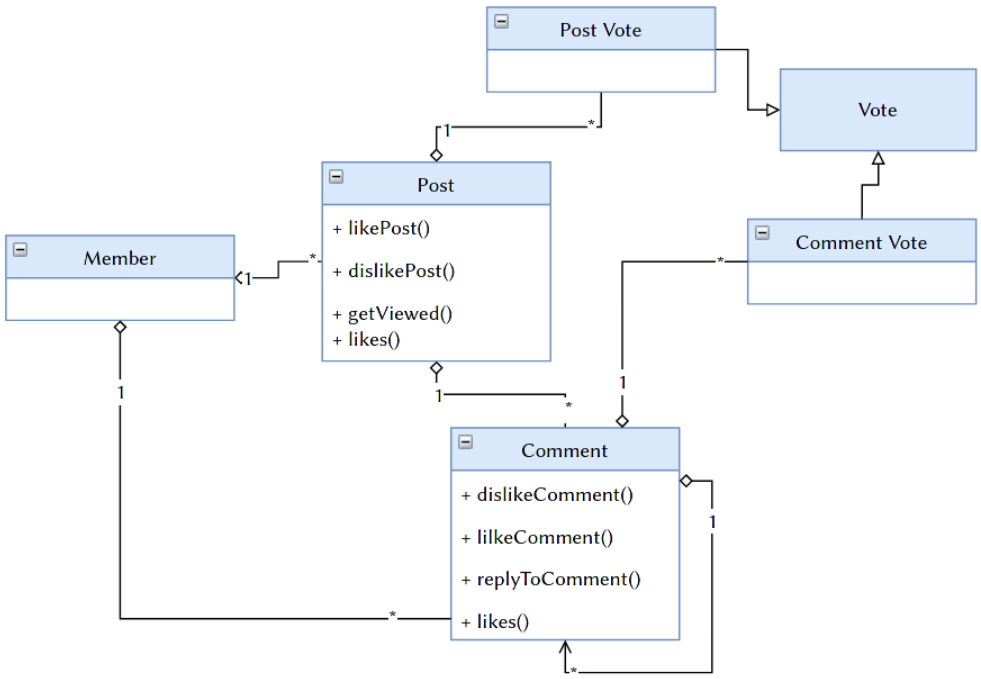

Рисунок 2.4 – UML діаграма класів бізнес-логіки

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 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.

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;

Реалізація базового класу доменної події зображено на додатку А.

ПЕРЕЛІК ПОСИЛАНЬ

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

2. Domain Driven Design - URL: https://martinfowler.com/bliki/DomainDrivenDesign.html (дата звернення 05.05.2021).

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. 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).

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

20. 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).

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

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

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

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

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

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

27. 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.

28. 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).

29. 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

ДОДАТОК А
БАЗОВІ МОДУЛІ СИСТЕМИ

Базовий клас DomainEvents

```
import EventEmitter from 'events';

import { DomainEvent } from './domain-event.interface';
import { UniqueID } from '../unique-id.base';
import { AggregateRoot } from '../aggregate-root.base';

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

```
import { v4 as uuidv4 } from 'uuid';
import { Identifier } from './id.base';

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

```
import { UniqueID } from './unique-id.base';
```

```
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

```
import { Entity } from './entity.base';
import { DomainEvent } from './event/domain-event.interface';
import { DomainEvents } from './event/domain-events.base';

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

```
import { NotSpecification } from './internal';
import { AndSpecification } from './internal';
import { OrSpecification } from './internal';

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 Б
МОДУЛІ БІЗНЕС РІВНЯ

Сутність Member

```
import { AggregateRoot } from '../../../shared/aggregate-root.base'
  ;
import { UserName } from './username.value';
import { Email } from './email.value';
import { FirstName } from './first-name.value';
import { LastName } from './last-name.value';
import { UniqueID } from '../../../shared/unique-id.base';
import { Result } from '../../../shared/core/result.base';
import { Guard } from '../../../shared/core/guard.base';
import { MemberCreated } from '../../events/member-created.event';

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

```
import { AggregateRoot } from '../../../shared/aggregate-root.base'
  ;
import { Guard } from '../../../shared/core/guard.base';
import { Result } from '../../../shared/core/result.base';
import { UniqueID } from '../../../shared/unique-id.base';
import { PostCreated } from '../../events/post-created.event';
import { Comments } from '../comment/comments.list';
import { Hashtags } from '../hashtag/hashtags.list';
import { PostVotes } from './post-votes.list';
import { PostText } from './post-text.value';
import { PostTitle } from './post-title.value';
import { Comment } from '../comment/comment.entity';
import { CommentPosted } from '../../events/comment-posted.event';
import { PostViewed } from '../../events/post-viewed.event';
import { PostEdited } from '../../events/post-edited.event';
import { PostVote } from './post-vote.entity';
```

```
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

```
import { Entity } from '../../../shared/entity.base';
```

```
import { Result } from '../../../shared/core/result.base';
import { Guard } from '../../../shared/core/guard.base';
import { UniqueID } from '../../../shared/unique-id.base';
import { CommentText } from './comment-text.value';
import { CommentVotes } from './comment-votes.list';
import { CommentVote } from './comment-vote.entity';

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

```
import { Result } from '../../shared/core/result.base';
import { CommentVote } from '../entities/comment/comment-vote.
   entity';
import { Comment } from '../entities/comment/comment.entity';
import { Member } from '../entities/member/member.entity';
import { PostVote } from '../entities/post/post-vote.entity';
import { Post } from '../entities/post/post.entity';

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

```
import { Validator } from '../../shared/validator.base';
import { PostTitle } from '../entities/post/post-title.value';
import { ShorterThanSpec } from './shorter-than.spec';
import { LongerThanSpec } from './longer-than.spec';
import { Specification } from '../../shared/specificatinon/
   specification.base';

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

```
import { DomainEvent } from '../../shared/event/domain-event.
  interface';
import { Post } from '../entities/post/post.entity';
import { UniqueID } from '../../shared/unique-id.base';

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

```
import { DomainEvent } from '../../shared/event/domain-event.
  interface';
import { UniqueID } from '../../shared/unique-id.base';
import { PostVote } from '../entities/post/post-vote.entity';

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

```
import { DomainEvents } from '../../shared/event/domain-events.base
  ';
import { Handler } from '../../shared/event/handler.interface';
import { PostCreated } from '../events/post-created.event';
import { Logger } from '../ports/logger.port';
```

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

  setupSubscriptions(): void {
    DomainEvents.register(this.handle.bind(this), 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

```
import { GetMemberUseCase } from '../../application/use-cases/
  member/get-member.use-case';
import { DomainEvents } from '../../shared/event/domain-events.base
  ';
import { Handler } from '../../shared/event/handler.interface';
import { Member } from '../entities/member/member.entity';
import { PostRated } from '../events/post-rated.event';
import { EmailServicePort } from '../ports/email-service.port';

export class PostRatedHandler implements Handler {
  constructor(
    private readonly _emailService: EmailServicePort,
    private readonly _getMemberService: GetMemberUseCase,
  ) {
```

```
    this.setupSubscriptions();
  }

  setupSubscriptions(): void {
    DomainEvents.register(this.handle.bind(this), PostRated.name);
  }

  async handle(event: PostRated): Promise<void> {
    const memberResult = await this._getMemberService.getMember({
      memberId: event.postVote.memberId.toString(),
    });

    const member = memberResult.value as Member;

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