Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

МОДЕЛЮВАННЯ ЗА ДОПОМОГОЮ DDD ТА АРХІТЕКТУРНІ ПАТЕРНИ

Текстова частина до магістерської роботи
за спеціальністю „Інженерія програмного забезпечення”

Виконав: студент 2-го року навчання
Гавришко Ярослав Олегович
Керівник Франчук О.В.,
Доцент, кандидат технічних наук
Магістерська робота захищена
з оцінкою
«_____»
Секретар ДЕК _____
«____» _____2021 р

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
доц., канд. фіз.-мат.
наук _____
С. С. Гороховський
(підпис)
8 листопада 2020 р

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на дипломну роботу

студенту Гавришку Я.О. факультету інформатики 1 курсу МП ТЕМА Моделювання за допомогою DDD та архітектурні патерни

Зміст ТЧ до дипломної роботи:
Індивідуальне завдання
Вступ
1. Огляд принципів Domain-driven design
2. Огляд основних архітектур
3. Проектування системи за допомогою DDD
Список використаних джерел

Дата видачі 8 листопада 2020 р. Керівник _____
(підпис)

ЗМІСТ

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	листопад 2020 р.	
2.	Оволодіння принципами DDD.	15.01.2021	
3.	Огляд архітектурних патернів	10.02.2021	
4.	Дизайн системи за допомогою DDD	15.03.2021	
5.	Створення слайдів для доповіді та написання доповіді	01.04.2021	
6.	Надання роботи керівнику для перевірки	17.05.2021	
7.	Корегування роботи за результатами перевірки керівником	20.05.2021	
8.	Остаточне оформлення пояснювальної роботи та слайдів	24.05.2021	
9.	Захист дипломної роботи	25.05.2021	

Студент _____

Керівник _____

“_____”_____

Анотація

Дана дипломна робота зосереджена на Domain-driven design, підходу до розробки програмного забезпечення, архітектурних моделей і стилів. При аналізі патернів в контексті Domain-driven design, досліджується сумісність та можливість поєднання даного патерну та Domain-driven design. Зразкову систему представлено як побудовану на принципах Domain-driven design і використанні описаних архітектурних патернів та стилів.

Вступ

Так як інформаційні технології набрали популярність у другій половині XX століття, вимоги до інформаційних систем зросли досить швидко. Адаптуючись до зростання кількості запитів клієнтів, невеликі програми еволюціонували в складні системи з багатою функціональністю. З ростом системи зростає і її складність, а це, у свою чергу, робить її важчим для розуміння, підтримання та розвитку. Великі проекти були особливим викликом для архітекторів, яким потрібно було спроектувати системи, які б відповідали вимогам клієнтів, були б корисними на практиці та вирішували поставлені проблеми. Часто, помилки у розумінні вимог замовника ставали причиною того, що такі системи не були корисні та не забезпечували належних вирішень проблем.

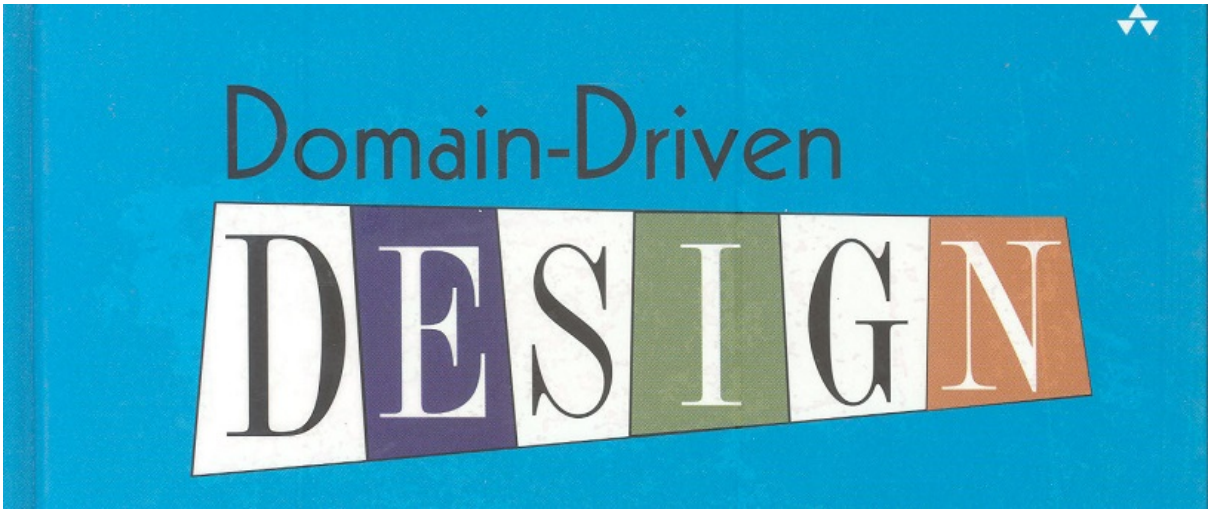
У 2003 році Ерік Еванс опублікував свою книгу “Domain-Driven Design: Tackling Complexity in the Heart of Software”, яка намагається вирішити проблеми проектування великих систем. У цій книзі автор представив підхід до проектування програмного забезпечення Domain-driven design (DDD), головний фокус якого спрямований на моделюванні домену та співпраці між розробниками та доменними експертами протягом усієї фази розробки програмного забезпечення. Головним внеском Еріка Еванса є якраз таки принципи, патерни та практики, які допомагають на всіх етапах проектування.

З моменту публікації даної книги, DDD було розглянуто та описано в багатьох подальших книгах та статтях. Особливої популярності набули дизайн патерни, описані Евансом. У цій дипломній роботі розглянуто різні моделі архітектури та їх сумісність з використанням DDD.

Дипломна робота має наступну структуру. Перша частина знайомить з DDD, його принципами та патернами. Наступні розділи зосереджені на архітектурних моделях та стилях. Основна мета даної роботи - аналіз

використання згаданих у цій роботі патернів з DDD, їх сумісність та можливість використання у системах з DDD. Найбільш важливі патерни демонструються на практиці.

РОЗДІЛ 1: Принципи Domain-driven design



Загальні характеристики

Domain-driven design - це підхід до розробки програмного забезпечення, який зосереджує розробку на програмуванні доменної моделі, яка відображає глибоке розуміння процесів та правил домену. Назва походить від книги Еріка Еванса, написаної у 2003 році, яка описує даний підхід через каталог патернів.

Домени

Домен - це зазвичай те, чим займається компанія, область в якій працює компанія, а програмне забезпечення призначене для вирішення проблем у цій галузі. Домен можна поділити на менші частини (субдомени). Domain-driven design має провідні три типи субдоменів:

- основний домен

- допоміжний домен
- загальний домен

Основний домен

Основний домен - це найбільш важлива частина бізнес логіки. Це те, що приносить дохід компанії, те на чому спеціалізується компанія, а також те, що робить її успішною. Для прикладу, візьмемо онлайн магазини, основним доменом для якого є продаж товарів. Дуже важливо при моделюванні та проектуванні найбільшу увагу приділити саме цьому субдомену. Це можна досягти за допомогою збільшення інтенсивності комунікації з доменними експертами.

Допоміжний домен

Допоміжні домени не описують головний фокус компанії, але допомагають підтримувати основні домени. Наприклад, для онлайн магазинів управління складом не є основною сферою, це не те, що приносить дохід, це лиш домен, який допомагає підтримувати основну сферу компанії - продаж товарів.

Загальний домен

Даний субдомен є найменш важливим для компанії. Такі субдомени існують у багатьох системах, наприклад, розсилка емейлів тощо. Вони не вимагають особливої уваги. Іншими словами, загальні домени це бізнес активності, які кожна компанія виконує однаково або ж дуже схоже.

Розподіл знань

У Domain-driven design центром кожної стадії розробки програмного забезпечення є домен. Щоб збудувати дійсно корисну та хорошу систему, програмісти повинні розуміти домен, процеси, які відбуваються у різних ситуаціях, що є дуже важливим, а що ні, та інше. Люди які надають знання про домен називаються доменими експертами.

Доменні експерти та розробники повинні часто взаємодіяти протягом усього процесу розробки ПЗ. Навіть якщо така взаємодія потребує багато часу, це все одно слід робити, оскільки підхід орієнтований на довготривалу перспективу. Ідеальна ситуація коли доменний експерт є одним із команди розробників. Важливою частиною є постійна взаємодія між доменними експертами та розробниками. Згідно цього, традиційна каскадна модель розробки програмного забезпечення не буде релевантною для DDD, оскільки передбачає лише одну фазу комунікації з доменими експертами.

Загальновживана мова (Ubiquitous language)

Для того, щоб зробити комунікацію легшою розробники та домені експерти повинні використовувати мову, яку Ерік Еванс визначив як загальновживана мова (ubiquitous language). Це результат розподілу знань та спільного розуміння речей над якими ведеться розробка. Домені експерти повинні надавати правильні терміни для різних ситуацій. Усі терміни повинні мати точне визначення, більше того, один термін може мати лише одне значення, щоб уникнути проблем двозначності в

майбутньому. Це одна з причин, чому такі часто вживані слова, як: менеджер, контроллер або сервіс, не є добрими ідеями для найменування. Загальновживана мова повинна використовуватись у всіх аспектах розробки ПЗ. Її слід використовувати в коді з тими самими термінами та поняттями, що використовуються як імена класів, методів, властивостей тощо.

