Дніпровський національний університет імені ОЛЕсЯ Гончара

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА МАТЕМАТИЧНОГО ЗАБЕЗПЕЧЕННЯ ЕОМ

**РЕФЕРАТ**

з дисципліни «Об'єктно-орієнтоване програмування»

на тему: «Паттерни проектування. Реализація на Си++»

Студента \_\_2\_\_ курсу \_ПЗ-21-2\_ групи

спеціальність\_\_\_121\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Інженерія програмного забезпечення

Гудимова Дениса

(прізвище та ініціали)

м. Дніпро, 2023 р.

1 БАЗОВІ ВІДОМОСТІ ПРО ПАТЕРНИ ПРОЕКТУВАННЯ ТА ЇХНЄ ВИКОРИСТАННЯ.

1.1 Загальна інформація

Паттерни проектування - це відомі рішення проблем проектування програмного забезпечення, які були успішно випробувані та документовані. Вони дозволяють під час розробки програмного забезпечення ефективно використовувати кращі практики та досвід розробників.

Патерни проектування складаються з трьох складових:

* 1. Назва патерну - це коротка назва, яка ідентифікує патерн проектування;
  2. Проблема - це опис проблеми, яку вирішує патерн проектування;
  3. Рішення - це опис рішення, яке пропонує патерн проектування.

Патерни проектування можуть бути поділені на чотири категорії:

* 1. Породжуючи патерни - використовуються для створення об'єктів та класів;
  2. Структурні патерни - використовуються для організації класів та об'єктів;
  3. Поведінкові патерни - використовуються для визначення взаємодії між об'єктами та класами;
  4. Патерни архітектури - використовуються для організації великих систем та підсистем.

Використання патернів проектування має декілька переваг, зокрема:

1. Вони дозволяють розробникам використовувати кращі практики та досвід при розробці програмного забезпечення;
2. Вони сприяють покращенню зрозумілості та підтримці коду;
3. Вони сприяють покращенню ефективності розробки програмного забезпечення.

Однак, не слід зловживати використанням патернів проектування, оскільки вони можуть стати зайвою складовою коду та привести до збільшення складності проекту. Використання патернів проектування має бути обумовлене і розсудливе, а також повинне бути зроблене з урахуванням конкретних потреб та вимог проекту.

1.2 Породжуючі патерни

Найпопулярніші породжуючі патерни включають такі:

* 1. Singleton (Одинак) - цей патерн забезпечує, що клас має тільки один екземпляр, та надає глобальну точку доступу до цього екземпляра;
  2. Factory Method (Фабричний метод) - цей патерн дозволяє створювати об'єкти підкласів за допомогою інтерфейсу базового класу;
  3. Abstract Factory (Абстрактна фабрика) - цей патерн дозволяє створювати сімейства пов'язаних об'єктів без вказування конкретних класів;
  4. Builder (Будівельник) - цей патерн дозволяє створювати складні об'єкти крок за кроком, дозволяючи змінювати процес побудови об'єкта без зміни його структури;
  5. Prototype (Прототип) - цей патерн дозволяє створювати нові об'єкти шляхом копіювання існуючих об'єктів.

Ці патерни дозволяють забезпечувати більш гнучку та ефективну роботу з об'єктами та їх створенням в програмному забезпеченні. Вони забезпечують відділення деталей реалізації від використання та забезпечують більшу гнучкість та можливість змінювати поведінку об'єктів у процесі виконання програми.

1.3 Структурні патерни

Найпопулярніші структурні патерни включають такі:

* 1. Adapter (Адаптер) - цей патерн дозволяє об'єднати дві непов'язані системи, перетворивши інтерфейс однієї системи на інтерфейс іншої системи;
  2. Decorator (Декоратор) - цей патерн дозволяє додавати нові функції до об'єкта без зміни його інтерфейсу;
  3. Facade (Фасад) - цей патерн дозволяє створити простий інтерфейс для складного підсистеми;
  4. Bridge (Міст) - цей патерн дозволяє розділити абстракцію від її реалізації, дозволяючи обидві частини розвиватися незалежно;
  5. Composite (Композит) - цей патерн дозволяє об'єднати об'єкти в деревоподібну структуру, щоб обробляти їх як єдиний об'єкт.

