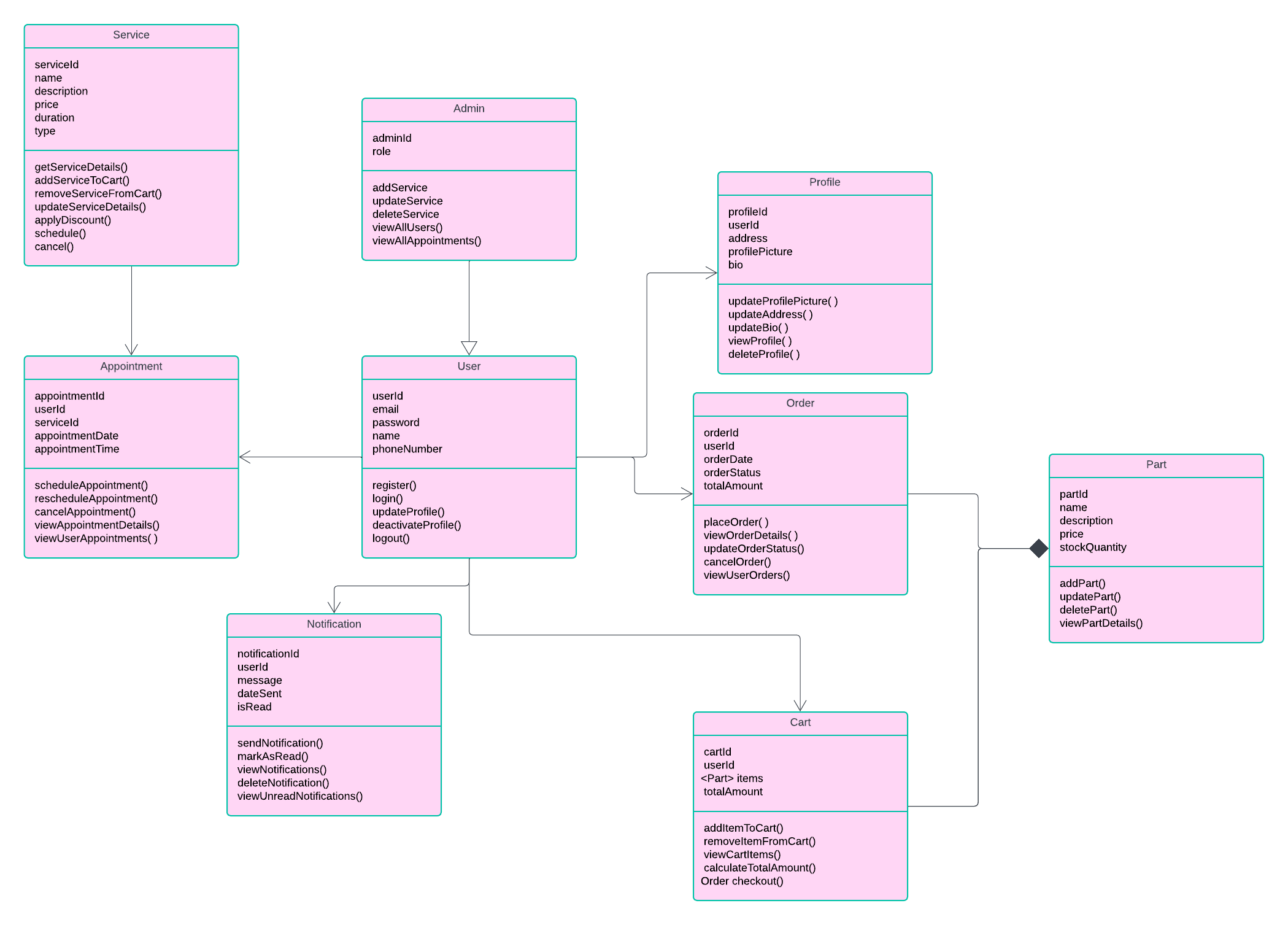
**Звіт до лабораторної роботи №4**

**«Застосування патернів програмування»**

**Завдання**

1. Аналізуючи діаграми класів (Лабораторна робота №3), обґрунтувати застосування підібраних для реалізації патернів.
2. Використовуючи лабораторні роботи із дисциплін професійної підготовки, створити репозиторій, розмістити файли проектів. Надати доступ викладачам на github/gitlab ресурсі.

Предметна область: СТО



GitHub репозиторій: https://github.com/sasha-mykhailk/Design-of-Software-Systems-PPS-.git

### 1. Клас Service

Проблема:

* Управління створенням різних типів послуг може бути ускладнене, оскільки кожен тип послуги може мати свої специфічні параметри та методи. Це ускладнює підтримку та розширення системи.

