# Лабораторна робота №4

з навчальної дисципліни: “Проектування програмних систем”

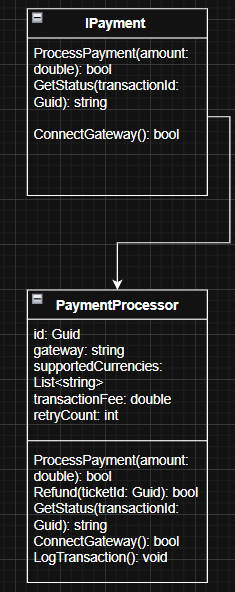
на тему:

“ Управління спортивними клубами, федераціями та змаганнями”

# 2025

У розглянутій діаграмі класів можна використати наступні архітектурні принципи:

**Dependency Inversion** - це принцип, суть якого полягає у тому, що модулі вищого рівня не повинні залежати від модулів нижчого рівня, а все повинно залежати від абстракції. Відповідно до цього принципу був побудований зв’язок IPayment -> PaymentProcrssor



**У розглянутій діаграмі класів можна використати наступні архітектурні принципи:**

**Dependency Inversion (Принцип інверсії залежностей)**

**Суть принципу полягає у тому, що модулі вищого рівня не повинні залежати від модулів нижчого рівня. Замість цього як високорівневі, так і низькорівневі модулі мають залежати від абстракцій. У нашій діаграмі цей принцип реалізовано через використання інтерфейсу IPayment, від якого залежить клас PaymentProcessor.**

**Стратегія (Strategy)**

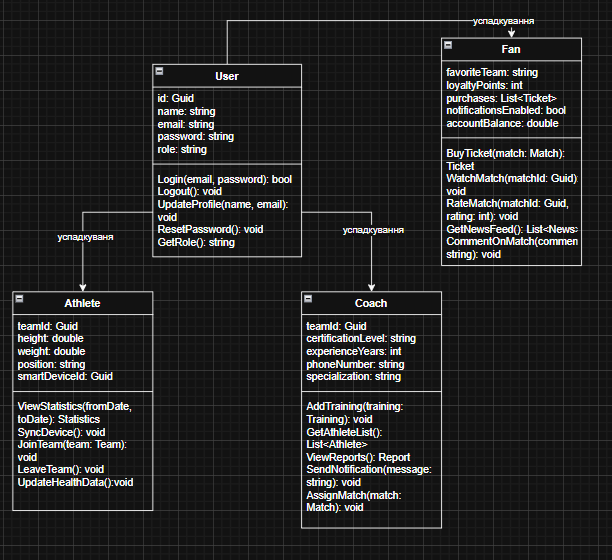
**Відповідно до цього принципу, було побудовано зв’язок IPayment → PaymentProcessor. Клас PaymentProcessor реалізовує всі методи, описані в інтерфейсі: ProcessPayment(), GetStatus() та ConnectGateway().**

**Стягнення плати в системі здійснюється через метод ProcessPayment(amount) інтерфейсу IPayment. У реальній реалізації це може бути PaymentProcessor, але, використовуючи патерн Стратегії, ми можемо реалізувати й інші класи (наприклад GooglePayProcessor, ApplePayProcessor, BankTransferProcessor), які також реалізовуватимуть IPayment.**

**Таким чином, логіка вищого рівня (наприклад, клас CheckoutService) не залежить напряму від конкретної реалізації способу оплати, а залежить лише від абстракції IPayment.**

**Dependency Injection (Впровадження залежностей)**

**Залежність від абстракції IPayment дозволяє нам легко змінювати реалізацію в залежності від конфігурації середовища або потреб користувача. Якщо ми хочемо змінити спосіб оплати на ApplePay — ми просто впроваджуємо нову реалізацію інтерфейсу IPayment, не змінюючи жодного рядка коду, який використовує інтерфейс.**

****

**Патерн: Шаблонний метод (Template Method)**

**Клас User є базовим (абстрактним або напівабстрактним) класом, від якого успадковуються класи Athlete, Coach та Fan. Це дозволяє реалізувати патерн Template Method: базовий клас надає спільні методи (Login, Logout, UpdateProfile, GetRole), а дочірні класи розширюють або перевизначають специфічну поведінку. Така реалізація дозволяє забезпечити:**

* **Єдиний інтерфейс авторизації та обробки профілю.**
* **Поліморфізм при роботі з різними типами користувачів.**
* **Гнучке розширення: нові типи користувачів (наприклад, Manager, Analyst) можуть додаватися без модифікації базового класу.**

**Патерн: Посередник (Mediator)**

**Комунікація між Coach, Athlete і Fan (наприклад, надсилання повідомлень, оновлення розкладу тренувань або отримання сповіщень про матч) може бути реалізована через посередника. Наприклад:**

**Coach не надсилає сповіщення напряму, а через сервіс: NotificationService.SendNotification(...)**

**Це дозволяє зменшити кількість прямих залежностей між об’єктами та інкапсулювати взаємодії в одному компоненті.**

**Принцип єдиної відповідальності (SRP)**

**Кожен клас має чітко визначену відповідальність:**

* **User — базовий обліковий запис;**
* **Athlete — дії спортсмена, пов’язані з командою та здоров’ям;**
* **Coach — організація тренувань і взаємодія з атлетами;**
* **Fan — споживання контенту та взаємодія з матчами.**

**Патерн: Спостерігач (Observer)**

**Методи SendNotification або CommentOnMatch можуть активувати інші реактивні дії, наприклад:**

**Відображення новин фанату у GetNewsFeed.**

**Надсилання результатів тренеру після завершення матчу.**

**Це типова ситуація для патерна Observer, де об'єкти сповіщають один одного про зміни стану.**

**Посилання на репозиторій та скріншот виконаної роботи**