Доменна модель

Знання, які ми отримали про домен, зображені в доменній моделі. Вона репрезентує вигляд домену, побудованим відповідати всім бізнес задумкам. Доменна модель описана загальновживаною мовою і працює як зв’язок між доменними експертами та розробниками. Не обов’язково щоб вона була досконалою, повністю відображала реальність тощо. Її мета - бути наближеною до реальності, але лише з точки зору бізнесу, і відображати лише те, що має значення для бізнесу.

Для того щоб зробити доменну модель більш корисною, необхідно постійно проводити процес синхронізації з актуальною кодовою базою. Доменна модель, яка не відображена в коді може швидко стати не актуальною, ба більше вводити в оману людей, які працюють над проектом. Тому був створений так званий model-driven design підхід, який налагоджує та підтримує тісні взаємозв’язки доменної моделі та кодової бази.

Обмежений контекст

Обмежений контекст (bounded context, BC) - це свого роду лінгвістичні рамки доменної моделі. В середині обмеженого контексту поняття моделі, як властивостей чи операцій, має особливе значення. Кожен термін в одному обмеженому контексті повинен мати лише одне значення. Але той самий термін може використовуватись в іншому обмеженому контексті, для того, щоб описати щось зовсім інше.

Обмежений контекст може бути дуже корисним у великих доменах з багатим словником термінів. У таких доменах може бути доволі складно зрозуміти, що всі терміни мають точне, єдине і чітке значення. Наприклад, слово фунт може означати або одиницю ваги, або грошову одиницю Великобританії. Якщо ми розділимо домен покупок товарів на кілька обмежених контекстів, то, зрозуміло, що в обмеженому контексті торгового складу, слово фунт буде використано для опису ваги товарів, тоді як у обмеженому контексті оплат, слово фунт буде використовуватись як валюта.

Контекстні мапи

Контекстна мапа описує всі обмежені контексти та їхні взаємозв’язки. Це діаграма, яка допомагає візуалізувати межі обмежених контекстів. Вона використовується розробниками та доменними експертами для того, щоб мати краще уявлення всієї системи і для захисту цілісності кожного обмеженого контексту.

Існує кілька паттернів, які описують взаємозв’язки між обмеженими контекстами: shared kernel, customer-supplier development, conformist, anti-corruption layer, separate ways, open host service, published language.

РОЗДІЛ 2: Тактичні патерни

Загальні характеристики

Впродовж стратегічної фази DDD відбувається маппінг бізнес домену і визначених обмежених контекстів для доменної моделі. Іншими словами, доменні моделі стають більш точними у своїх визначеннях. Тактичні патерни застосовуються в межах одного обмеженого контексту. У мікросервісній архітектурі найбільше особливо цікавлять entity і aggregate патерни, адже саме вони допомагають визначити природні зв’язки для сервісів у системі. Існує таке загальне правило - сервіс повинен бути не меншим за aggregate і не більшим за обмежений контекст.

Entities

Entities - це об’єкти з своїми атрибутами та функціями, у яких, насамперед, унікальність є важливою. Це означає, що при зміні якогось з атрибутів об’єкту, сам об’єкт не втрачає свою ідентичність. Як хороший приклад entity можна привести людину. Якщо людина змінить зачіску, то вона всеодно залишиться тією ж людиною.

Ці об’єкти зазвичай моделюються як змінні класи з унікальним ідентифікатором, якого неможливо змінити після створення та який використовується при порівнянні entities.

Value objects

Value objects - це об’єкти, які вміщують в собі якесь певне значення. Вони не мають ідентифікаторів та після кожної зміни набувають нового значення. Базовим прикладом value objects є дата. Якщо ми змінимо будь-який елемент дати (час, день, місяць), це вже буде зовсім інша дата. Особливістю value objects є те, що їх можна легко замінити, адже в програмі неважливо який екземпляр класу використовувати, якщо всі їхні атрибути однакові. Тому такі об’єкти прийнято моделювати як незмінні, тобто ми не можемо поміняти йому атрибут, адже для того, щоб таке зробити, нам потрібно буде створити новий об’єкт з новим атрибутом. Два value objects вважаються однаковими, якщо усі їх атрибути однакові. Value objects є достатньо легкими у користуванні, якраз таки через те, що вони незмінні та не потрібно дбати про їх ідентичність. Через те і рекомендується, де можливо, використовувати value objects замість entities.

Domain services

Domain services - це об’єкти, які не зберігають стан, а лише забезпечують доменну логіку. Їх використовують, коли присутня більш складна бізнес логіка і важко визначити до якого entity чи value object належить якась певна частина функціональності. Domain services варто використовувати з обережністю, тільки тоді, коли це є дійсно необхідним, адже це може призвести до того, що вся доменна логіка буде зосереджена у domain services, а не навпаки у entities і value objects.

Aggregates

Aggregates - це найскладніший тактичний інструмент у DDD. Aggregate називають кластер з entities і value objects. Тобто ці об'єкти розглядають як єдине ціле з точки зору зміни даних. У кожного aggregate є корінь і границя в якій завжди повинні бути дотримані інваріанти.

Всі запити до агрегату здійснюються через його корінь, який є entity з своїм унікальним ідентифікатором. Потрібно дуже уважно дизайнити aggregates, тому що при неправильному їх створенні може виникнути багато проблем. Для aggregates, які надто великі зазвичай складно забезпечувати консистентність, адже коли змінюєш якийсь з об’єктів aggregate, інші aggregates повинні бути заблокованими. При моделюванні aggregates потрібно знати інваріантність моделі і зв’язки між ними.

Domain events

Domain event - це подія, яка відбувається внаслідок якоїсь іншої окремої дії. При моделюванні важливо враховувати, що domain event - це те, що сталось в минулому часі, тому ім’я повинне теж відображати минулий час. Наприклад, якщо в системі створено нове замовлення, то подія, яка може виникнути буде називатись OrderCreated.

Зазвичай, подія проектується як незмінний об’єкт (як value objects), інтерфейс якого виражає його призначення, а атрибути відображають його причину.

Modules

Modules - це іменовані в середині моделі контейнери для деякої групи тісно зв’язаних між собою об’єктів доменної логіки. Їх ціль - послабити зв’язки між класами, які знаходяться у різних модулях.

Необхідно проектувати модулі зі слабким зв’язком. Це полегшує підтримку та рефакторинг концепцій моделювання. Якщо ж зв’язність все ж таки потрібна, то повинні організовувати якомога більше ациклічних залежностей між модулями одного рівня, тобто таких які мають однакову вагу в проекті. Модулі краще не робити як статичні концепції моделі, через те, що вони повинні змінюватись в залежності від тих об’єктів, які вони організовують.

Factories

Деякі aggregates і entities можуть бути досить складними. Складний об’єкт не може створювати самого себе через конструктор (у книжці Еріка Еванса є приклад двигуна автомобіля, який збирається або механіком, або роботом, але аж ніяк не сам). Краще виконувати створення складних агрегатів чи інших об'єктів окремо. Для того використовують фабрики, які якраз призначені для створення інших об’єктів. Частіше всього фабрики проектуються як фабричний метод у корені агрегату.

При створені фабричного метода обов’язково потрібно дотримуватись усіх інваріантностей агрегата, створюючи його як єдине ціле. Цей метод повинен бути єдиним і неподільним. Всі дані для створення повинні бути передані за одну комунікаційну операцію. Деталі конструкції повинні скриватись.

Value objects і entities створюються по різному. Так як value objects незмінні, то всі атрибути повинні передаватись одразу. Для entities можна добавляти лише ті, які важливі для створення конкретного агрегата і його інваріантів.

Repositories

Репозиторії використовуються для зберігання агрегатів. Вони інкапсулюють в собі логіку збереження, отримання, оновлення та видалення агрегатів з вказаного сховища.

З точки зору використання є два види репозиторіїв: collection-oriented та persistence-oriented. Collection-oriented репозиторії поводять себе як in-memory колекції. Це означає, що вони не мають методів збереження та оновлення. А це вже приводить до більш кращого коду, адже для того, щоб зробити якусь модифікацію, достатньо буде завантажити агрегат, а потім змінити його, без виклику інших методів репозиторію. Недоліком даного підходу є складніша імплементація, оскільки основний механізм збереження повинен мати можливість відстежувати зміни об’єктів і в кінці транзакції відображати ці зміни у хранилищі.

Тимчасом, persistence-oriented репозиторії навпаки виступають як фізичні сховища, які мають методи збереження та оновлення, через те їх значно легше імплементувати.

РОЗДІЛ 3: Архітектура

Загальні характеристики

Однією з переваг DDD є те, що відсутня необхідність використистовувати конкретні архітектури. Як було розглянуто у минулих розділах, моделі (включаючи модель основного домену) перебувають у обмежених контекстах, де існують різні види взаємозв'язків між обмеженим контекстом. Завдяки цьому уможливлюється застосування різних архітектурних стилів або патернів в межах одного обмеженого контексту, або системи в цілому.

Провідною метою даної дипломної роботи є розгляд та аналіз найбільш поширених архітектурних стилів та патернів, їх переваг та недоліків, а також їхній спосіб використання разом з Domain-driven design.