Рішення: Патерн: Factory Method (Фабричний метод)

* Фабричний метод дозволяє створювати об'єкти без вказання конкретних класів об'єктів, які будуть створені. Це спрощує додавання нових типів послуг.
* Використання фабричного методу дозволяє делегувати процес створення об'єктів підкласам, що забезпечує гнучкість у створенні об'єктів.

Приклад застосування:

* Введення абстрактного базового класу або інтерфейсу ServiceFactory з методом createService().
* Реалізація підкласів для кожного типу послуг (CleaningServiceFactory, MaintenanceServiceFactory), які реалізують метод createService() для створення відповідного об'єкта.

2. Клас User

Проблема:

* Керування сесіями користувачів та їх авторизацією може бути складним, особливо якщо сесії повинні зберігатися і керуватися централізовано.

Рішення: Патерн: Singleton (Одинак)

* Патерн Singleton забезпечує створення лише одного екземпляра класу та надання глобальної точки доступу до нього.
* Це допомагає централізувати керування сесіями користувачів, забезпечуючи єдиний екземпляр класу UserSession, який керує автентифікацією та сесіями користувачів.

Приклад застосування:

* Введення класу UserSession, який містить методи для входу, виходу та отримання поточного користувача.
* Забезпечення того, що UserSession має лише один екземпляр через приватний конструктор та статичний метод getInstance().

3. Клас Order

Проблема:

* Замовлення може мати багато параметрів, що ускладнює його створення та ініціалізацію, особливо якщо параметри змінюються або додаються нові.

Рішення: Патерн: Builder (Будівельник)

* Патерн Builder розділяє процес побудови складного об'єкта на окремі кроки, дозволяючи поступово створювати об'єкт.
* Це дозволяє створювати об'єкти з багатьма параметрами без необхідності мати конструктор з великою кількістю параметрів, роблячи код читабельнішим та підтримуваним.

Приклад застосування:

* Введення внутрішнього класу OrderBuilder в клас Order, який містить методи для встановлення значень параметрів.
* Метод build() в класі OrderBuilder створює і повертає об'єкт Order.

4. Клас Notification

Проблема:

* Сповіщення користувачів про зміни в системі, такі як нові замовлення або зміну статусу замовлень, може бути складним без механізму сповіщення.

Рішення: Патерн: Observer (Спостерігач)

* Патерн Observer дозволяє одному об'єкту (видавцю) повідомляти інші об'єкти (спостерігачі) про зміни свого стану.
* Це дозволяє створити систему сповіщень, де Notification виступає видавцем, а інші компоненти системи підписуються на сповіщення про зміни.

Приклад застосування:

* Введення інтерфейсу Observer з методом update().
* Реалізація класів, які підписуються на сповіщення та реалізують метод update() для обробки сповіщень.
* Додавання методів для додавання, видалення та повідомлення спостерігачів в класі Notification.

5. Клас Profile

Проблема:

* Користувачі повинні мати можливість змінювати свої дані профілю, але іноді потрібно мати можливість відновити попередній стан профілю, наприклад, після помилкових змін.

Рішення: Патерн: Memento (Знімок)

* Патерн Memento дозволяє зберігати та відновлювати попередній стан об'єкта без порушення інкапсуляції.
* Це дозволяє зберігати стан профілю перед внесенням змін і відновлювати його у разі потреби.

Приклад застосування:

* Введення класу ProfileMemento, який зберігає стан об'єкта Profile.
* Метод в класі Profile для створення мементо та метод для відновлення стану з мементо.

6. Клас Cart

Проблема:

* Додавання, видалення та управління товарами в кошику може вимагати різних операцій та алгоритмів.

Рішення: Патерн: Strategy (Стратегія)

* Патерн Strategy дозволяє визначити сімейство алгоритмів, інкапсулювати їх та робити взаємозамінними.
* Це дозволяє динамічно змінювати алгоритми, які використовуються для управління товарами в кошику.

Приклад застосування:

* Введення інтерфейсу CartStrategy з методами для додавання, видалення та управління товарами.
* Реалізація різних стратегій для управління товарами в кошику (AddItemStrategy, RemoveItemStrategy).

7. Клас Part

Проблема:

* Частини можуть мати різні варіанти представлення та функціональності, які потрібно динамічно додавати або змінювати.

Рішення: Патерн: Decorator (Декоратор)

* Патерн Decorator дозволяє динамічно додавати нові поведінки до об'єктів, обертаючи їх у нові класи.

Приклад застосування:

* Введення базового інтерфейсу Part.
* Реалізація базового класу BasePart та декораторів, які додають нові функціональності до об'єкта Part.