# Лабораторна робота №4

з навчальної дисципліни: “Проектування програмних систем”

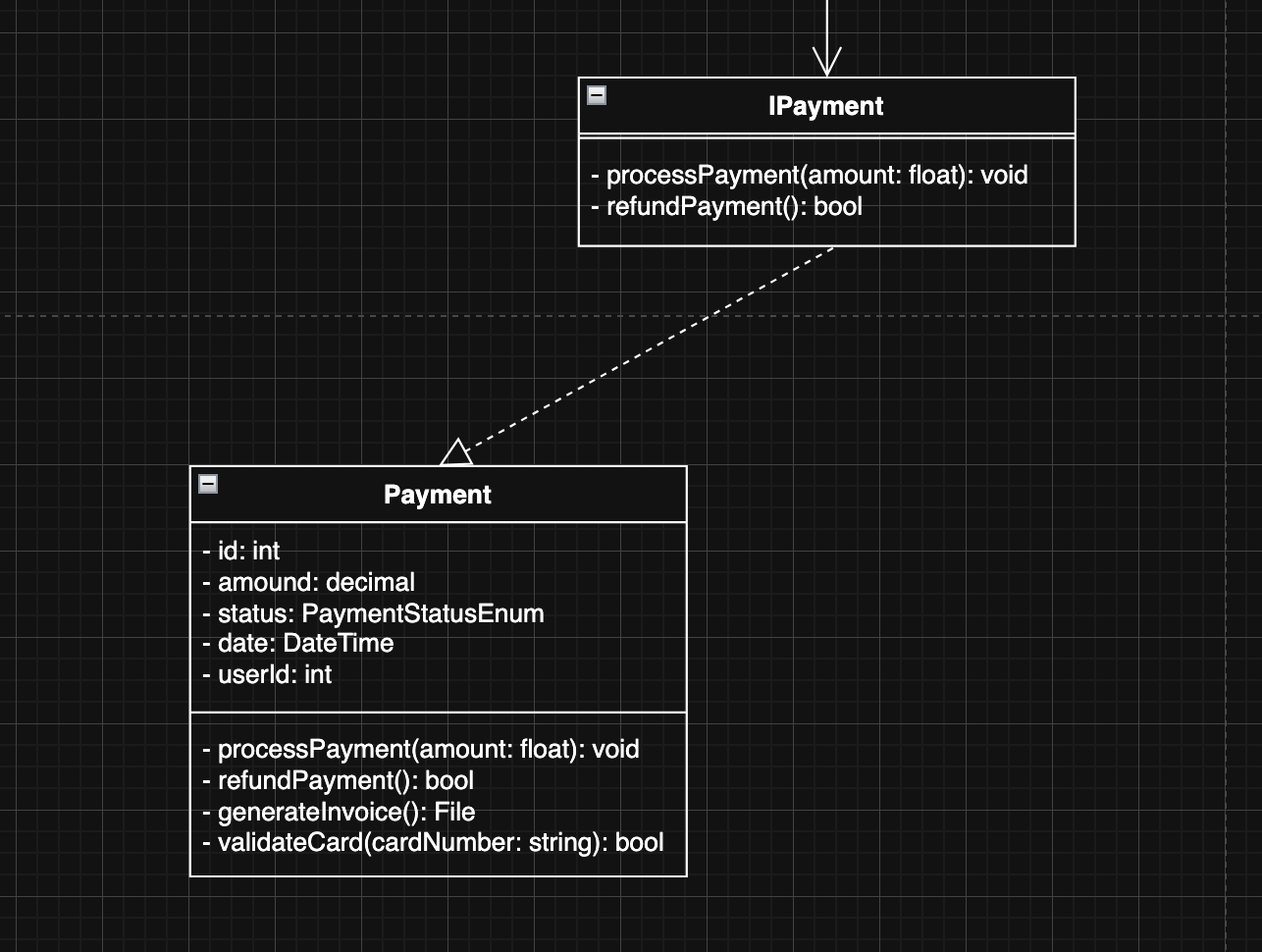
на тему:

“Курси іноземних мов”

# 2025

У розглянутій діаграмі класів можна використати наступні архітектурні принципи:

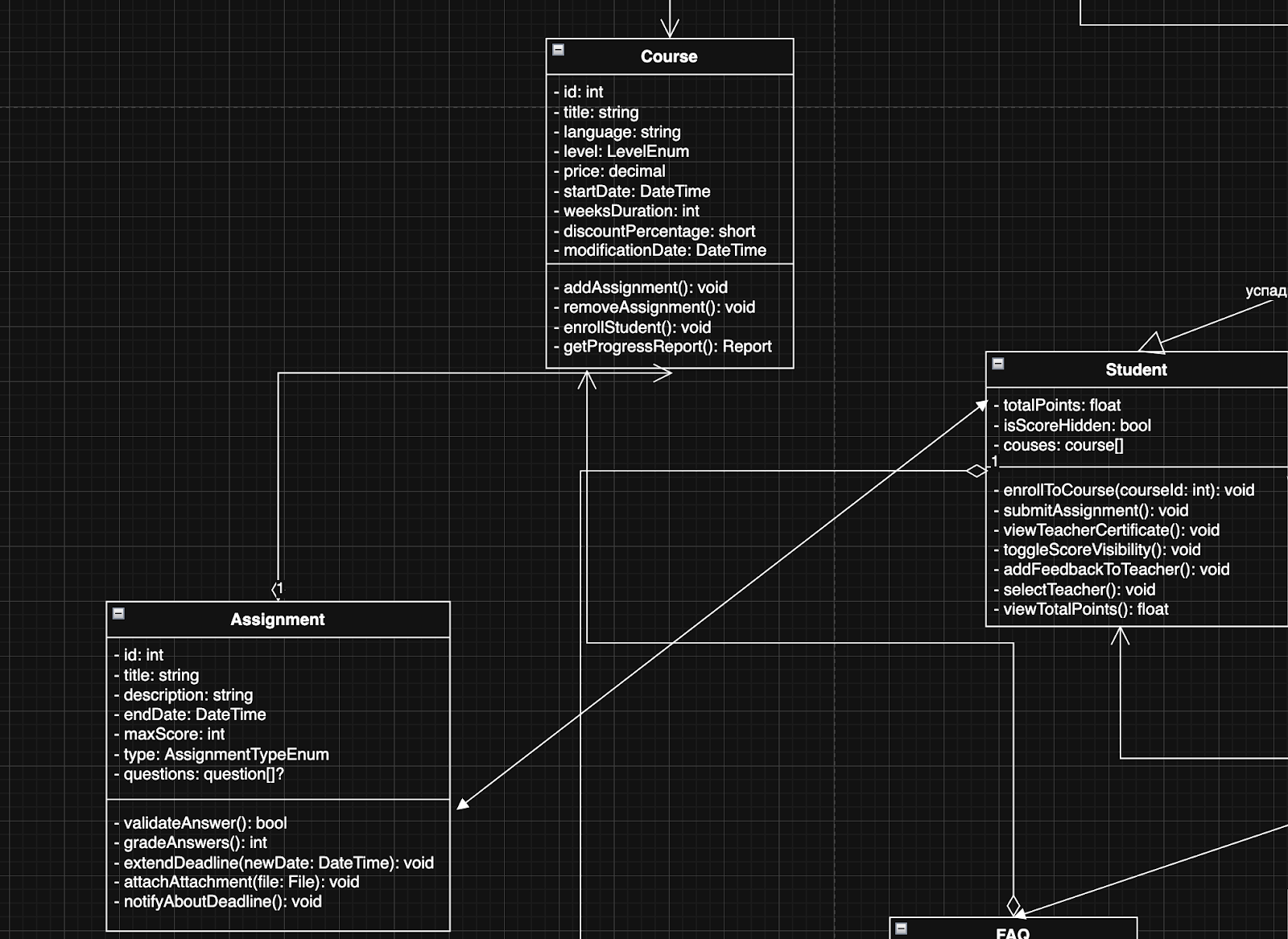
**Dependency Inversion** - це принцип, суть якого полягає у тому, що модулі вищого рівня не повинні залежати від модулів нижчого рівня, а все повинно залежати від абстракції. Відповідно до цього принципу був побудований зв’язок IPayment -> Payment



Стягнення плати відбувається через метод “processPayment()” інтерфейсу IPayment. Через патерн **Стратегії** вже відбувається використання саме реалізації Payment інтерфейсу IPayment. Проте, на сайті може бути декілька видів оплати, наприклад, через Google Pay та PayPal. Проте, використовуючи вказані підходи, ми можемо реалізувати швидке переключення між цими двома видами оплати, залежачи лишень від основного функціоналу IPayment.