Кожен з цих патернів дозволяє розробникам покращити якість програмного забезпечення, знизити ризики при зміні програмного забезпечення та зменшити складність програми. Використання патернів структури дозволяє створювати більш гнучку та легку в зміні програмну архітектуру.

1.4 Поведінкові патерни

Найпопулярніші поведінкові патерни включають такі:

* 1. Observer (Спостерігач) - цей патерн дозволяє об'єктам відслідковувати стан інших об'єктів і автоматично реагувати на зміни стану;
  2. Strategy (Стратегія) - цей патерн дозволяє визначити сімейство алгоритмів, які можуть використовуватися взаємозамінно, і забезпечує вибір підходящого алгоритму в залежності від потреб програми;
  3. Command (Команда) - цей патерн дозволяє зберігати команди у вигляді об'єктів, що дозволяє виконувати їх асинхронно, створювати черги команд та відмінити виконання команди в будь-який момент;
  4. Template method (Шаблонний метод) - цей патерн дозволяє визначити загальний алгоритм у батьківському класі, а дочірні класи реалізують конкретні кроки алгоритму;
  5. Iterator (Ітератор) - цей патерн дозволяє отримувати доступ до елементів колекції, не розкриваючи її внутрішньої реалізації.

Кожен з цих поведінкові патерни дозволяє розробникам зменшити зв'язаність між класами та дозволяє більшу гнучкість в реалізації функціоналу програми. Використання поведінкові патерни дозволяє створювати більш просту та ефективну архітектуру програмного забезпечення.

1.5 Патерни баз даних

Існує кілька патернів для баз даних, які можуть забезпечити більш ефективну та гнучку роботу з даними. Найпопулярніші з них:

* 1. Active Record - цей патерн дозволяє створювати об'єкти, які взаємодіють з базою даних без необхідності використовувати окремий шар доступу до даних;
  2. Data Mapper - цей патерн розділяє базу даних від об'єктів в програмі, забезпечуючи зручну взаємодію з даними;
  3. Repository - цей патерн дозволяє розділити логіку доступу до даних від логіки бізнес-логіки, що забезпечує більшу гнучкість та повторне використання коду;
  4. Unit of Work - цей патерн забезпечує збереження змін в об'єктах бази даних з однієї транзакції, що забезпечує більшу надійність та цілісність даних.

Ці патерни можуть допомогти забезпечити більш ефективну та гнучку роботу з базами даних, що є важливим в сучасному програмуванні, де даних стає все більше і більше

В цілому, патерни проектування є важливим інструментом для розробки якісного та підтримуваного програмного забезпечення. Використання патернів дозволяє розробникам ефективно використовувати досвід та кращі практики у процесі розробки, а також сприяє поліпшенню зрозумілості та підтримці коду. Однак, використання патернів повинно бути обдуманим та зрозумілим, а також має враховувати конкретні потреби та вимоги проекту.

2 ДОКЛАДНІ ВІДОМОСТІ ПРО ПАТЕРНИ, ЯКІ БУЛИ РЕАЛІЗОВАНІ ЗГІДНО ІНДИВІДУАЛЬНОГО ВАРІАНТУ.

Згідно індивідуального завдання потрібно було реалізувати патерни Декоратор(Decorator) та Репозиторій(Repository)

Для реалізації патерну Decorator з курсової роботи було залишено фігури квадрат та коло. На основі абстрактного класу(інтерфейсу) Figure був створений клас FigureDecorator який реалізує усі методи абстрактного класу і має поле типу Figure. Також було створено 3 класи : InternalColorDecorator, OutlineColorDecorator, RotateDecorator які успадковують клас FigureDecorator у кожному класі конструктор реалізований таким чином що параметр це абстрактний клас Figure, а також перевизначено метод change\_color(sf::Color color). Цей метод у всіх 3 класах Decorator реалізований таким чином, що він робить свою задачу незалежно від інших і викликає цей же метод для вкладеного об’єкту і так доки цей метод послідовно не відпрацює для усіх об’єктів які загорнуті у середину.

Для реалізації патерну Repository було створено новий проект у якому створені класи User який зберігає інформацію про користувача, FileWriter котрий записує у json файл інформацію про користувача, Repository абстрактний клас, UserRepository наслідує Repository та реалізує усі методи класу Repository. Такія як “add\_user”, “find\_by\_id”, “get\_older\_than”, “get\_all\_user” для додавання пошуку та виводу користувачів. Клас UserRepository створюєтьяс для того щоб не порушувати концепт ООП і не повторювати поширені запроси і не порушувати концепт DRY коли потрібно викликати запроси із різних місць. Отже все працює таким чином що код пишеться в одному класі і коли будуть потрібні запити до бази даних будуть викликатися методи, а не писатиметься запит ще раз.

2.1 Патерн Декоратор(Decorator)

Патерн декоратор (Decorator) - це структурний патерн проектування, який дозволяє динамічно додавати нову функціональність до існуючих об'єктів, обгортаючи їх в об'єкти-декоратори. Цей патерн дає можливість розширювати функціональність об'єктів на льоту, не змінюючи їхньої структури.

Проблема, яку вирішує цей паттерн, полягає в тому, що часто потрібно додати до об'єкта додаткову функціональність, але змінювати основну структуру об'єкта не можна або не бажано. Наприклад, якщо ми маємо об'єкт-продукт, може знадобитися додати до нього додаткові властивості, такі як кольори, форми або розміри, але змінювати основну структуру продукту не можна.

Рішення полягає в тому, що ми створюємо обгортки (декоратори), які додають додаткову функціональність до основного об'єкта. Кожен декоратор реалізує той самий інтерфейс, що і основний об'єкт, тобто має ті ж методи і властивості. При цьому, він також має посилання на основний об'єкт, який він доповнює. Таким чином, можна динамічно додавати декілька декораторів, які додають різні функціональність до одного і того ж об'єкта.

Аналогія з життя: людина це основний об'єкт, а додаткові складові, такі як штани, футболка, светр, шарф, ремінь, капелюх або брошка, можуть бути декораторами. Якщо ви хочете додати до свого одягу додаткову функціональність, наприклад, зробити його теплішим, більш стильним або елегантнішим, ви можете вибрати кілька декораторів, щоб додати до свого одягу ту саму додаткову функціональність, яку ви хочете, і при цьому не змінювати основну структуру (дивитись рисунок 2.1).

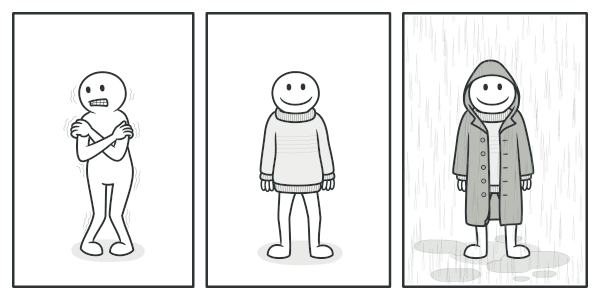


Рисунок 2.1 – Патерн Декоратор анологія з одягом

Структура патерну декоратор складається з базового класу Component, який визначає базові методи, що будуть реалізовані декораторами. Також існують конкретні компоненти (ConcreteComponent), які реалізують базовий функціонал.

Класи-декоратори (Decorator) виконують ті ж базові функції, що і класи ConcreteComponent, але додають до них свою власну функціональність. Крім того, декоратор може мати посилання на об'єкт Component, тому декоратори можуть легко зв'язуватися між собою в ланцюжок (дивитись рисунок 2.2).

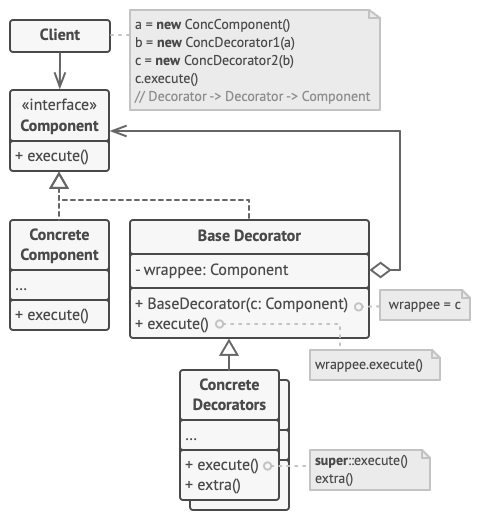


Рисунок 2.2 – Структура патерну Декоратор

Приклад використання патерну декоратор може бути в редакторі тексту, де ми можемо мати базовий клас для тексту, до якого можна додавати різні декоратори, такі як декоратор для вирівнювання тексту, декоратор для кольорового виділення, декоратор для підсвічування тексту тощо.

