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.

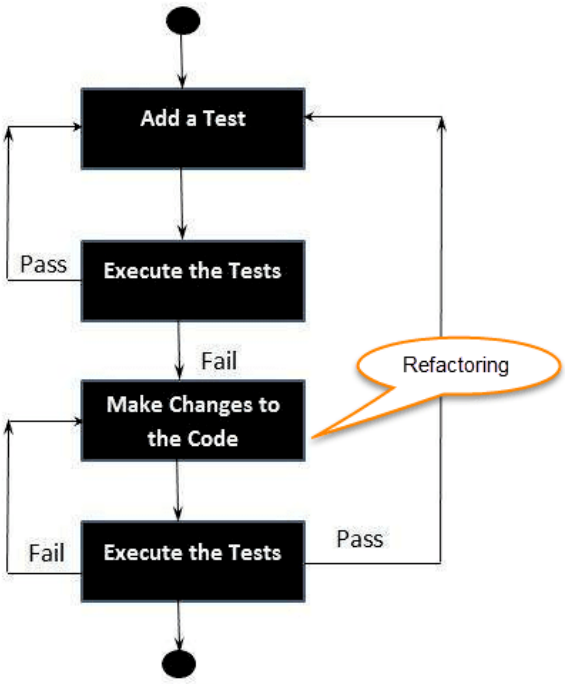

Рисунок 1.1 – Цикл керованою тестами розробки

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.

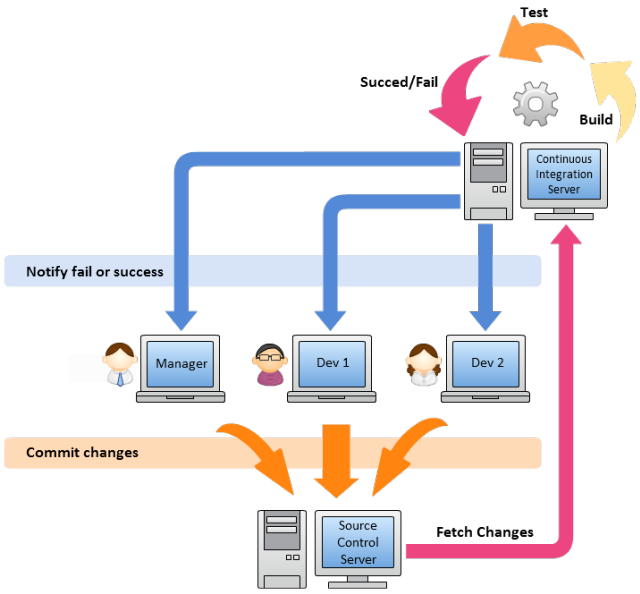

Рисунок 1.2 – Цикл роботи неперервної інтеграції

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

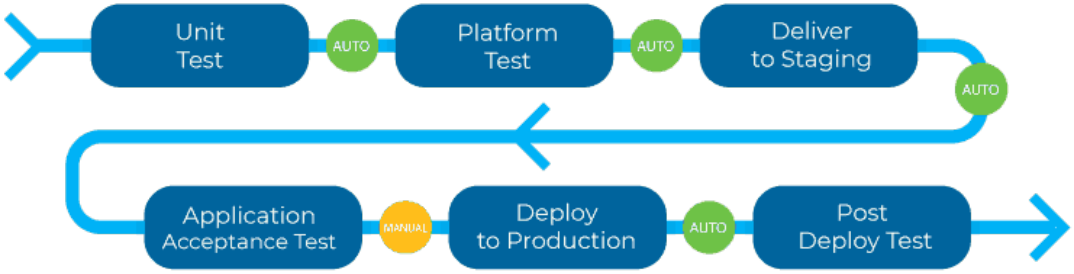

Рисунок 1.3 – Модель безперервної доставки

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 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.

1.4 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 [24].

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.

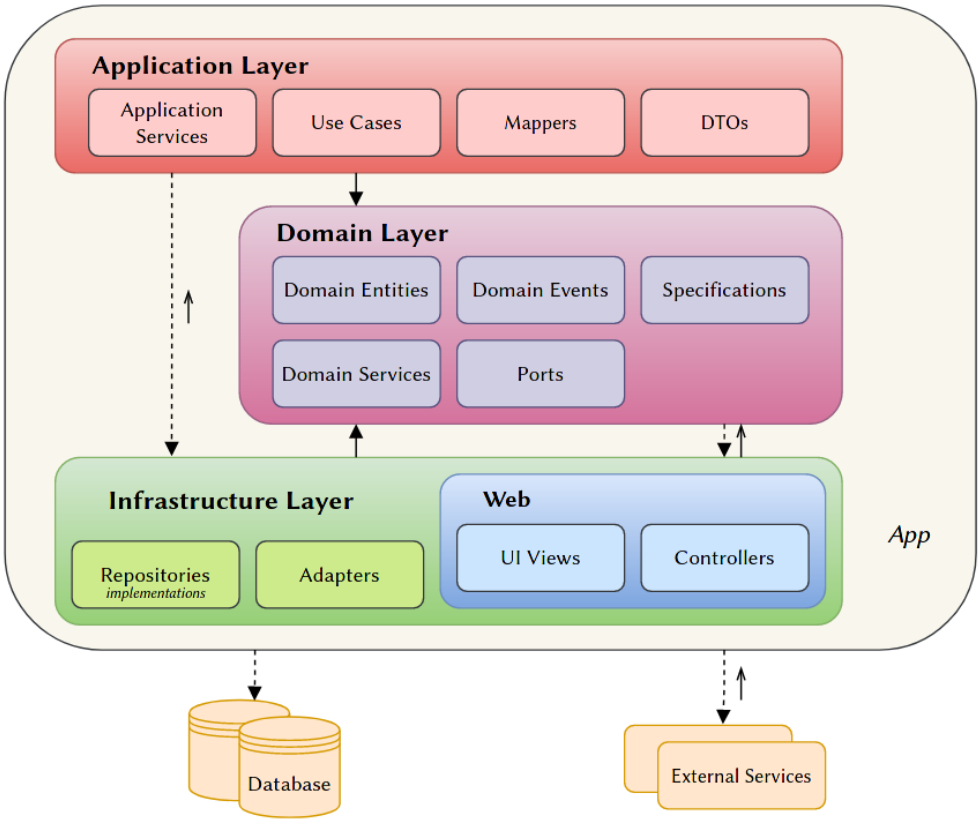

Рисунок 2.1 – Архітектура додатку

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 [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 2.2)

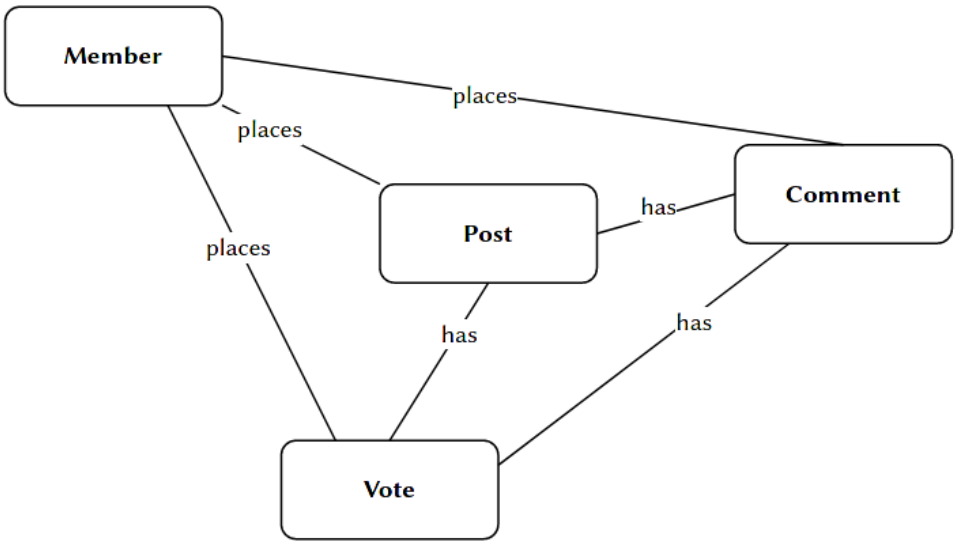

Рисунок 2.2 – Діаграма об’єктної взаємодії

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.

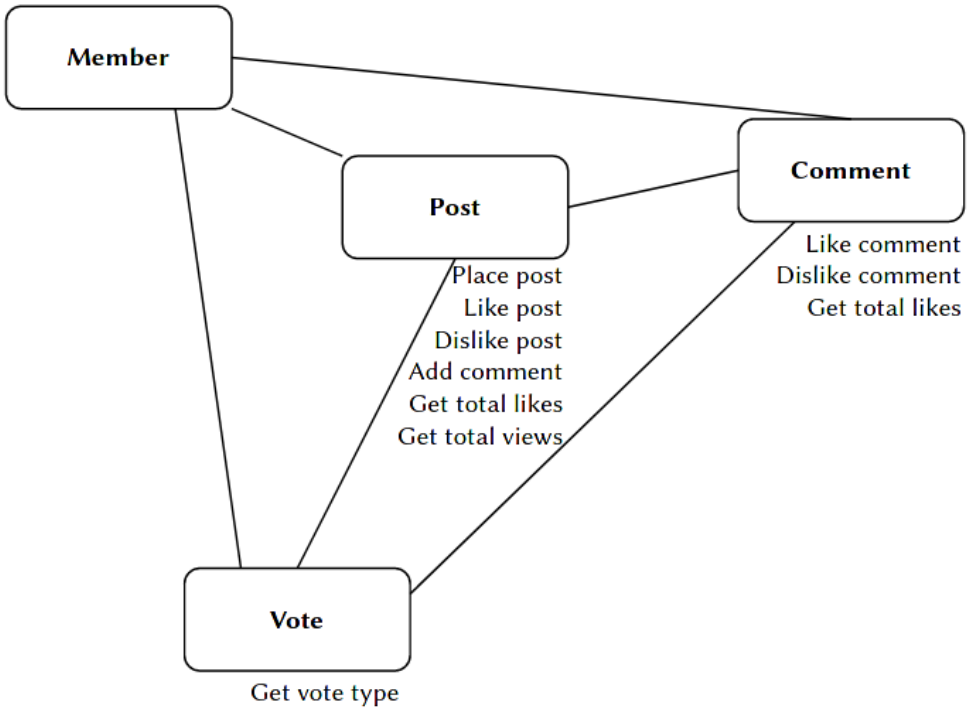

Рисунок 2.3 – Діаграма відповідальності об’єкта

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

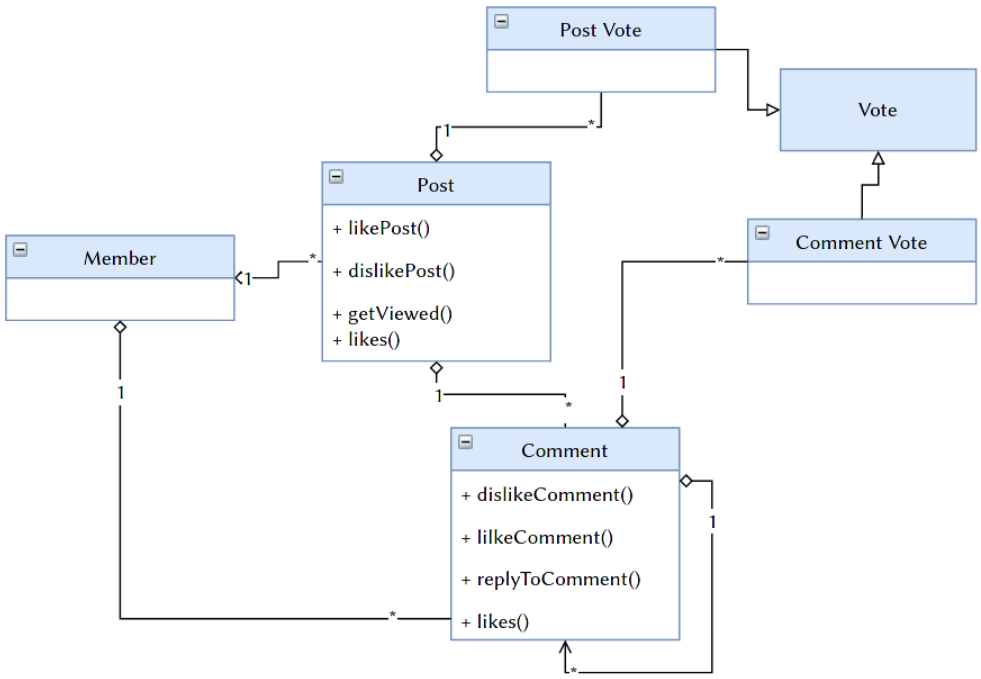

Рисунок 2.4 – UML діаграма класів бізнес-логіки

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 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 пакетиний 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
```

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;

Реалізація базового класу доменної події наведено у додатку А.

4 ТЕСТУВАННЯ ТА ФАКТОРИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Основні поняття тестування

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

Існує велика кількість підходів вирішення задач тестування.

Тестування може виконуватися на всіхстадіях 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.

Частіше всього на тестування недостатньо виділяється часу, щоб провести найбільш повне тестування. Тоді основна задача складається в тому, щоб вибрати відповідний набір тестів, для проведення в стислі строки. Для цього в тестуванні використовують нарощувальний підхід. Тестування поділяється на стадії, кількість яких пропорційна часу на тестування [29, 30].

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

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

– основна функціональність;

– надійність;

– ефективність;
– практичність;
– можливість супроводжувати;
– практичність;
– мобільність;
– функціональність.

Основний список критеріїв і атрибутів було складено у стандарті 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)

До стадій тестування відносять наступні [29, 31]:

Стадія 1 - Вивчення - Ознайомлення з програмою/системою.

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

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

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

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

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

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

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

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 коду програмного забезпечення:

– тестування чорного ящика (black box) - немає доступу до коду, доступ тільки через інтерфейси, до яких має доступ як замовник, так й користувач;

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

– тестування білого ящика (white box testing) - є доступ до коду та виконувач може писати код, використовуючи розроблені бібліотеки;

За *ступенем автоматизації*:

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

За *ступенем ізольованості компонентів*:

– компонентне (модульне) тестування (component/unit);
– інтеграційне тестування (integration);
– системне тестування (system/end-to-endtesting).

За *часом проведення*:

– альфа-тестування (alpha testing);
– регресійне тестування (regression);
– тестування при прийомці (smoke testing);
– тестування нової функціональності(new feature testing);
– тестування при здачі (acceptance);
– бета тестування (beta testing).

За *ознакою позитивності сценаріїв*:

– позитивне тестування (positive testing);
– негативне тестування (negative testing).

За *ступенем підготовленості до тестування*:

– тестування по документації (formal testing);
– тестування ad hoc або інтуїтивне тестування (ad hoc testing).

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

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

При необхідності виконання тестування не функціональних параметрів програми -створюються/описуються тести, необхідні для визначення характе-

ристик ПЗ, що можуть бути виміряні різними додатковими елементами/величинами. До таких тестів відносять:

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

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

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

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

Альфа-тестування – імітація реальної роботи програми/продукту Найчастіше альфа-тестування проводиться на ранніх стадіях розробки продуктів, але іноді може застосовуватися як внутрішнє приймальне тестування. Виявлені помилки можуть бути передані для додаткового дослідження у середовищі, подібному тому, в якому буде 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 зворотній зв’язок про продукт від його майбутніх користувачів.

Часто стадія альфа-тестування характеризує функціональне наповнення коду, а бета-тестування - стадію виправлення помилок. При цьому, як правило, на кожному етапі розробки проміжні результати роботи доступні кінцевим користувачам [29, 31].

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?

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).