Залежність від абстракції дозволяє нам також у подальшому швидко та майже без затратно змінити реалізацію. Наприклад, якщо наш додаток змінить свою цільову аудиторію на IOS-пристрої (або просто версія додатку буде збудована під IOS), то нам варто буде лишень створити клас ApplePayment, який буде реалізувати функціонал IPayment. При залежності виключно від функцій IPayment, це може бути швидко здійснено. Таку дію нам допоможе реалізувати патерн Dependency Injection. Він є подібним до Стратегії, але, як у першому випадку, Стратегія передбачає, що реалізація може змінюватися в ході програми, у той час як при **Dependency Injection** більш прийнято оголосити один тип залежності протягом усього циклу життя програми.

Розглянемо  наступні зв’язки класів:



Завдання, призначені у курсі, можуть мати декілька типів: слухання, читання, письмо. Всі вони можуть вільні відповіді або тести. Для всіх видів завдань можна оголосити спільний абстрактний клас: Assignment. Від нього можуть походити ReadingAssignment, WritingAssignment і т.д. Відповідно, у нас має бути можливість зручно створювати завдання будь-якого типу. При такій реалізації, у додатку може бути використаний поліморфізм, та за виникнення таких умов нам може знадобитися патерн **Фабрики**, який дозволить зручно та швидко створювати об’єкти будь-яких спільних класів.

Вчитель повинен бути проінформованим щоразу, коли учень здає завдання. Після цього вчитель має можливість оцінити завдання та, за бажанням, надати відгук. Про будь-яку з цих дій учень теж має бути проінформованим. Для таких дій, коли певна дія має відбуватися за умови трігеру іншої дії, можна використати патерн **Спостерігач**. Цей патерн також може стати в нагоді при розробці користувацького інтерфейсу, оскільки стан фронтенду має реактивно реагувати на будь-які дії користувача.

У декількох місцях у програмі також може бути використаний патерн **Замісника**. Наприклад, створивши замінник для сертифікатів викладача, замість постійного тривалого завантаження файлу, ми можемо його закешувати та повертати з пам’яті. При наявності відеоматеріалів, вони теж можуть бути використані через Замісника, хоча за цим треба слідкувати уважніше, оскільки такий підхід може зайняти багато пам’яті.

Замісник також може бути використаним для лінивої завантаження з бази даних. Це корисно, коли, після формування запиту до бази даних на нашому сервері, замість негайного завантаження записів у пам’ять, ми виконуємо цю дію лишень тоді, коли вона дійсно необхідна, бо врешті цей запит може стати просто зайвим. Такий вид замісника також може знадобитися на клієнтській стороні, допомагаючи не навантажувати браузер клієнта сторінками, які він не запросив і йому ще не потрібні (або взагалі не будуть потрібними).

У деяких мовах, для побудови подібних запитів, може бути використаний патерн **Будівельника**, коли замість формування одного єдиного об’єкта за раз, ми поступово відтворюємо його. Можливо, саме тут цей патерн може бути не влучним, проте він точно може знадобитися при створенні курсу. Оскільки об’єкт курсу може бути величезним, то зручніше було би будувати його поступово.

Коли ж відбуваються будь-які інші дії відносно цього курсу (операції на зміни, додавання, або видалення), можна використати патерн **Команди**. Він буде дозволяти перетворювати запити на об’єкти як аргументи для методів, і це буде зручно для логування дій з ними. Оскільки курси - це центральна сутність у нашій діаграмі, то логування відносно них важливе. Так само будь-які дії з ними можуть бути доволі тривалими та затратними, і правильна реалізація патерну Команди дозволить створити скасування операції.

Курси також можна “обгорнути” у патерн **Декоратора**. Декоратор дозволяє додати додатковий функціонал до об’єктів, не впливаючи на його поточну реалізацію. У випадку курсів це зручно для визначення тимчасово недоступних (або прихованих курсів), або додавши на курс знижку.

Патерн **Одинак** передбачає, що на весь час існування програми такий об’єкт є єдиним у своєму роді. Такий вид операції є доволі небезпечним, проте важливим для підтримки зовнішніх з’єднань. Наприклад, він може бути використаним для підтримку зв’язку з базою даних (якщо з’єднань буде декілька, то кожне з них буде єдиним у своєму роді).

Для загальної комунікації в проекті може бути використаним патерн **Посередника**. Правильно його реалізувавши, або підібравши для цього правильну бібліотеку, можна інкапсулювати комунікацію між компонентами проєкту.

**Посилання на репозиторій та скріншот виконаної роботи**

<https://github.com/ArgasDTiger/Software-System-Design>

