SW Deliverable 3

Structural Design Patterns

## Overview

This report focuses on implementing the structural design patterns into our EduCertify system. It's a learning and certification system that enables learners worldwide to be able to enroll in courses and earn certification upon the completion of the courses. This report will tackle different structural design patterns in which each design pattern will communicate a specific problem and will be able to solve it in a maintainable and extensible manner.

## Bridge Design Patterns

### 1. Certificate Export System

**Intent**:

- Decouple different certificate types (Completion or Excellence) from the export format (PDF, HTML, Image) so that both can evolve independently in different hierarchies.

**Problem**:

- Each certificate type may need to be exported in multiple formats
- Without bridge, all combinations will grow exponentially which will cause the code to be not maintainable or extensible.

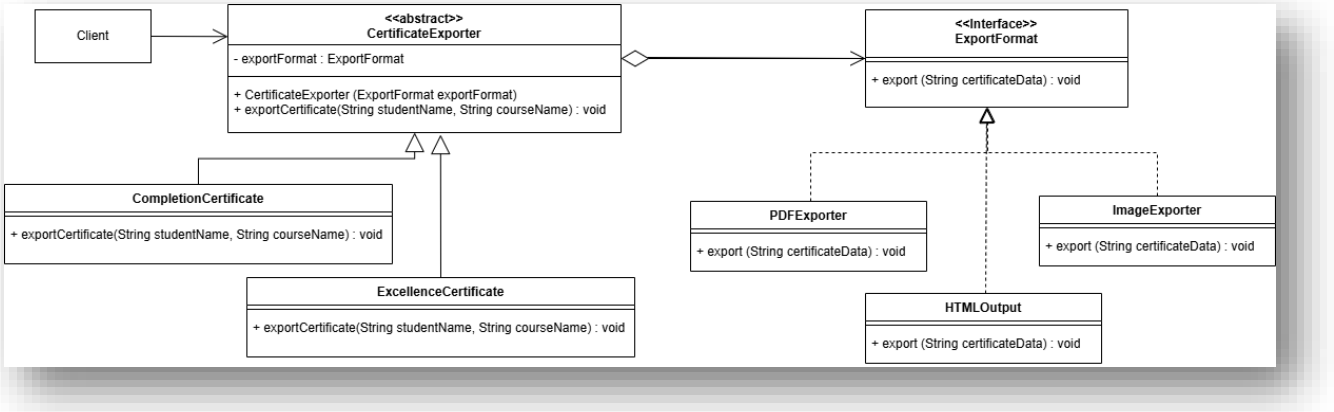
**Solution**:

- Use bridge design pattern to separate the two independent dimensions
    - Certificate Type (Abstraction)
    - Export Format (Implementation)

**Components**:

- Abstraction: Certificate Type
- Refined Abstractions: CompletionCertificate, and ExcellenceCertificate
- Implementor: Export Format
- Concrete Implementor: PDF, HTML, Image

**UML**:




**Consequences**:

- Achieves decoupling
- Extensible: adding new formats and certificates without changing the old code

## 2. Course Content Display System

**Intent**:

- Decouple how course content is displayed (UI layout) from what content is being shown (video, text, quiz) modules.

**Problem**:

- Instructors want to choose content types and display formats flexibly
- Hard-coding leads to rigid classes and inflexibility.
- Every new combination will lead to a new class which will cause an explosion to the codebase.