Переваги використання патерну декоратор:

* 1. Дозволяє додавати функціональність до об'єктів без зміни їхньої структури;
  2. Можливість динамічного додавання та видалення декораторів;
  3. Розширення функціональності можливе без створення великої кількості підкласів;
  4. Забезпечує розділення обов'язків та їхню ієрархію.

Недоліки використання патерну декоратор:

* 1. Можлива перевантаженість коду через велику кількість декораторів;
  2. Складність розуміння декораторів може бути високою для початківців;
  3. Може бути важко підтримувати систему, де кілька декораторів одночасно працюють з об'єктом.

У загальному, патерн декоратор дозволяє розширювати функціональність програми, не порушуючи її основну структуру. Це дозволяє підтримувати код в доброму стані та збільшувати його ефективність без ризику порушити функціональність програми.

2.1 Патерн Репозиторій (Repository)

Патерн Репозиторій (Repository) є одним з найбільш поширених патернів архітектури програмного забезпечення. Він є частиною підходу розділення відповідності (Separation of Concerns), що полягає в тому, щоб відокремити логіку бізнес-логіки від деталей збереження даних. Патерн Репозиторій забезпечує більш гнучкий та простий доступ до даних та дозволяє працювати з ними, не залежно від механізму збереження.

Проблема, яку вирішує патерн Репозиторій, полягає у тому, що зазвичай інформація зберігається в базах даних або інших зберігаючих механізмах, тоді як бізнес-логіка зазвичай залежить від цієї інформації. Це може привести до того, що бізнес-логіка стає залежною від конкретного механізму збереження даних, що у свою чергу може ускладнити підтримку та розвиток проекту.

Рішення, яке пропонує патерн Репозиторій, полягає в тому, що дані зберігаються в окремому компоненті - репозиторії. Цей компонент забезпечує необхідний інтерфейс для доступу до даних, що приховує деталі збереження даних. Таким чином, бізнес-логіка може взаємодіяти з репозиторієм, не знаючи, як саме дані зберігаються .

Аналогія з життя: Репозиторій можна порівняти з бібліотекою, де книжки зберігаються в одному місці, а люди можуть брати їх напрокат, не звертаючи уваги на те, де саме книжки зберігаються і як саме організована бібліотека. Таким чином, люди можуть зосередитися на читанні та вивченні книжок, не звертаючи уваги на деталі їх збереження (дивитись рисунок 2.3).

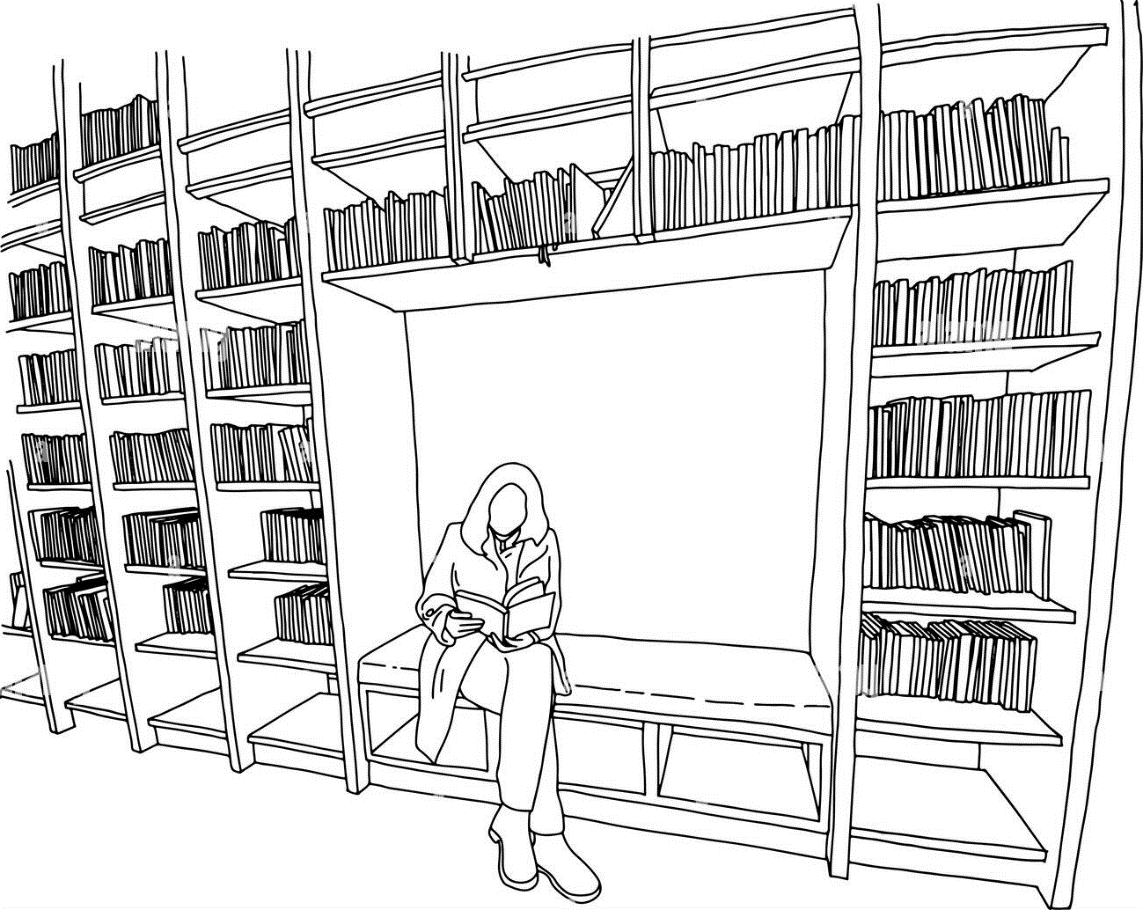


Рисунок 2.3 – Патерн Репозиторій анологія з бібліотекою

Основні переваги використання патерну Репозиторій включають:

* 1. Покращення гнучкості та простоти доступу до даних;
  2. Відокремлення бізнес-логіки від деталей збереження даних;
  3. Зниження кількості залежностей між класами в програмі.

Хоча патерн Репозиторій має свої переваги, він також має кілька недоліків, серед яких можна виділити наступні:

* 1. Потенційно великий розмір класу. Якщо програма має багато типів об'єктів, то клас репозиторію може стати дуже великим та складним для управління;
  2. Надмірна абстракція. Занадто високий рівень абстракції може призвести до того, що код стає складним для розуміння та управління;
  3. Специфічність використання бо Патерн Репозиторій не є універсальним рішенням і може бути непридатним для деяких випадків використання;
  4. Потенційна втрата продуктивності. Якщо програма має об'єкти великого обсягу або велику кількість даних, то робота з репозиторієм може призвести до затримок у продуктивності.

Основні складові патерну Репозиторій (дивитись рисунок 2.4):

* 1. Інтерфейс репозиторію - описує методи, що необхідні для роботи з даними;
  2. Клас конкретного репозиторію - реалізує методи, описані в інтерфейсі репозиторію. Він містить код для взаємодії з механізмом збереження даних;
  3. Клас сутності - описує об'єкт, який буде зберігатися в базі даних. Цей клас може містити різноманітні властивості.

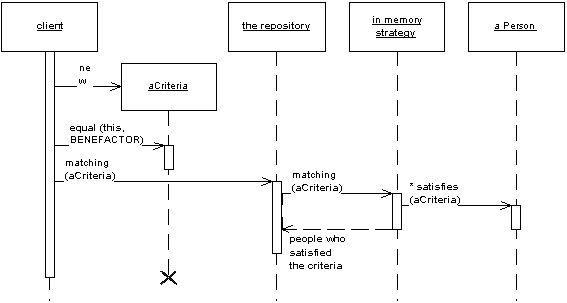


Рисунок 2.4 – Структура патерну Репозиторій

Також про Репозиторій можна сказати, що це — досить складне типове рішення, також у його реалізації зазвичай використовують кілька інших типових рішень. Насправді воно нагадує невеликий фрагмент об'єктно-орієнтованої бази даних і цим подібно до об'єкта запиту, який частіше набувають у складі комерційних засобів об'єктно-реляційного відображення, ніж розробляють самі. Репозиторій в поєднанні з об'єктом запиту суттєво збільшує можливості шару об'єктно-реляційного відображення.

Незважаючи на складні механізми, реалізовані всередині Репозиторія, його інтерфейс дуже простий. Клієнт створює об'єкт критеріїв, вказуючи характеристики об'єктів, які мають бути повернуті внаслідок виконання запиту. Наприклад, щоб знайти співробітників із зазначеним ім'ям, необхідно задати об'єкт критеріїв, описуючи кожен Критерій приблизно так: criteria, equals (Person. LAST\_NAME, "Fowler") і criteria, like (Person. FIRST\_NAME, "M"). Потім потрібно викликати метод repository.matching (criteria), щоб повернути список співробітників із прізвищем Fowler, ім'я яких починається на літеру "М". Для більшої зручності в абстрактному класі Репозиторій можна визначити й інші методи; наприклад, для пошуку єдиної відповідності було б непогано скористатися методом на кшталт soleMatch (criteria), який повертав би об'єкт, а не колекцію об'єктів. До числа інших поширених методів входить метод пошуку за ідентифікатором byObject id (id), який зовсім нескладно реалізувати, використовуючи метод soleMatch.

Клієнтський код, що працює із репозиторієм, сприймає його як просту колекцію об'єктів домену, розташовану в оперативній пам'яті. Той факт, що об'єкти домену насправді не знаходяться у Репозиторії, повністю прихований від клієнтських програм. Зрозуміло, що клієнтський код має бути поінформований про те, що "видима" колекція об'єктів (наприклад, товарів, що розповсюджуються за каталогом) може відображатися на таблиці із сотнями тисяч записів, і застосування методу all до Репозиторія ProductRepository - не вдала ідея.

Методи Репозиторію замінюють спеціалізовані методи пошуку перетворювача даних, виконуючи вилучення об'єктів на основі специфікацій. Порівняйте це з безпосереднім використанням об'єкта запиту, у якому клієнт створює об'єкт критеріїв (простий різновид типового рішення специфікація (Specification)), додає цей критерій безпосередньо до об'єкта запиту та виконує запит. При роботі із Репозиторієм клієнтський код формує критерії та передає їх Репозиторію із проханням повернути об'єкти, які підійдуть під заданий опис. Отже, з погляду клієнта, виконання запиту замінюється вилученням об'єктів, що задовольняють специфікації запиту. Ця відмінність добре ілюструє описову природу взаємодії об'єктів у Репозиторії, що становить значну частину сутності останнього.

Внутрішня структура Репозиторію є поєднанням відображення метаданих (Metadata Mapping) з об'єктом запиту, яке застосовується для автоматичної генерації SQL-коду на основі заданих критеріїв. Те, який об'єкт відповідатиме за додавання критеріїв до об'єкта запиту — об'єкт критеріїв, об'єкт запиту або саме відображення метаданих повністю залежить від переваг розробника. Джерело об'єктів для Репозиторію зовсім не обов'язково має бути реляційним базою даних. Це цілком допустимо, тому що Репозиторій легко пристосувати до зміни шару відображення у вигляді спеціальних об'єктів стратегій. Тому таке типове рішення особливо добре підходить для систем, що працюють з кількома схемами баз даних або об'єктів домену, що мають кілька джерел. Стане в нагоді Репозиторій та для тестування, яке дуже бажано проводити тільки на об'єктах, розташованих в оперативній пам'яті, щоб уникнути проблем зі швидкістю доступу.

Репозиторій може послужити хорошу службу у покращенні читабельності та ясності коду, що інтенсивно використовує механізм запитів. Наприклад, система, заснована на використанні оглядача і містить безліч сторінок запитів, потребує чіткого механізму перетворення об'єктів HttpRequest на результати запиту. За наявності сховища коду обробника запитів буде набагато легше перетворити об'єкт HttpRequest в об'єкті критеріїв; передача критеріїв відповідному сховищу вимагатиме лише двох-трьох додаткових рядків коду.

Враховуючи усе вищеперелічене, перед використанням паттерна Репозиторій потрібно ретельно оцінити можливі ризики та переваги, що він може принести у конкретному випадку.