У наступних розділах описуються різні архітектурні патерни та їх застосування на рівні логіки домену та обмеженого контексту, а також у системі в цілому. Варто зазначити, що фокус зосереджений на патерни, що безпосередньо впливають на DDD концепти, як модель домену чи обмежений контекст. Це означає, що ряд відомих архітектурних зразків опущено.

Патерни для доменної логіки

Домен або ж бізнес логіка є найважливішою частиною системи і лежить в основі обмеженого контексту. Незалежно від того, яка архітектура була обрана для обмеженого контексту, логіка домену повинна бути інкапсульована у власному юніті. У цьому розділі представлено, які патерни можна використовувати для дизайну та декомпозиції цього

юніта[1]. Попри те, що доменна модель є домінуючим патерном в DDD, у цьому розділі розглядаються альтернативні патерни для доменної логіки.

Транзакційний скрипт

Транзакційний скрипт - це процедурний стиль організації доменної логіки. Створений транзакційний скрипт виконує запит напряму до бази даних або через тонку обгортку бази даних і виконує крок за кроком певні дії: спочатку валідацію, потім завантажує дані з бази даних, виконує якісь калькуляції і вкінці зберігає результат в базу даних. Всі ці кроки - є частиною одного транзакційного скрипта.

Найбільш вагомою перевагою даного підходу є його простота. Це тривіальна процедурна модель, яка є добре відомою для кожного розробника. Її можна скоро та легко імплементувати, також легко визначити транзакційні межі. Проте, у великих системах доволі складно тримати це у правильному та добре задизайненому стані. Для складного домену система може стати заплутаним набором процедур без структури.

Транзакційний скрипт у DDD

Основною проблемою транзакційного скрипта є те, що він не використовує об’єктну модель для репрезентації домену, тому всі взаємозв’язки, обмеження і правила представляються лише як певні кроки у методі, вони не зберігаються в моделях через тактичні патерни. Ось чому досить складно досягти того, що пропонує DDD за допомогою транзакційних скриптів.

Транзакційний скрипт все ж таки можна використовувати у DDD, але лише для дуже малих потреб домену, які природно підходять під процедурний стиль скрипту. Це зазвичай ситуації коли поведінка більш важлива ніж самі дані. Для прикладу, система, яка займається аналізом великих об’ємів даних може використовувати транзакційний скрипт, замість доменної моделі. У такій системі велика кількість неоднорідних даних унеможливлює використання об’єктної моделі.

Табличний модуль

Визначення табличного модулю знаходиться десь між доменною моделлю і транзакційним скриптом. Головна відмінність від доменної моделі - сенс екземплярів. У доменній моделі один екземпляр класу зазвичай відповідає одному екземпляру сутності у реальному житті. Для прикладу, екземпляр класу Order визначає одне замовлення у реальному житті. З іншої сторони, у табличному модулі існує лише один екземпляр класу, який використовується для управління всіх екземплярів у реальному житті. Клієнти такого екземпляру повинні надати дані з яким саме екземпляром вони будуть працювати.

Даний підхід надає більше структурованості, але все одно не є таким чітким і виразним як доменна модель. Найбільш вагомою перевагою такого підходу є те, як він підходить для різних утиліт забезпечення доступу до бази даних, таких як: ADO.NET або JDBC, тому він не потребує ніяких ORM та складних мапінгів.

Анемічна доменна модель

Анемічна доменна модель - це ситуація коли об’єкти моделі (сутності) не мають в собі жодної логіки поведінки, лише зберігають в собі дані. Це прийнято вважати анти-патерном, оскільки такий підхід перечить ідеям об’єктно орієтованого програмування. Інша проблема полягає в тому, що створити таку модель дорого, адже для неї ще потрібно створювати об’єкти та зв’язки між ними, також виникає необхідність використовувати ORM утиліти.

Архітектура обмежених контекстів

У системі, яка поділена на декілька обмежених контекстів, можна підібрати певну архітектуру для кожного окремого обмеженого контексту, незалежно від інших обмежених контекстів. Для того, щоб мати незалежні архітектури у всіх обмежених контекстах, слід вибрати правильну архітектуру на верхньому рівні. У цій главі описуються можливі архітектурні патерни для декомпозиції одного обмеженого контексту.

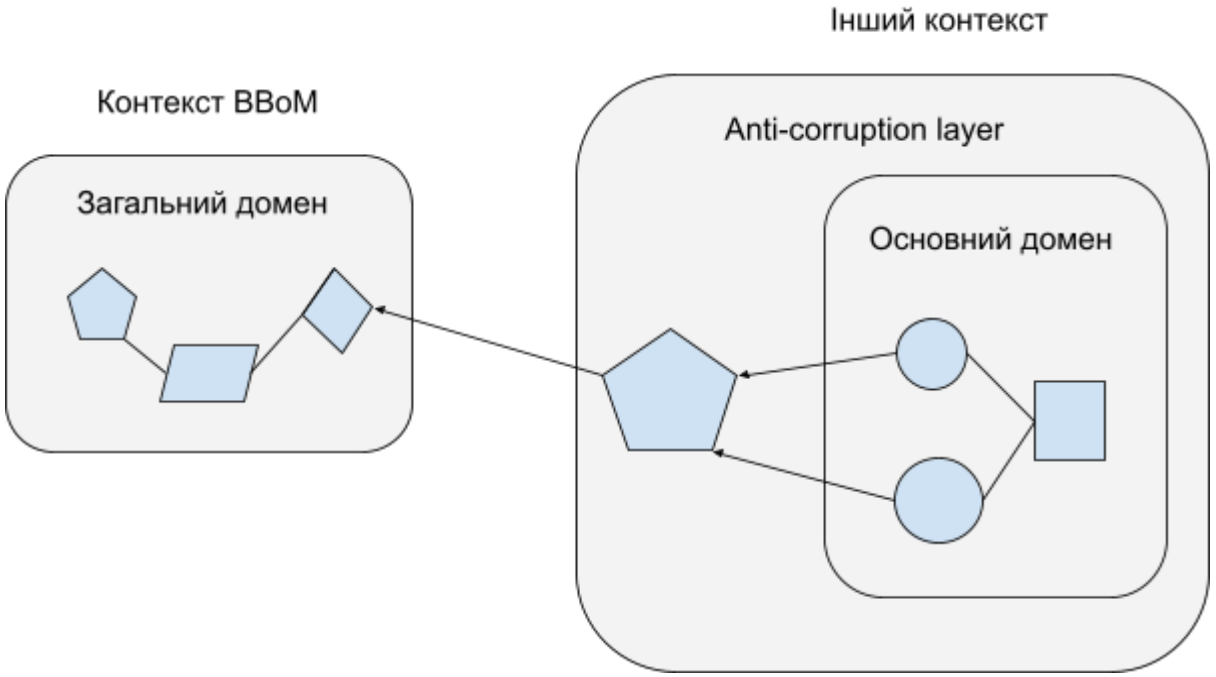
Big ball of mud

ВВоМ - це антипатерн, в основі якого лежить неструктурований код. Він зазвичай є результатом відсутності архітектури, швидкої та бездумної розробки, або ж браку знань правильної побудови ПЗ. Через те, що код

взагалі немає жодної структури, його дуже складно підтримувати, розуміти та розширювати.

Такий підхід прийнято не використовувати для домену, але все ж таки є моменти, де його можна застосувати. Він може використовуватись у нескладному та не зовсім важливому коді, особливо у загальному домені. Це зекономить час та ресурси, які можуть бути використані для покращення та рефакторингу основного домену.

ВВоМ повинен перебувати у власному обмеженому контексті. Інші обмежені контексти, які комунікують з ним, мають використовувати anti-corruption layer, який буде слугувати як адаптер між поганим та хорошим кодом.



Малюнок 1: Big ball of mug разом із DDD

Багаторівнева архітектура

Багаторівнева архітектура - це один із найбільш вживаних архітектурних патернів[2], у якому програмне забезпечення логічно або фізично поділено на рівні, де кожний рівень - це окремий рівень абстракції[3]. Кожний рівень інкапсулює в собі логіку і надає свій публічний інтерфейс рівню, який знаходиться вище. Кожен рівень може використовувати лише один рівень який знаходиться нижче. Не дозволяється залежність на рівень вище. Ось перелік типових рівнів:

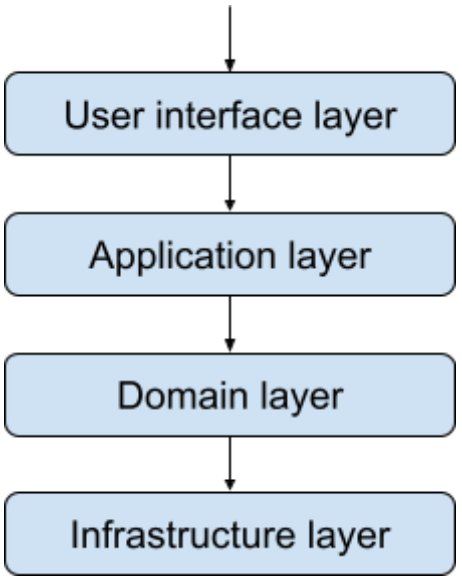
- presentation layer - використовується для виведення контенту для користувача
- domain layer - містить доменну логіку
- data source layer - використовується для роботи з базою даних

Рівнева структура ПЗ робить її легшою для розуміння. Адже для того, щоб зрозуміти як виглядає та чи інша бізнес операція, потрібно знати лише domain layer, без додаткових знань як це все відображається користувачу, або ж зберігається в базу даних. Також, можливо замінити цілий рівень на зовсім інше рішення, якщо збережена імплементація інтерфейсів. Для прикладу, ми можемо змінити data source layer з реляційною базою даних на нереляційну.

Проблемою багаторівневої архітектури є недотримання dependency inversion принципів. Вищі рівні зазвичай мають вищий рівень абстракції, але вони залежать від рівнів нижче. Особливо це стосується domain layer, який залежить від data source layer, що є інфраструктурним кодом низького рівня.

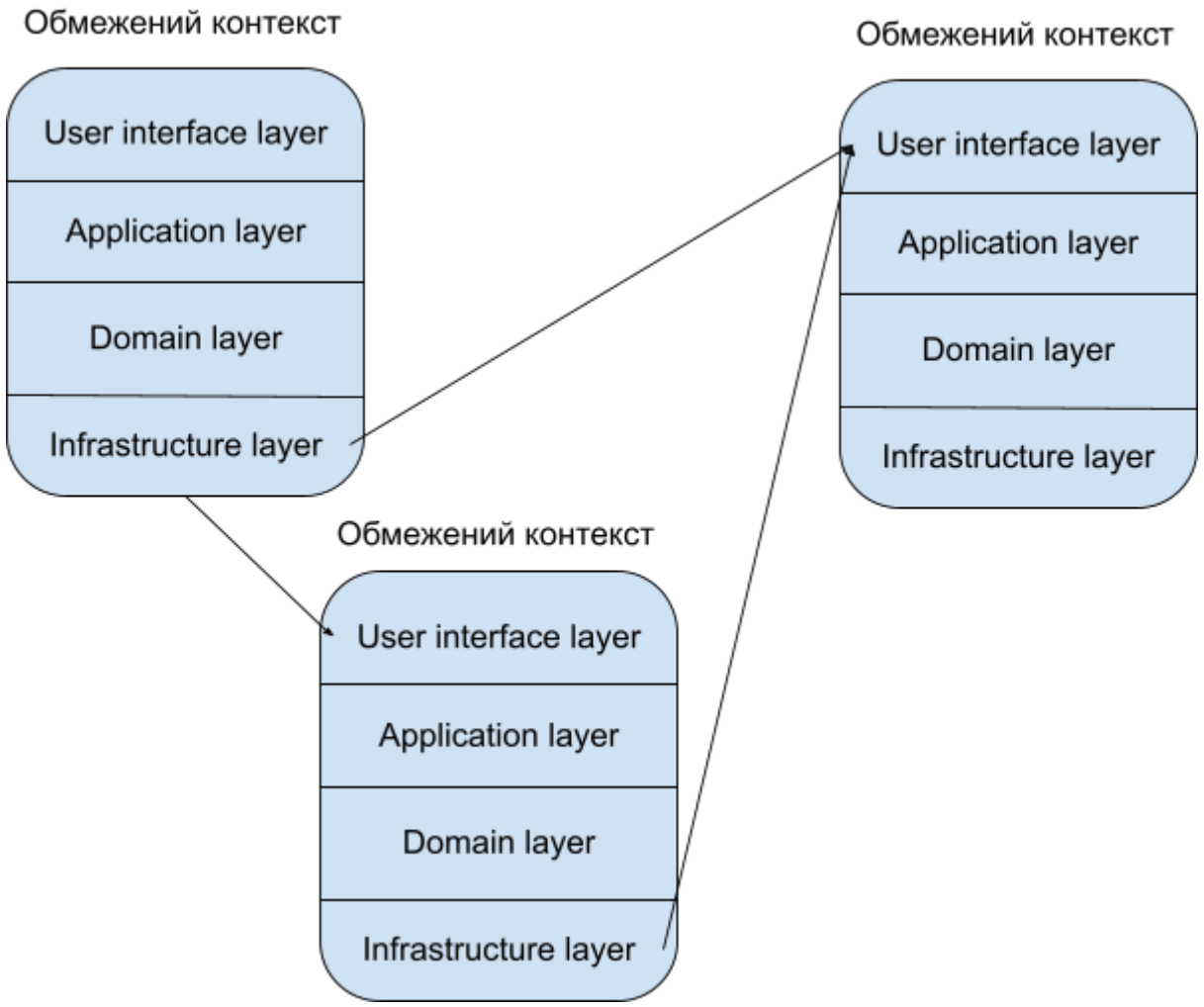
Ерік Еванс у своїй книжці найбільший фокус робив якраз таки на багаторівневу архітектуру. Він запропонував чотири рівні для DDD:

- user interface layer - відповідає за виведення даних користувача, користувач не обов’язково повинен бути людиною, це може бути інша інформаційна система.
- application layer - це тонка прослойка яка лише направляє, координує команди до domain layer. Він не містить жодної бізнес логіки та виступає лише як медіатор між user interface і domain layers. Також він відповідальний за менеджмент транзакцій та безпеки.
- domain layer - це основна частина системи, місце, де знаходиться уся бізнес логіка.
- infrastructure layer - містить в собі технічні імплементації. Це може бути доступ до бази даних через репозиторії, або ж комунікація з іншими системами.



Малюнок 2: Рівнева архітектура

Рівні можна також створювати і в рамках обмежених контекстів. Для того потрібно, щоб кожен обмежений контекст був окремим юнітом та не ділив жоден рівень з іншими обмеженими контекстами.



Малюнок 3: Архітектура на рівні обмженихконекстів

Шестикутна архітектура

Дана архітектура за принципом схожа до багаторівневої, адже також ділить систему на рівні. Але її головною відмінністю є те, що domain layer розташовується в центрі[4]. Це дозволяє архітектурі дотримуватись dependency inversion принципів, тому що тепер domain layer - найвищий рівень в коді та не залежить від жодних інших нижчих рівнів імплементацій. Шестикутна архітектура пропонує рівень адаптерів, котрі служать як переклад між моделлю даної шестикутної архітектури та зовнішнього світу. Зовнішні комунікують через порти (наприклад HTTPS) з адаптерами, які перетворюють запит та направляють його до моделі.

Цибулева архітектура дуже схожа на шестикутну архітектуру. Вона також поміщає рівень домену в центр і всі інші рівні можуть залежати від рівнів які ближче до центру, але не навпаки. Для простоти, через їхню велику схожість, у даній дипломній роботі, вони вважатимуться однаковими.

IDesign

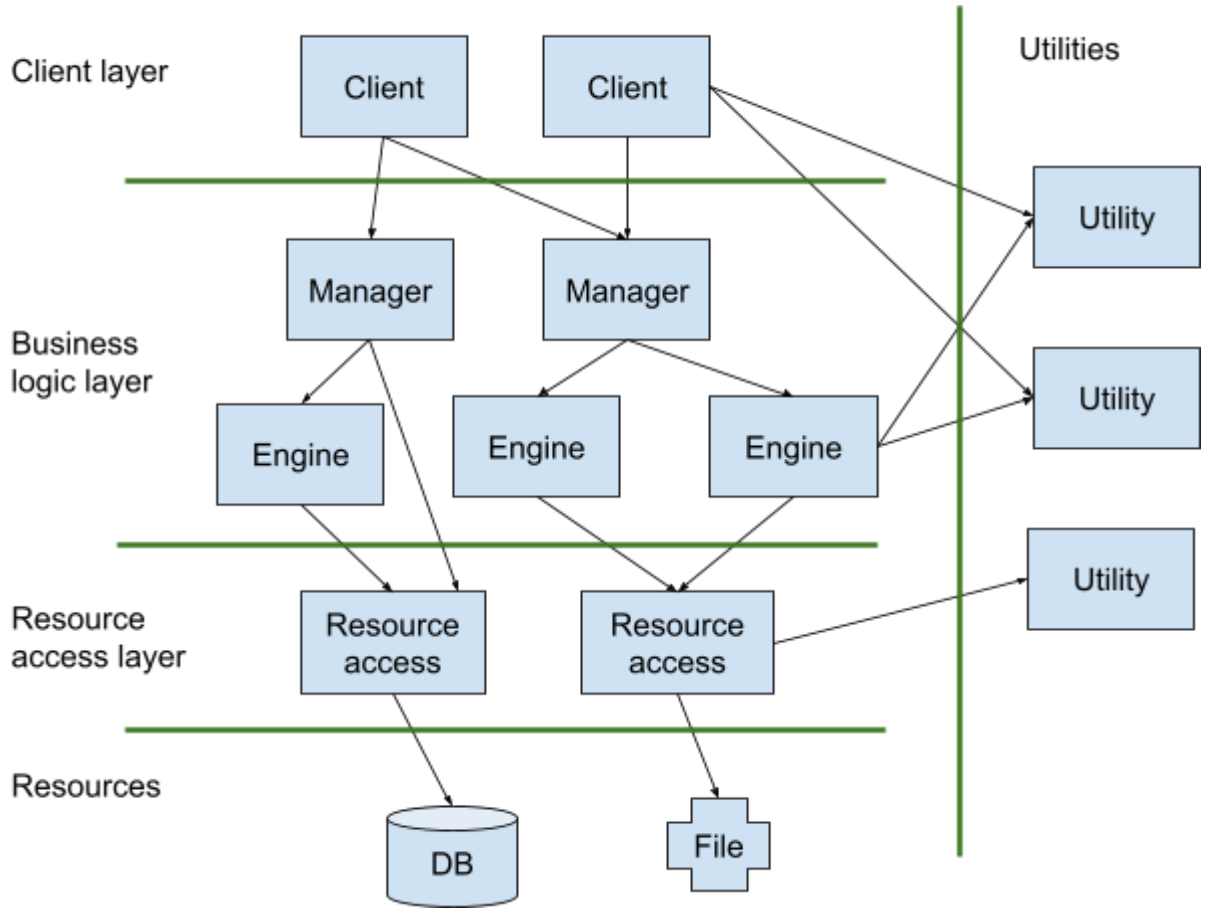
IDesign метод - це техніка, яка механізує рішення дизайну та зосереджена на run-time поведінці[5]. Один з основних фокусів даного метода є декомпозиція системи на компоненти, але не на основі функцій. Через це він не дуже добре пристосований до змін, призводить до дублювання поведінки у інших сервісах, а також зв’язує сервіс до якогось конкретного випадку, тим самим ускладнюючи використання сервісу для

іншого випадку. Але взамін він сприяє швидкій декомпозиції, яка інкапсулює ділянки потенційних змін у компоненти. Потім система реалізується як взаємодія між створеними компонентами.

IDesign ділить систему на наступні рівні:

- client layer - слугує для взаємодії з системою
- business logic layer - вміщає в себе бізнес логіку
- resource access layer - слугує для доступу до ресурсів таких, як бази даних, файли чи інші системи
- resource layer - рівень фізичних ресурсів
- utility layer - спеціальний, вертикальний рівень, який зосереджує в собі спільну функціональність, таку як: логування, обробку подій, безпеку. Використовується всіма іншими рівнями.

На відміну від традиційної багаторівневої архітектури, вона додатково розкладає бізнес рівень на компоненти, використовуючи волатильний підхід. Існує два вида компонентів: engines та managers. Engines інкапсулюють в собі бізнес правила і логіку, а managers інкапсулюють послідовності, які необхідні для виконання випадку. Кожен engine працює незалежно один від одного, він не може використовувати інші engine, але спроможний використовувати компоненти з resource access layer. Managers координують engines для того, щоб виконати певний випадок. Один manager може використовувати багато engines і один engine може бути використаний багатьма managers. Managers не можуть викликати інших managers напряму, тільки асинхронно, наприклад через івенти.



Малюнок 4: Вигляд компонентів патерну IDesign

IDesign метод найкраще використовувати разом із DDD коли на рівні обмеженого контексту зроблена декомпозиція на основі волатильності. Це може бути особливо корисним, коли доменна модель всередині контексту є доволі великою і було б добре розбити її на менші компоненти чи модулі. Рівні запропоновані в цьому методі дуже добре підходять рівням DDD описаним Еріком Евансом. Client layer відповідає та абсолютно ідентичний з presentation layer у DDD, resource access layer схожий на infrastructure layer, managers layer - це насправді application layer, а engines ведуть себе як domain layer.

Використовуючи даний метод разом із DDD на рівні обмеженого контексту, кожен контекст захищає свою цілісність за допомогою client і resource access layers. Модель одного обмеженого контексту не може

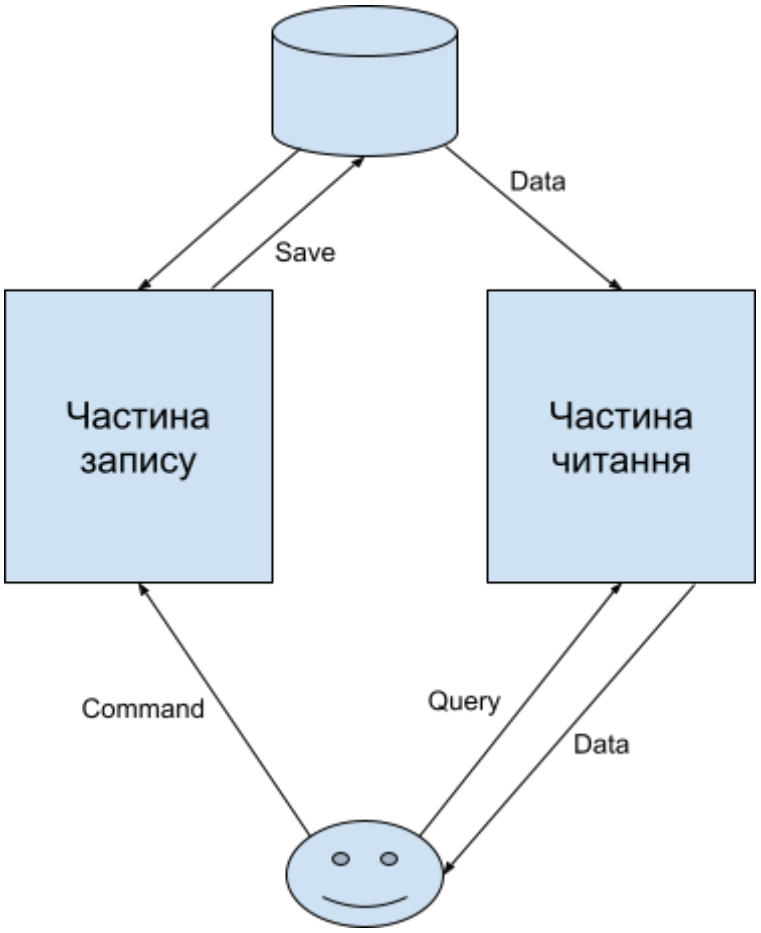
доступитись до моделі іншого контексту, але можна отримати необхідні дані через client і resource access layers. Тут також можна використати патерни для зв’язків між обмеженими контекстами. Наприклад, open host service можна легко визначити як клієнт у client layer, який видає назовні контракт для інших обмежених контекстів. Також можна створити anti-corruption layer як точку доступу до ресурсів у resource access layer.

CQRS

CQRS (command query responsibility segregation) - патерн в основі якого лежить розподіл об’єктів на два типи - ті, які містять методи для команд, що змінюють стан системи, але не повертають жодного значення, і ті, які містять лише методи для запиту даних без жодних побічних модифікацій системи[6].

Ми можемо застосувати CQRS підхід до всієї системи, розділивши її на дві частини: частина запису та частина читання. Це допоможе значно покращити масштабованість у майбутньому. Завдяки такому поділу, ми можемо оптимізовувати обидві частини незалежно одна від одної. У частині запису/команд може бути ефективним використання ORM, так як у частині читання/запитів можна використати raw SQL чи навіть збережені процедури для того, що оптимізувати запити. Більше того, можна частину запису і читання розділити фізично на різні машини та масштабувати горизонтально. Щоб отримати ще кращу продуктивність, можна розділити взагалі базу даних на дві. Такий підхід дозволяє використовувати різні схеми та нормальні форми, залежно від того, що більше і краще підходить під ту чи іншу ситуацію.

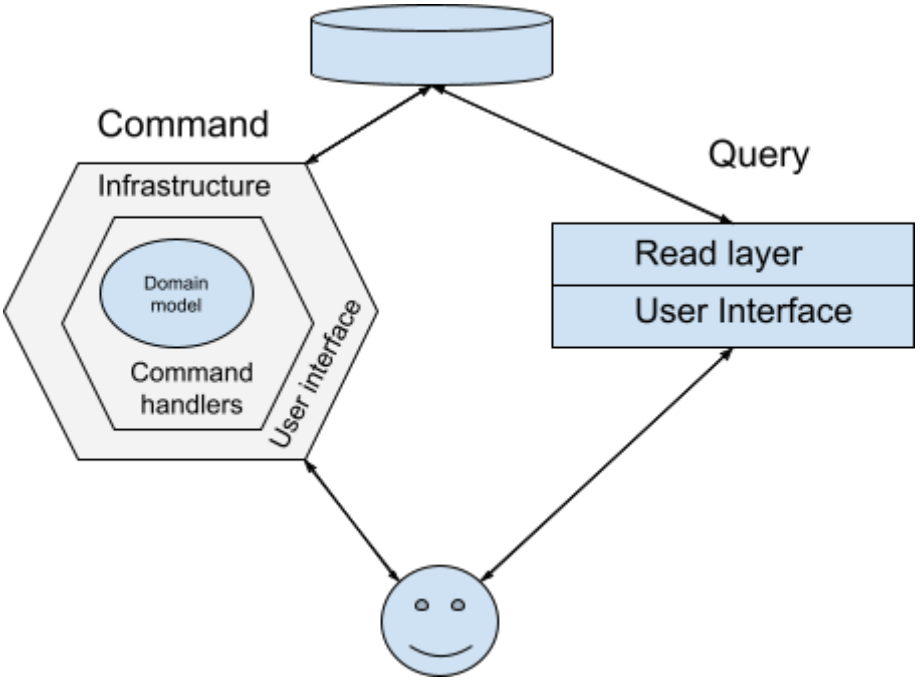
Попри велику кількість переваг CQRS, даний патерн надає складності проекту. Тому перед тим як імплементувати його, слід ретельно подумати чи вартує воно тих зусиль. Багато доменів не мають власної незалежної логіки запитів чи модифікації. Часто є достатнє перекриття для обох частин, щоб залишити їх об’єднаними разом. Також багато систем скоріш за все не потребують такої маштабованості, яку пропонує CQRS.



Малюнок 5: CQRS

Через те що CQRS ділить систему на дві зазвичай великих частини, потрібно дизайнити архітектуру окремо під кожну частину. Наприклад, для частини запису можна використати шестикутну архітектуру, а для частини читання в середньому достатньо простої багаторівневої архітектури.

Цей підхід може мати кілька переваг для систем які використовують DDD. Найбільшою перевагою є можливість розглядати доменну модель з точки зору поведінки. Доменні об’єкти можуть бути більш сфокусовані на поведінці та бізнес правилах, а не на тому як зберігати чи отримувати дані. Це робить модель більш ближчою до загальновживаної мови та значно краще відображає як доменні експерти використовують програмне забезпечення.



Малюнок 6: CQRS разом з DDD

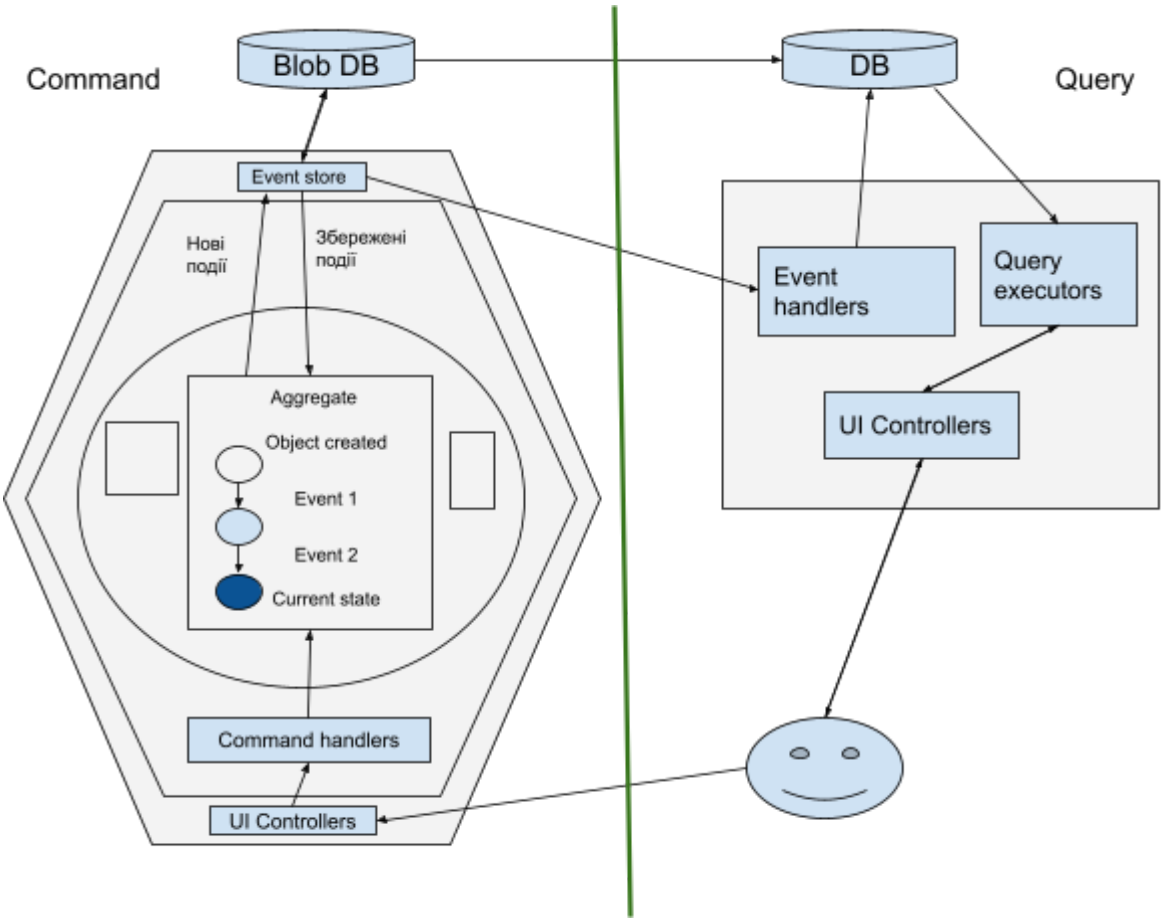
Event sourcing

Event sourcing це архітектурний патерн, який представляє поточний стан системи, як послідовність подій, яка привела до такого стану. Він відрізняється від традиційного підходу, де поточний стан система представлений як знімок даних. Подія - це одна дія, яка вже відбулась в

минулому і називають її зазвичай дієсловом в минулому часі. Зберігаються події у спеціальному сховищі для подій, в якому лише можна добавити подію в кінець списку.

Можливість відтворення всіх подій, дає нам можливість відтворити будь яку ситуацію з історії системи. Це і є найбільша перевага даного підходу. Також з даним підходом дуже легко виправляти помилки, адже ми можемо відкотити систему до цієї точки коли виникли проблеми і відтворити лише правильні події. З технічної точки зору event sourcing також приносить у проект більшу масштабованість. Так як ми лише добавляємо івенти у базу даних, нам не потрібно ніяких блокувань табличок і тд.

Такий підхід також приносить складнощі проекту під час розробки. По-перше, потрібно думати як будуть отримуватись дані, коли користувач надсилає запит на них. Вони не можуть отримуватись напряму з сховищ івентів, оскільки це буде дороговартісною операцією будувати стан системи через івенти по запиту користувача. По-друге, щоб отримувати поточний стан системи потрібно постійно відтворювати усі івенти, які відбулись за всю історію. Це значний удар по продуктивності і займає багато часу та ресурсів. Але цю проблему можна обійти, для прикладу, за допомогою механізму збереження знімків системи. Допустимо ми раз в день відтворюємо стан системи через івенти які відбулися, зберігаємо цей стан, та потім всі подальші запити на стан системи відтворюємо як: сьогоднішній знімок системи + події, які відбулись сьогодні. Іншою проблемою є підтримка версій подій, яка теж потребує значних зусиль при розробці.



Малюнок 7: Дуже абстрактна діаграма event sourcing з використанням CQRS та DDD

Архітектура верхнього рівня

В цьому розділі описуються архітектурні патерни, які можуть використовуватись на найвищих рівнях системи. Розглядаються лише складні системи з декількома обмеженими контекстами, оскільки для простої системи з лише одним обмеженим контекстом ці ідеї не потрібні. Для них можна використати патерни згадані у минулих розділах.

Монолітна архітектура

При побудові системи, найпростішим способом є - покласти все в одне місце. Такий підхід створює моноліт. Переваги такого паттерну - простота і швидкість у розробці системи, особливо на початковій фазі. Також, так як немає затримок комунікації всередині системи спричинених мережею[7].

Проте, так як все зберігається в одному місці, є й багато недоліків такого патерну. Якщо не бути дуже уважним, то легко зробити код взаємопов’язаним та взаємозалежним. Також, коли все в одному юніті, важко горизонтально масштабувати систему. Неможливо добавити іншу машину для покращення продуктивності. Під час деплою, система деякий час буде недоступною для користувачів.

З точки зору DDD, потенційною небезпекою монолітної архітектури є те, що всі обмежені контексти зосереджені у одному місці та немає жодних фізичних границь між ними.

SOA

Важко точно визначити, що означає Service-oriented architecture (SOA) через те, що термін став дуже неоднозначним і для різних людей означає різні речі[8].

З технічної точки зору, SOA розділяє функції у окремі блоки, сервіси. Сервіси та їх користувачі спілкуються разом по мережі, передаючи дані чітко у визначеному форматі[9].

В іншому визначенні використовуються більш конкретні терміни, що описують його як архітектурний стиль для побудови систем, заснований на взаємодії слабо зв'язаних та автономних сервісів, кожен з яких має певні контракти для комунікації з ним[10].

Хоч не існує точних галузевих стандартів для складу SOA, але принципи та практики для проектування сервісів все ж таки є[11]:

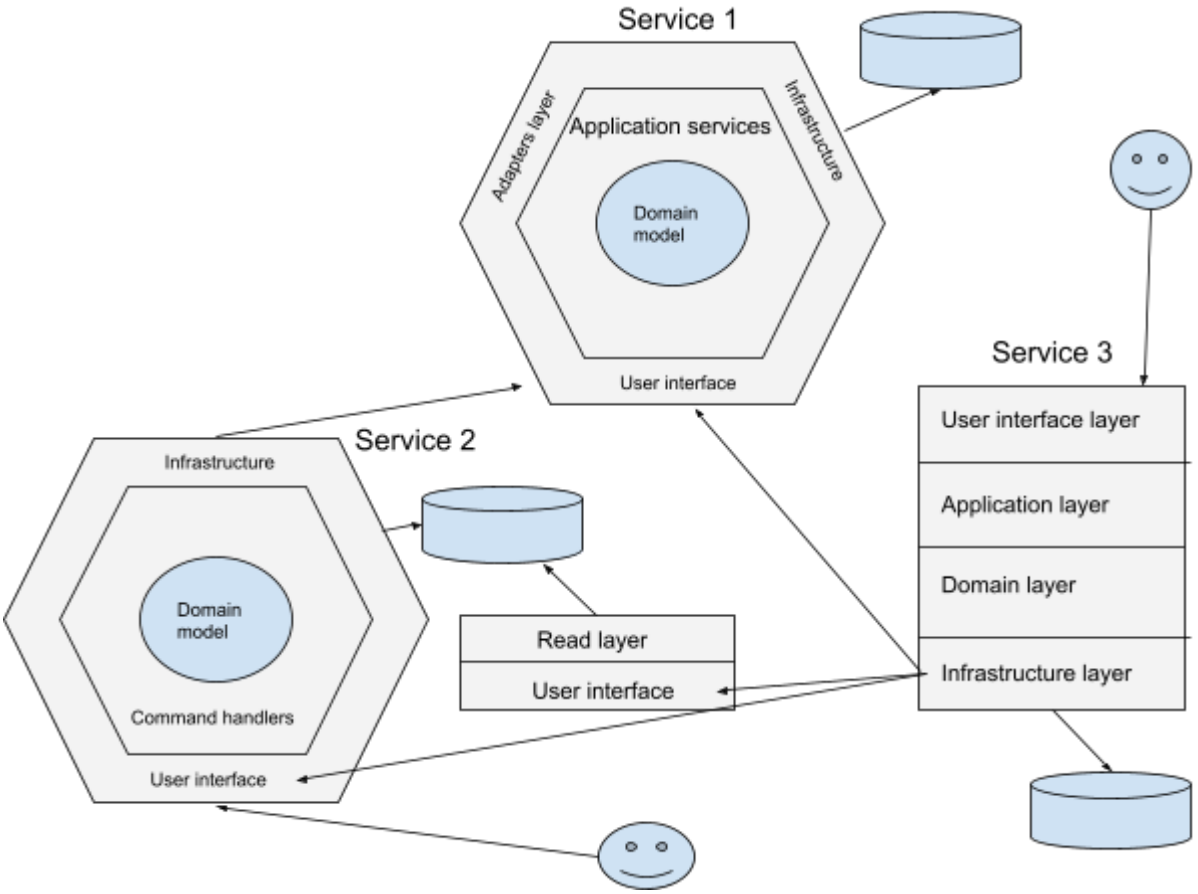
- standardized contract - сервіси показують свою функціональність через контракти, яких повинні дотримуватись інші сервіси для спілкування з ним.
- loose coupling - сервіси повинні мати мінімальну залежність один між одним
- abstraction - сервіси повинні інкапсулювати внутрішню імплементацію і видавати назовні лише контракт.
- reusability - важливо мати змогу перевикористати сервіс у різних сценаріях
- autonomy - сервіси є незалежними від інших і самі контролюють своє середовище
- statelessness - сервіси не керують станом
- discoverability - сервіси повинні бути добре і чітко описані

Назви деяких принципів вже говорять про переваги SOA підходу, такі як: слабка зв’язність між сервісами та можливість перевикористання. Є й інші переваги даного патерну:

- маштабованість
- доступність
- гнучкість

SOA також може використовуватись разом з DDD для декомпозиції системи у сервіси, де кожен сервіс є одним обмеженим контекстом. Через це, ми можемо імплементувати різні архітектури обмежених контекстів у різних сервісах. Для одного обмеженого контексту ми можемо вибрати класичну багаторівневу архітектуру, домен у іншому обмеженому контексті може підходити для event sourcing паттерну, а загальний домен й взагалі може бути реалізований як big ball of mud. Завдяки контракту сервісу, що, до речі і є реалізацією open host service паттерну, архітектура зберігається в середині середині сервісу, а самп імплементація не потрапляє у інші сервіси.

Через те, що SOA надає системі додаткової складності, такий підхід краще пасує для великих систем з багатьма обмеженими контекстами, які в результаті будуть окремими сервісами.



Малюнок 8: Обмежені контексти, організовані за допомогою різних архітектур у сервісах з SOA

Інтеграція обмежених контекстів

У минулих розділах архітектури, які можна використати для обмежених контекстів, були описані, як архітектури високого рівня. Але ще досі не зрозуміло який зв’язок між контекстами та яким чином вони будуть інтегровані для реалізації випадків, які охоплюють більше контекстів. У цьому розділі сама інтеграція буде описано коротко, адже це швидше технічний аспект, ніж аспект домену. Немає жодних потреб для

інтеграцій у монолітному застосунку, цей розділ буде сфокусований тільки на SOA.

Існують 4 головних інтеграційних стилів:

- file transfer - найпростіший спосіб інтегрування сервісів використовуючи файли. Якщо один сервіс хоче надати якусь інформацію іншому, він пише це у файл, який може прочитати інший сервіс. Через те, що сервіси не можуть бути автоматично повідомленні про якусь нову інформацію у файлі, потрібно писати механізм, який буде періодично перевіряти контент файла.
- shared database - тримає дані у стандартизованому вигляді у табличках. Завдяки цьому, сервісам легше споживати дані, тому що робота з базою даних загальновідома і всі сервіси швидше за все матимуть доступ до якоїсь бази даних, тому не прийдеться працювати з різними форматами. Також, набагато легше оновити один рядок бази даних, аніж оновлювати цілий файл - це робить синхронізацію між сервісами більш простою[12].
- remote procedure invocation - сервіс напряму викликає інші сервіси для того, щоб отримати дані чи виконати якусь функцію. Перевагою даного рішення полягає в тому, що кожен сервіс зберігає свою цілісність, керуючи даними самостійно. Реалізація простіша ніж обмін повідомленнями. Недоліком такого підходу є велика зв’язність між сервісами, оскільки для того, щоб одному сервісу викликати функцію іншого сервісу, йому потрібно знати де він знаходиться.
- messaging - схожа ідея як у file transfer, але з спробою вирішити її недоліки. Замість файлів сервіси асинхронно обмінюються маленькими повідомленнями. Сервісу непотрібно знати куди саме він повинен доставити повідомлення. Таке рішення добре підходить

для складних систем з багатьма сервісами, які інтегровані за допомогою івентів.

РОЗДІЛ 4: Дизайн DDD системи

У цьому розділі продемонстровано найбільш важливі архітектурні патерни під час дизайну DDD системи, прикладом якої є псевдо компанія, яка потребує розширення інформаційної системи. Головна ціль цього розділу - показати архітектурні патерни на реальному прикладі. Основний фокус зроблено на патерни, тому деякі речі є спрощеними, наприклад велика частина системи не змодельована взагалі.

Вимоги до системи

Компанія, яка займається онлайн освітою, вже має свою власну інформаційну систему для менеджменту студентів, курсів і тд. Але вона хоче розширити дану систему: добавити онлайн підтримку та онлайн платежі до системи. Стара система мала назву System1, нова буде мати назву System2. Ось головні функціональні вимоги до System2:

- можливість для викладача ділитись матеріалами курсу з студентами через файли або сторінки вікіпедії.
- викладач повинен мати можливість створити тест, який студенти мають пройти
- система повинна мати можливість оцінювати результати тестування автоматично
- система повинна дозволяти студентам спілкуватись у форумах, присвячених кожному курсу
- компанія хоче мати їх власну підсистему для грошових транзакцій.

Також, ось перелік не функціональних вимог до системи: розширюваність, адаптивність, масштабованість, продуктивність.

Аналіз

System1 було розроблено давно і тоді не передбачалось ніякої архітектури. Головний фокус був на тому, щоб швидко добавити нові функції, а якість програмного забезпечення відходила на другий план. Ось чому зараз компанія вирішила виділити більше грошей і задизайнити нові елементи системи краще. Незважаючи на те, що грошей для перебудови старої кодової бази недостатньо, новий код буде розроблений із використанням принципів DDD та відповідних архітектурних патернів.

Після розмов з доменними експертами було визначено шість основних доменів системи:

- обмін знань
- тестування
- оцінювання результатів
- онлайн дискусії
- фінансовий акаунт
- платежі

Так як це є лише розширення системи, то нічого з цього не попадає у головний домен, який у нашому випадку є саме менеджмент студентів та курсів. Все вище перелічене не є частиною головного домену. Онлайн дискусії взагалі є частиною загального домену, адже це відома функціональність у багатьох інших системах.

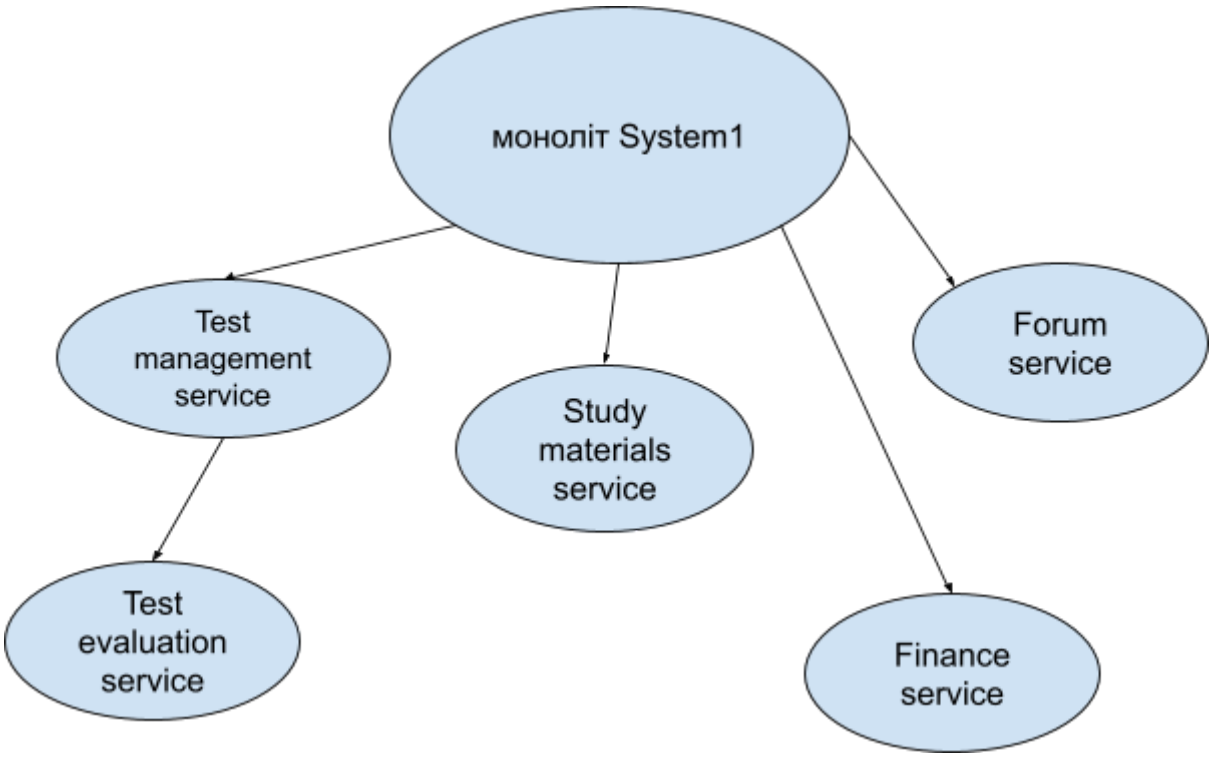
Архітектор намалював межу навколо п’яти обмежених контекстів, які слугуватимуть мовною межею:

- фінанси
- оцінка тесту
- управління тестом
- научні матеріали
- форум

Найвищий рівень архітектури

Архітектор, обираючи найвищий рівень архітектури, думав над двома варіантами - моноліт чи SOA. Оригінал системи є монолітом, який з самого початку сприяв швидшій розробці. Так як масштабованість була важливою вимогою до System2, було обрано SOA. Таким чином контексти з високим трафіком, такі як, наприклад фінансовий контекст, можна буде масштабувати незалежно. Також архітектор запропонував підхід - для кожного обмеженого контексту - окремий сервіс.

Для комунікації між сервісами було вибратно метод RMI (remote method invocation) з використанням REST підходу. Раціональність такого вибору полягає в тому, що небагато нових сервісів буде додано і система залишиться тим же монолітом як і раніше. А отже, більшість комунікації буде відбуватись у синхроному запит/відповідь стилі, що підходить для RMI. Ввести у систему обмін повідомленнями було б набагато складніше, через те що System1 на даний момент не підтримує цього. Незважаючи на те, що було обрано RMI, люба комунікація буде відбуватись у anti-corruption layer, тому як тільки System1 буде переписано на сервіси, можна буде з легкістю замінити спілкування способом RMI на обмін повідомленнями.



Малюнок 9: Загальний вигляд сервісів найвищого рівня

Архітектура обмежених контекстів

Так як ми обрали SOA патерн як архітектуру вищого рівня, ми можемо скористатись його перевагами та використати різні архітектури для різних обмежених контекстів. Це доволі зручно, адже кожний обмежений контекст має різні нефункціональні вимоги, які можуть бути виконані за допомогою різних патернів.

Фінансовий сервіс матиме дві головні функції - отримання балансу та створення транзакції. Через те, що ці дві функції є роздільними: одна читає щось, інша записує, архітектор вирішив використати CQRS підхід, який ділить систему на дві частини: запис - грошові транзакції, читання - отримання балансу.

Також архітектор побачив, що найбільше всяких подій буде відбуватись у цій підсистемі та вирішив використати для частини запису event-sourcing підхід. Це допоможе гарно відображати транзакційний домен, як послідовність подій маніпуляцій з акаунтом, що є тим самим, як доменні експерти бачать транзакції.

Так як даний підхід зберігає події у специфічному форматі, не буде можливості виконувати запити, щоб отримувати напряму інформацію про баланс і тд. Через це, буде створена нова база даних, лише для читання, яка буде оновлюватись як тільки нова подія виникатиме в системі.

Тепер у нас дві бази даних і потрібно думати про їхню консистентність. Є два варіанти:

- strong consistency - досягається тим, що в межах однієї транзакції потрібно буде оновлювати одразу дві бази. Перевагою такого варіанту, є те, що дані будуть оновлені та відображені користувачу одразу. Але недоліком буде саме пропускна здатність.
- eventual consistency - такий варіант робить базу даних для читання консистентною лише в якийсь момент часу, тобто не одразу. Це збільшує пропускну здатність для платежів, але дані не будуть відображені користувачу моментально.

Архітектор вирішив використовувати eventual consistency варіант, так як пропускна здатність платежів є важливою вимогою до системи.

Для test management service архітектор вирішив не використовувати жодних специфічних архітектурних патернів. Він оцінив, що логіку запису/читання є не настільки відділеною, щоб використовувати CQRS. Також розглядалось використання event sourcing, але модель не є орієнтована на події, щоб використовувати такий підхід. Зазвичай більшість об’єктів не змінюватимуть свій стан. IDesign можна було б

використати для майбутньої декомпозиції сервісу на компоненти, але сервіс не є достатньо складним, щоб витрачати такі зусилля. Архітектор думав над шестикутною та багаторівневою архітектурою. Через те, що він хотів мати головний фокус на доменну логіку і зробити інфраструктуру незалежною, він обрав шестикутну архітектуру, яка дозволяє це зробити.

Study materials service має дві головні функції - додавання нових матеріалів викладачем та завантаження цих матеріалів студентами. Архітектор зазначив, що дві частини є достатньо незалежними, та обрав CQRS підхід для цього сервісу. Це зробить читання і запис більш незалежними, а також дасть змогу окремо оптимізовувати підсистеми. Так як, частина читання не буде складною, було вирішено не ускладнювати сервіс та не використовувати event sourcing. Обидві частини читання/запис будуть мати просту рівневу архітектуру для того, щоб відділити користувацький інтерфейс від бізнес логіки.

Для forum service архітектор вирішив використати вже готове існуюче рішення. Компанія не потребує свого власного форуму, їм потрібно лише мати можливіть комунікації користувачів у системі. Для того, щоб не робити інші частини системи залежним від 3rd party компонентів, архітектор вирішив інкапсулювати даний компонент у сервіс.

Висновки

У даній дипломній роботі проведено аналіз Domain-driven design і архітектурних патернів. За допомогою короткого введення до DDD, вдалось розібрати архітектурні патерни і стилі, що розглядаються на рівнях логіки домену, обмеженого контексту та всієї системи.

Повертаючись до логіки домену, варто зазначити, що його модель пропонує найкращі варіанти при застосуванні DDD. Що цікаво, значна кількість різних патернів може використовуватись на рівні обмеженого контексту в залежності від домену. Варто зазначити, що для типового домену використовується шестикутна або багатошарова архітектура. Тимчасом, більш точні домени можуть бути задизайнені як у CQRS, так і event sourcing стилі. Два варіанти було представлено, як архітектура найвищого рівня: монолітна та service-oriented архітектура.

Якщо монолітна архітектура вважається хорошим варіантом для простіших систем з незначною кількістю обмежених контекстів, service-oriented архітектура використовується для більш складних систем з багатьма обмеженими контекстами.

Також, підсумовуючи, було представлено чотири стилі інтеграції, а саме: shared files, shared database, remote procedure invocation та messaging.

У завершальній частині дипломної роботи було розроблено зразкову систем, яка відображала доменні та нефункціональні вимоги, та як вони можуть впливати на вибір архітектурного патерну.

Список посилань

1. FOWLER, Martin. Patterns of Enterprise Application Architecture. Addison - Wesley, 2002.
2. RAJ, Pethuru; RAMAN, Anupama; SUBRAMANIAN, Harihara. Architectural Patterns. Packt Publishing, 2017.
3. BUSCHMANN, Frank; MEUNIER, Regine; ROHNERT, Hans; SOMMERLAD, Peter; STAL, Michael. Pattern-Oriented Software Architecture. Wiley, 1996.
4. VERNON, Vaughn. Implementing Domain-Driven Design. Addison - Wesley, 2014.
5. LÖWY, Juval. Zen of Architecture [online]. 2013 [visited on 10.10.2018]. Available also from: https://youtu.be/Jxm2rgeuC6s.
6. YOUNG, Greg. CQRS, Task Based UIs, Event Sourcing agh! [online]. 2010 [visited on 2018-10-19]. Available also from: http://codebetter. com / gregyoung / 2010 / 02 / 16 / cqrs - task - based - uis - event - sourcing-agh/.
7. ROUSE, Margaret. monolithic architecture [online]. 2016 [visited on 2019-01-15]. Available also from: https : / / whatis . techtarget . com/definition/monolithic-architecture
8. FOWLER, Martin. ServiceOrientedAmbiguity [online]. 2005 [visited on 2019-01-04]. Available also from: https : / / martinfowler . com / bliki/ServiceOrientedAmbiguity.html.
9. BELL, Michael. Service-Oriented Modeling: Service Analysis, Design, and Architecture. Wiley, 2008.
10. ROTEM-GAL-OZ, Arnon. SOA Patterns. Manning Publications, 2012.
11. ERL, Thomas. SOA: Principles of Service Design. Prentice Hall, 2007

12. ERL, Thomas. SOA: Principles of Service Design. Prentice Hall, 2007