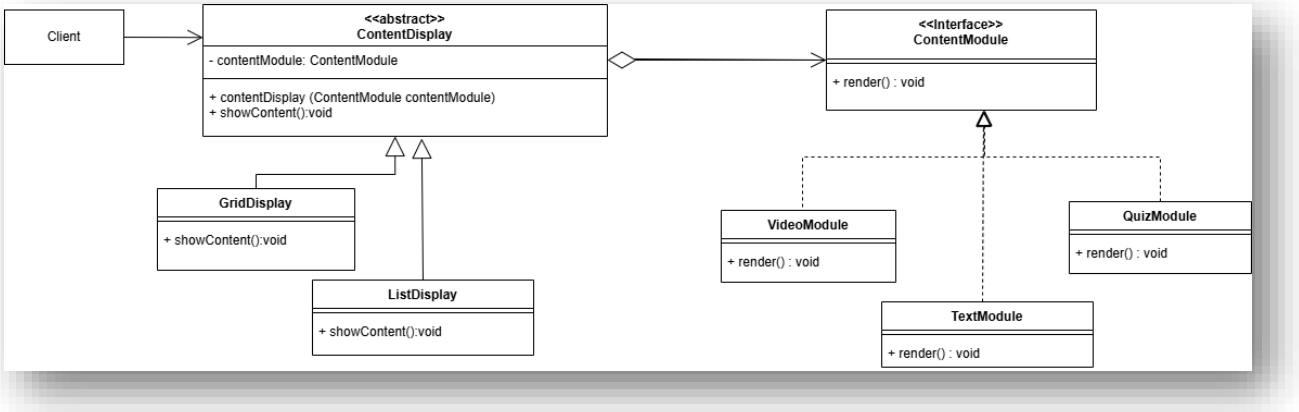
**Solution**:

- Use bridge pattern
- Abstraction: Content Display (how content is shown)
- Implementor: Content Module (what content is shown)

**Components**:

- Abstraction: Content Display
- Refined Abstractions: GridDisplay, ListDisplay
- Implementor: Content Module
- Concrete Implementor: VideoModule, TextModule, QuizModule

**UML**:




## Adapter Design Patterns:

## 1. Third – Party Authentication

**Intent**:

- Allow EduCertify login system to accept authentication from external providers like Google or Facebook without modifying its own in-house interface.

**Problem**:

- External services like Google and Facebook
- Use different method names
- Return different token structures

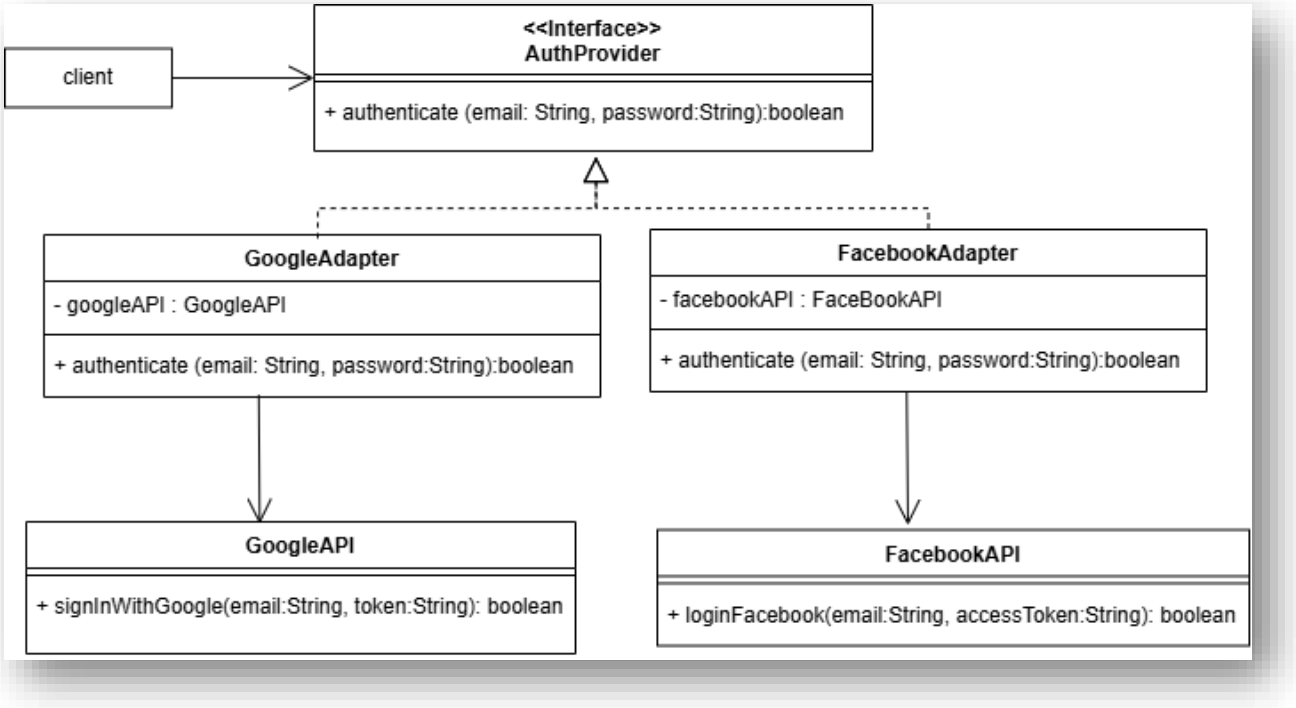
**Solution**:

- Use Adapter design pattern
- Create an AuthProvider Interface that our system uses and understands.
- Write adapters that translate the external services into our in-house interface.

**Components**

- Target Interface: AuthProvider (EduCertify)
- Adapters: GoogleAuthAdapter, FacebookAuthAdapter
- Adaptees: GoogleAPI, FacebookAPI

**UML**:




## 2. Video Hosting Integration

**Intent**:

- Allow EduCertify' s video player to support videos hosted on YouTube and Vimeo using a single consistent interface.

**Problem**:

- YouTube and Vimeo use different APIs.
- EduCertify wants a unified play() method across all sources.

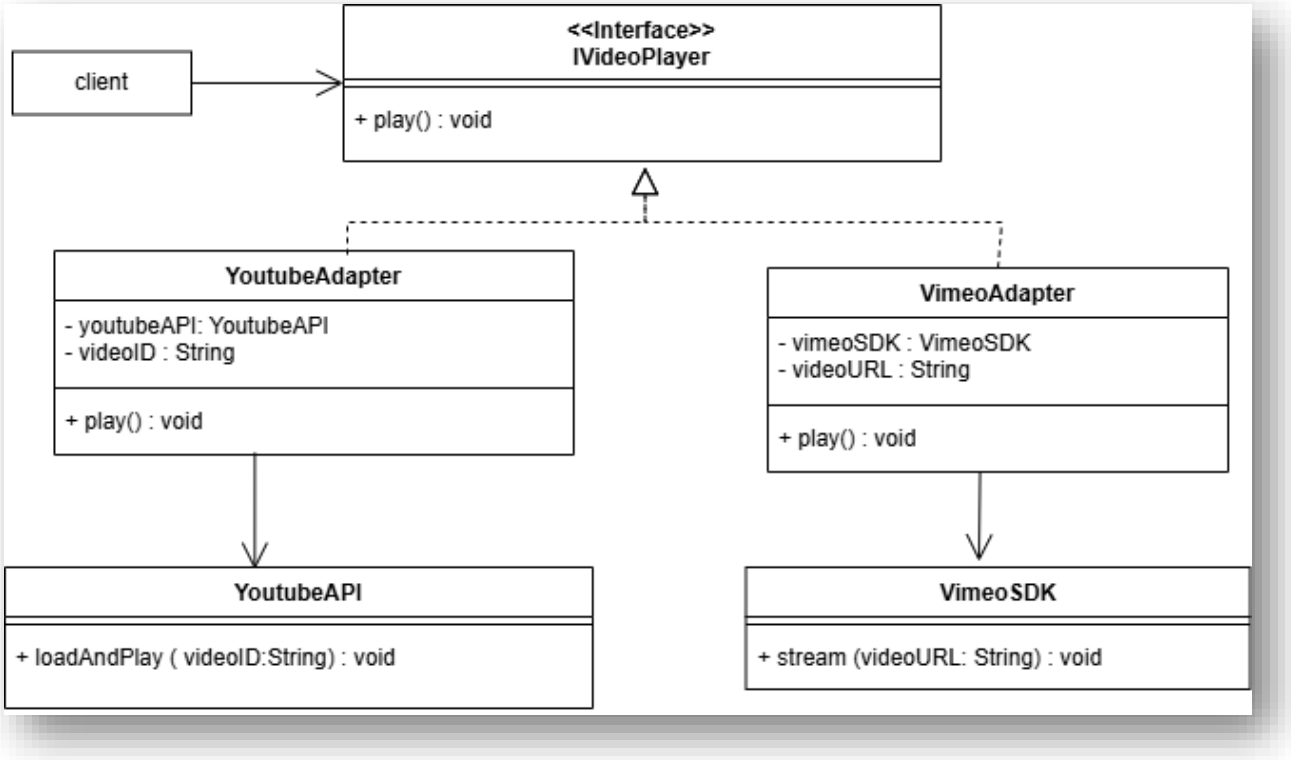
**Solution**

- Use the Adapter Pattern to wrap YouTube/Vimeo SDKs inside adapter classes that conform to a common interface: IVideoPlayer.

**Components**:

- Target Interface: IVideoPlayer
- Adapters: YoutubeAdapter, VimeoAdapter
- Adaptees: YoutubeAPI, VimeoSDK

**UML**:


<<Interface>>
IVideoPlayer
+ play() : void
client
YoutubeAdapter
- youtubeAPI: YoutubeAPI
- videoID : String
+ play() : void
VimeoAdapter
- vimeoSDK : VimeoSDK
- videoURL : String
+ play() : void
YoutubeAPI
+ loadAndPlay ( videoID:String) : void
VimeoSDK
+ stream (videoURL: String) : void

**Flyweight Design Pattern:**

**Course Category Tags:**

**Intent**

- Efficiently reuse common data objects like category tags ( AI, Data Science, Cybersecurity) across many courses without duplicating memory.

