Чернівецький національний університет імені Юрія Федьковича Факультет математики та інформатики

Кафедра математичного моделювання

**Лабораторна робота №4**

з навчальної дисципліни: “Проектування програмних систем”

Застосування патернів програмування

Виконав: студент 3 курсу

301 групи

спеціальності “Комп’ютерні науки”

Малованюк С.М

Перевірила: доцент Піддубна Л.А.

Варіант №26

**Чернівці – 2025**

***Завдання***

*Аналізуючи діаграми класів (Лабораторна робота №3), обґрунтувати застосування підібраних для реалізації патернів.*

**У діаграмі класів використані наступні патерни проєктування:**

1. **Шаблонний метод (Template Method)**
   * **Опис:** Визначає скелет алгоритму в операції суперкласу, дозволяючи підкласам перевизначати певні кроки цього алгоритму без зміни його загальної структури.
   * **Місце реалізації та спосіб:**
     + **Ієрархія Користувач:** Абстрактний клас Користувач (абстрактний) визначає загальний шаблон для операцій, таких як login(), logout(), updateProfile(), changePassword(), оголошуючи їх як абстрактні. Конкретні підкласи (Власник, Адміністратор, Працівник) надають специфічну реалізацію цих кроків, дотримуючись загальної структури, визначеної в базовому класі.
     + **Ієрархія Ресурс:** Аналогічно, абстрактний клас Ресурс (абстрактний) визначає шаблон для роботи з ресурсами через абстрактні методи updateQuantity() та checkThreshold(). Конкретний підклас Корм реалізує ці методи відповідно до своєї специфіки.
   * **Обґрунтування:**
     + **Уникнення дублювання коду:** Загальна логіка алгоритму виноситься в базовий клас.
     + **Чітка структура:** Усі підкласи дотримуються однакової послідовності кроків для певних операцій.
     + **Гнучкість та розширюваність:** Дозволяє підкласам кастомізувати окремі етапи алгоритму, не впливаючи на його загальну структуру, що спрощує додавання нових типів користувачів або ресурсів з унікальною поведінкою на певних етапах.
2. **Фабричний метод (Factory Method)**
   * **Опис:** Визначає інтерфейс для створення об'єкта (у моєму випадку, результату операції, що діє як "продукт", тобто тут впроваджений не зовсім фабричний метод, але суть та сама), дозволяє підкласам вирішувати, екземпляр якого саме класу (або яку саме логіку "виробництва") використовувати.
   * **Місце реалізації та спосіб:**
     + **Клас Користувач (абстрактний):** Оголошує абстрактний метод generateAccessToken(): String. Цей метод виступає як фабричний метод (для фабричного методу тут мав би використовуватись інтерфейс замість String).
     + **Підкласи Власник, Адміністратор, Працівник:** Кожен з цих класів надає власну конкретну реалізацію методу generateAccessToken(). Таким чином, кожен тип користувача "виробляє" токен доступу за власною, специфічною для його ролі логікою.
   * **Обґрунтування:**
     + **Делегування логіки створення:** Відповідальність за конкретну логіку генерації токенів переноситься на підкласи.
     + **Гнучкість системи:** Дозволяє легко додавати нові типи користувачів з унікальними способами генерації токенів без необхідності змінювати клієнтський код, який запитує токен. Клієнт просто викликає user.generateAccessToken(), а конкретна реалізація визначається типом об'єкта user.
3. **Фасад (Facade)**
   * **Опис:** Надає спрощений уніфікований інтерфейс до набору інтерфейсів у підсистемі. Фасад робить підсистему простішою у використанні.
   * **Місце реалізації та спосіб:**
     + **Клас Система:** Виступає в ролі фасаду. Він надає набір методів (authenticate(user: User), sendNotification(message: String), generateBackup(): Boolean, logAction(action: String), checkSystemHealth(): String), які слугують єдиною точкою входу для виконання відповідних операцій.
     + Ці методи приховують внутрішню складність взаємодії різних компонентів системи. Наприклад, метод authenticate(user: User) інкапсулює логіку автентифікації, яка може включати перевірку облікових даних, роботу з базою даних користувачів, генерацію сесій тощо, використовуючи об'єкти типу Користувач (як показано залежністю "Auth").
   * **Обґрунтування:**
     + **Спрощення інтерфейсу:** Зменшує складність взаємодії з комплексною системою для клієнтського коду.
     + **Зменшення зв'язності:** Ізолює клієнтів від внутрішньої реалізації підсистеми, що дозволяє змінювати внутрішні компоненти без впливу на клієнтів, доки інтерфейс фасаду залишається незмінним.
     + **Покращення структури:** Організовує доступ до функціональності системи через чітко визначену точку входу.

**Патерни, що можна впровадити:**

1. **Спостерігач (Observer)**
   * **Опис:** Визначає залежність "один-до-багатьох" між об'єктами таким чином, що при зміні стану одного об'єкта всі залежні від нього об'єкти (спостерігачі) автоматично сповіщаються та оновлюються.
   * **Ймовірне місце застосування та спосіб:**
     + **Сповіщення системи:** Метод Система.sendNotification(message: String) може бути реалізований так, що будуть відстежуватись зміни в класі Система, а різні компоненти або користувачі будуть спостерігачами, що підписані на отримання певних сповіщень.
     + **Оновлення статусу завдань/ресурсів:** Зміна статусу в Завдання.updateStatus() або досягнення мінімального значення в Ресурс.checkThreshold() може ініціювати сповіщення, наприклад, для Адміністратора, Власника.
   * **Обґрунтування потенційного застосування:**
     + **Слабка зв'язність:** Дозволяє об'єктам взаємодіяти без жорсткої прив'язки, що робить систему більш гнучкою та легкою для модифікації.
     + **Автоматизація оновлень:** Забезпечує автоматичне інформування залежних об'єктів про важливі зміни.
2. **Стан (State)**
   * **Опис:** Дозволяє об'єкту змінювати свою поведінку при зміні його внутрішнього стану.
   * **Ймовірне місце застосування та спосіб:**
     + **Управління станами Завдання:** Атрибут status: String у класі Завдання передбачає, що поведінка завдання (наприклад, доступні операції, переходи між станами) залежить від його поточного стану ("нове", "у роботі", "виконане"). Кожен стан може бути інкапсульований в окремий клас, що реалізує спільний інтерфейс стану. Об'єкт Завдання делегуватиме виконання операцій поточному об'єкту-стану.
     + **Управління станами Тварина:** Аналогічно, атрибут status: String у класі Тварина може вказувати на різні стани ("здорова", "хвора", "на лікуванні"), кожен з яких може мати специфічну поведінку або доступні дії.
   * **Обґрунтування потенційного застосування:**
     + **Інкапсуляція поведінки стану:** Логіка, специфічна для кожного стану, зосереджується в окремих класах, що покращує читабельність та керованість коду.
     + **Уникнення складних умовних конструкцій:** Замінює великі блоки if-else або switch-case.
     + **Спрощення додавання нових станів:** Нові стани та пов'язана з ними поведінка можуть бути легко додані шляхом створення нових класів стану.

*GitHub link to repo - https://github.com/LUMA002/Program\_System\_Design*