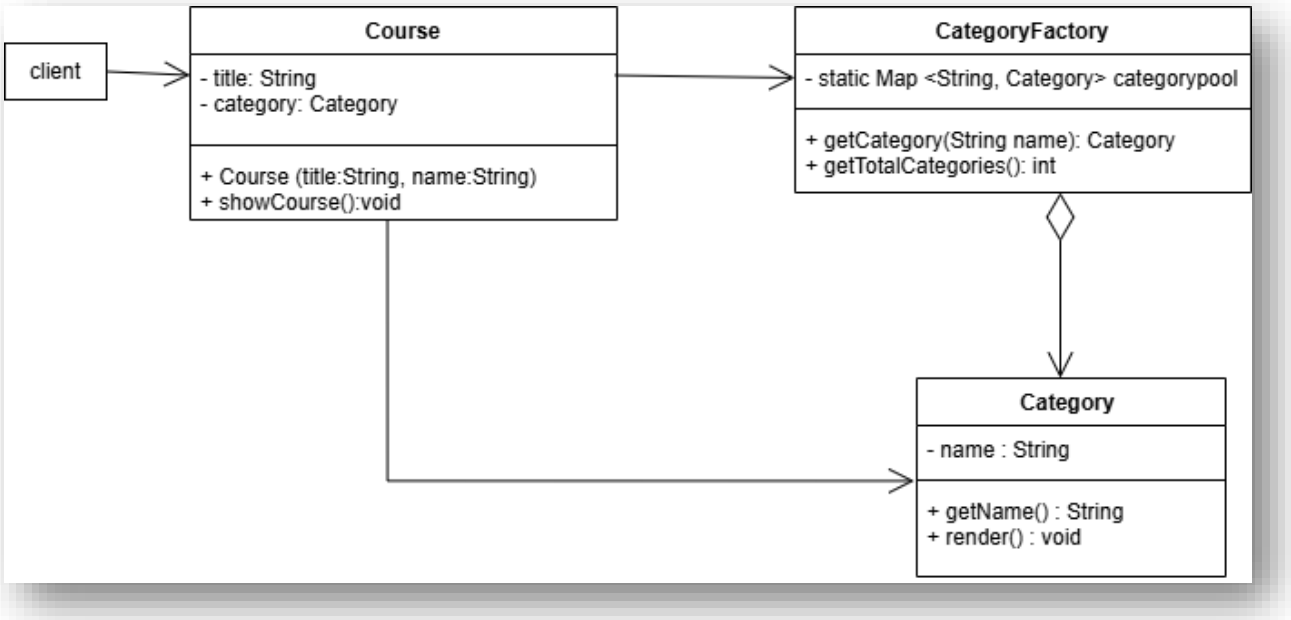
**Problem**

- Thousands of courses use the same categories.
- Without Flyweight, every course would store its own "AI" or "Data Science" string object which will waste memory and break UI consistency.

**Solution**

- Use a Flyweight Factory (TagFactory) to return shared Tag instances.
- If a tag already exists, reuse it.
- If not, create it once and store it.

**UML**:



## Decorator Design Pattern

## 1. Notification Enhancer

**Intent:**

- Adding a new notification channel like (SMS, Push Notification, WhatsApp, etc.) dynamically to a base notifier like (Email).

**Problem**:

- Flexible combinations of notifications
- Email only
- Email and Push and WhatsApp
- Subclassing each combination is impractical and causes the code to increase exponentially.

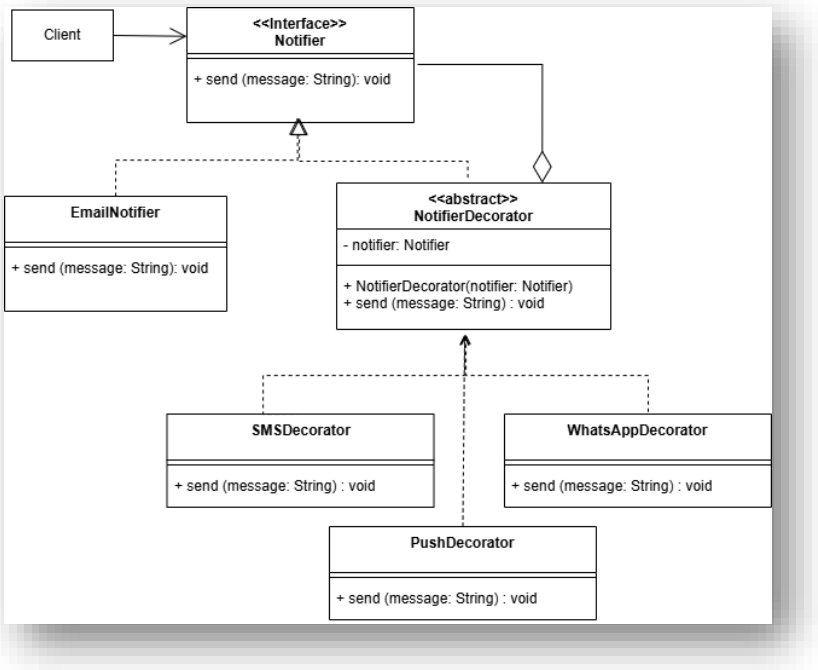
**Solution**:

- Decorator pattern to wrap base Notifier with additional behavior.

**Components**:

- Component Interface: Notifier
- Concrete Component: Email Notifier
- Decorators: SMS, Push, WhatsApp

**UML**:



## 2. Gamified Progress Bar

**Intent**

- Dynamically enhance progress bar with gamified features such as emojis, badges, and sound effects.
- No hardcoded progress bar subclasses (for each combination).

**Problem**:

- The creation of subclass for each combination leads to unmanageable code.

**Solution**:

- Decorators to add features with the base feature

**Components**:

- Component Interface: Progress Bar
- Concrete component: Basic Progress Bar
- Decorators: Emoji, Badge, SoundEffect Decorators.

**UML**: