مقرر تصميم نظم البرمجيات | كلية الهندسة المعلوماتية | المحاضرة السادسة

Design Patterns:
Singleton Façade Adapter Proxy

د. رياض سنبل

Design Patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - Factory Method
  - Abstract Factory
  - Singleton
  - Builder
  - Prototype
- **Structural Patterns** *(how objects/classes can be combined)*
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- **Behavioral Patterns** *(communication between objects)*
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Observer
  - State
  - Strategy
  - Chain of Responsibility
  - Visitor
  - Template Method

Singleton Pattern

Singleton Pattern

- The Singleton pattern is a design pattern that ensures a class has only one instance and provides a global point of access to that instance.
- It's commonly used when there's a need for a single, shared instance of a class throughout the application.

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // private constructor to prevent instantiation
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    // Other methods and properties
}
```

Using Singleton in Data Access Layer

- In the context of a Data Access Layer, the Singleton pattern might be used to ensure that there's only one instance of a database connection or a repository manager throughout the application.
- This can be beneficial for resource management and to avoid unnecessary overhead in creating multiple connections.

```
public class DataAccessLayer {
    private static DataAccessLayer instance;

    // private constructor to prevent instantiation
    private DataAccessLayer() {
        // Initialization logic for data access layer
    }

    public static DataAccessLayer getInstance() {
        if (instance == null) {
            instance = new DataAccessLayer();
        }
        return instance;
    }

    // Other data access methods
}
```

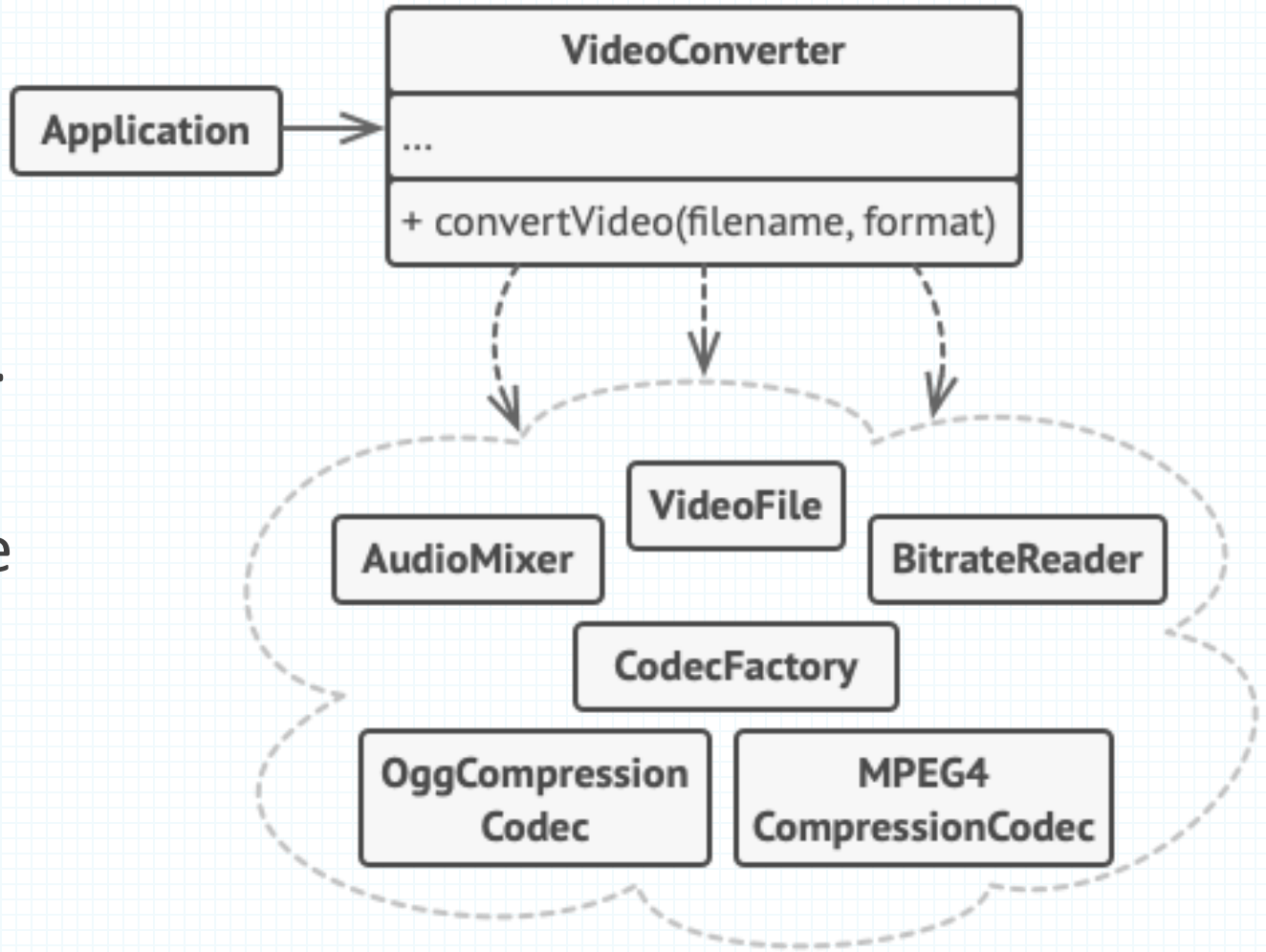
Design Patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - Factory Method
  - Abstract Factory
  - Singleton
  - Builder
  - Prototype
- **Structural Patterns** *(how objects/classes can be combined)*
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- **Behavioral Patterns** *(communication between objects)*
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Observer
  - State
  - Strategy
  - Chain of Responsibility
  - Visitor
  - Template Method

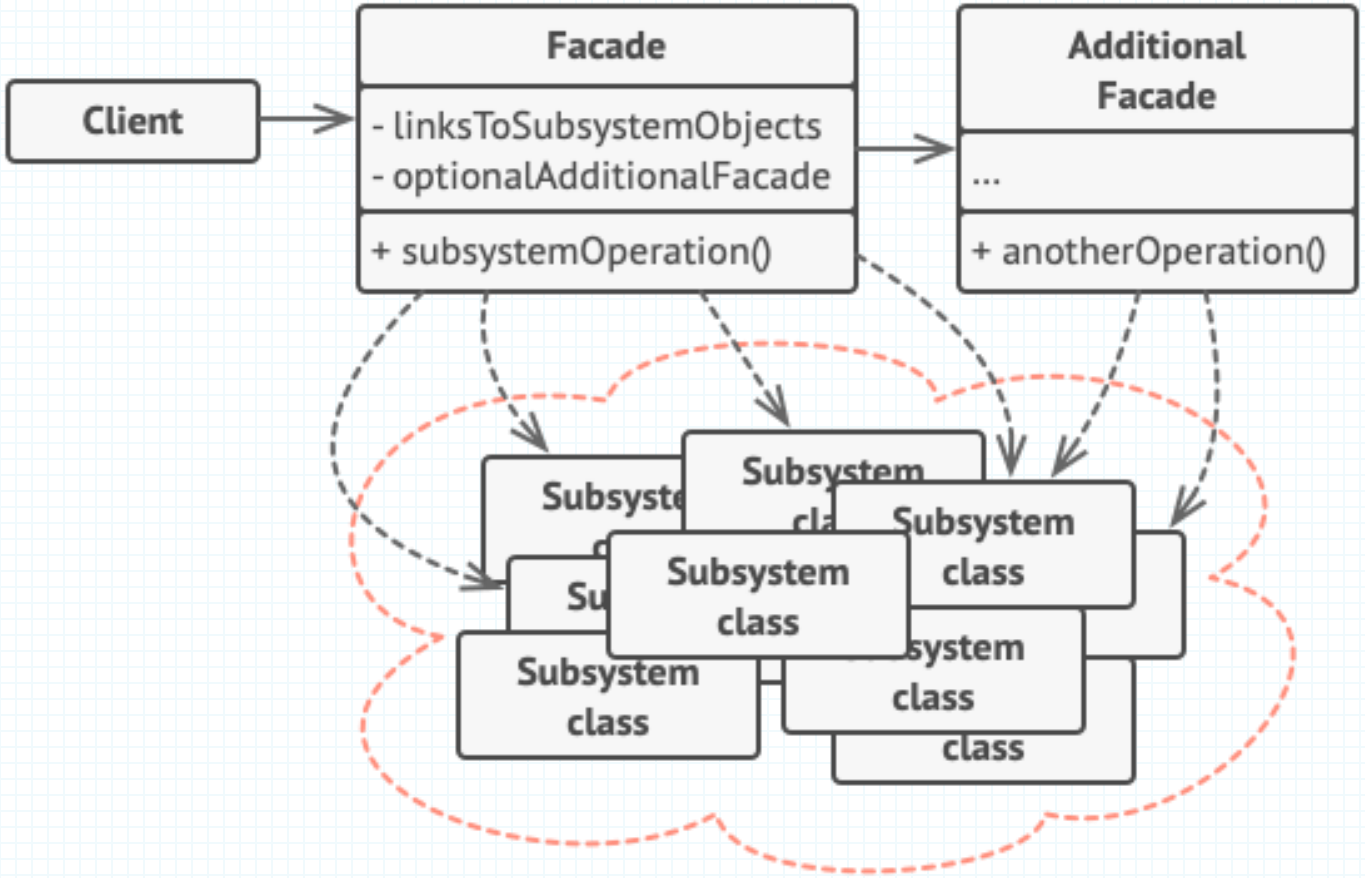
The Facade Pattern

The Facade Pattern

- Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.
- isolating multiple dependencies within a single facade class.



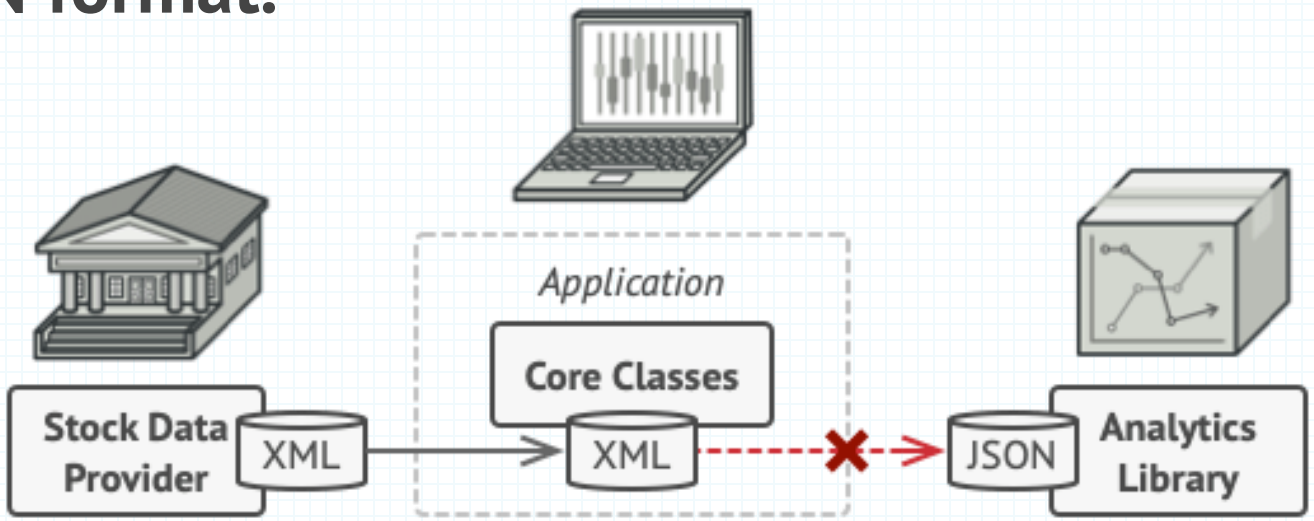
Structure


Client
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
Additional Facade
...
+ anotherOperation()
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class

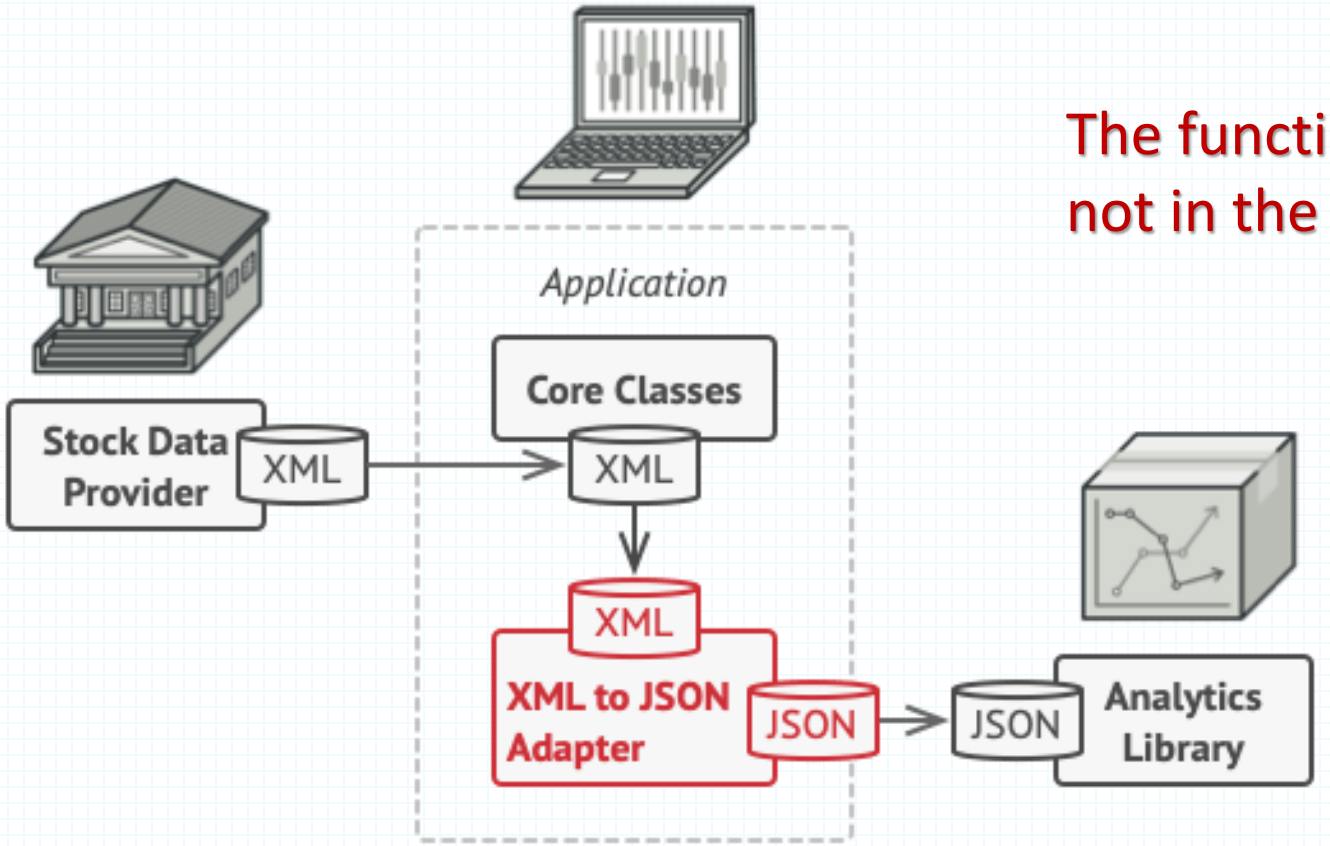
The Adapter Pattern

The problem

- You're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- You decide to improve the app by integrating a smart 3rd-party analytics library. But **the analytics library only works with data in JSON format.**



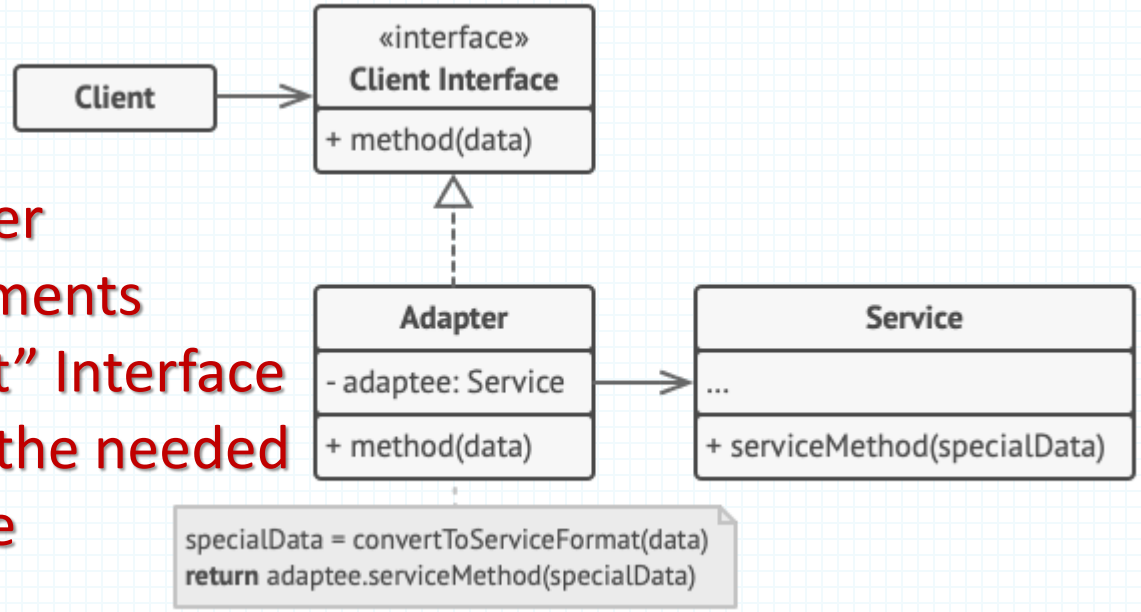
Solution



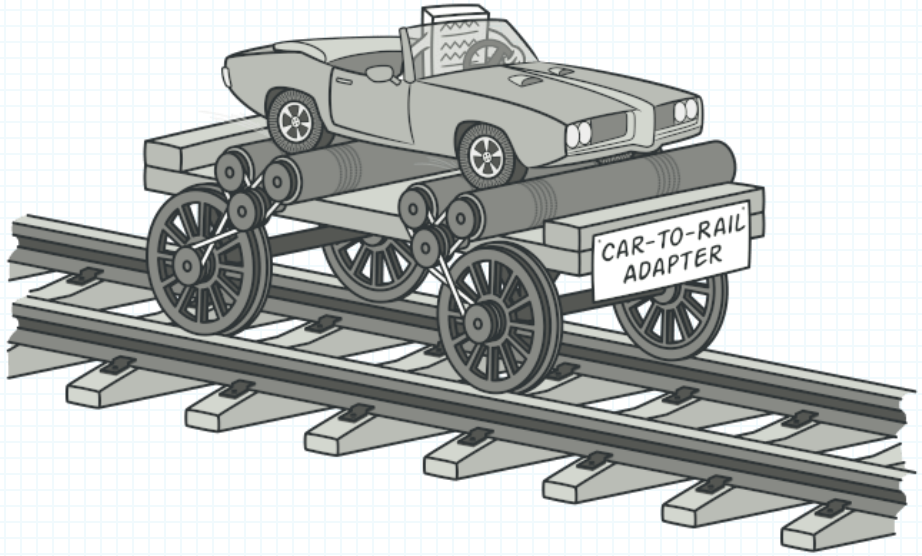
The functionality we need, but not in the way we want to use it.

The Adapter Pattern

- Adapter is a structural design pattern that allows **objects with incompatible interfaces to collaborate**.



Adapter implements “Client” Interface
Using the needed service



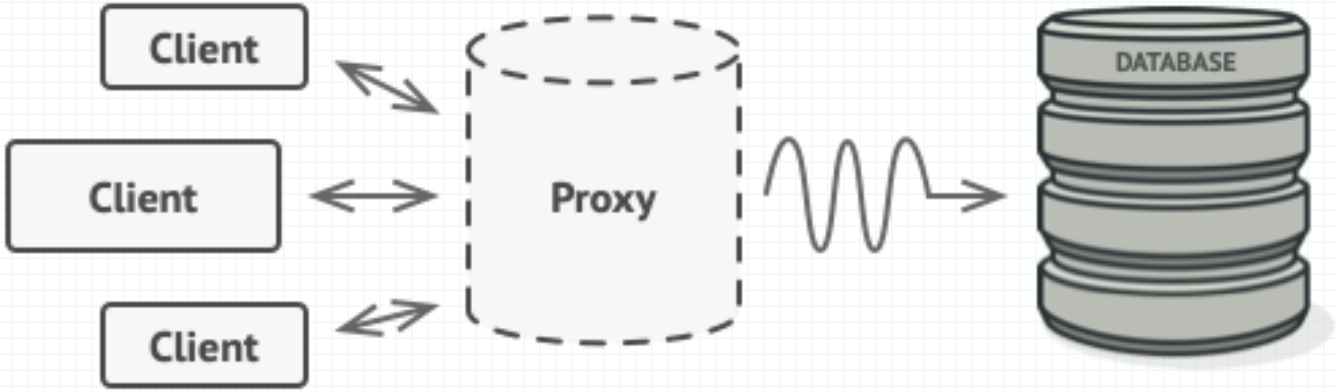
Design Patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - Factory Method
  - Abstract Factory
  - Singleton
  - Builder
  - Prototype
- **Structural Patterns** *(how objects/classes can be combined)*
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- **Behavioral Patterns** *(communication between objects)*
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Observer
  - State
  - Strategy
  - Chain of Responsibility
  - Visitor
  - Template Method

The Proxy Pattern

The Proxy Pattern

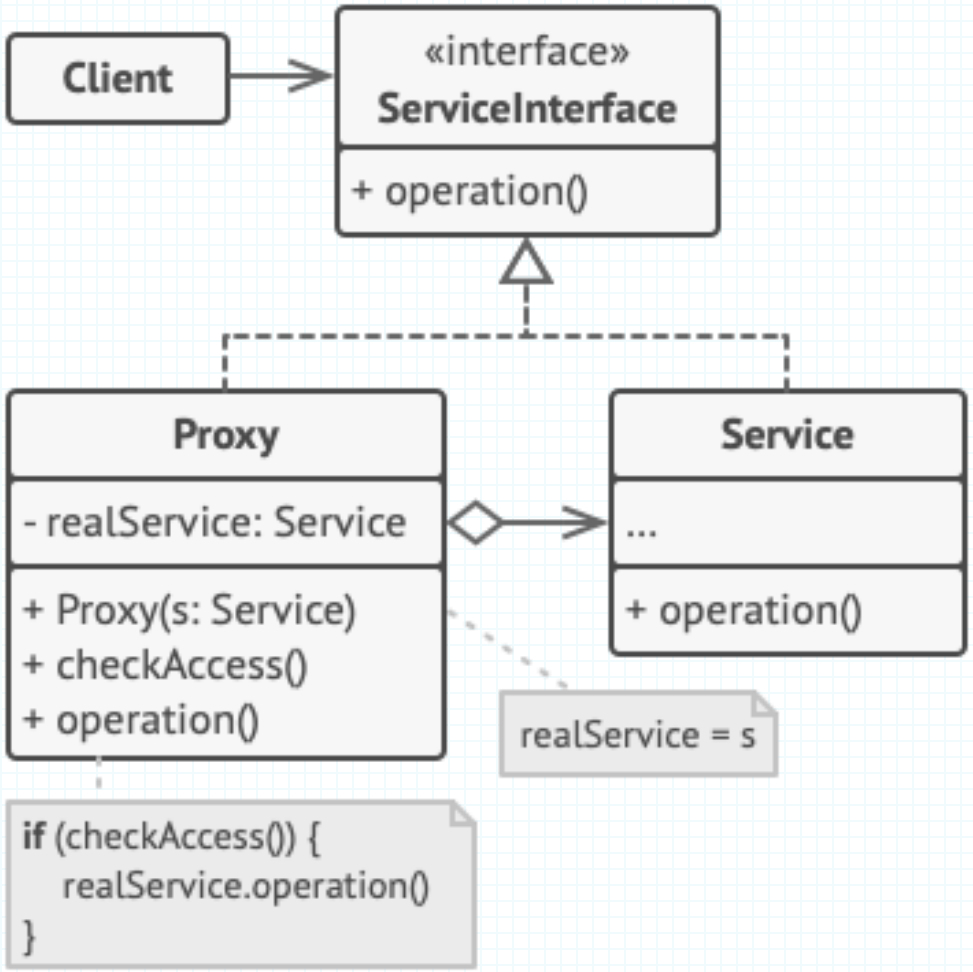
- Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object.



- WHY?

If you need to execute something either before or after the primary logic of the class, the proxy lets you do this without changing that class.

Structure



Note that BOTH (the service and our proxy implement the same interface!)
WHY?

it can be passed to any client that expects a real service object

When to use?

- Lazy initialization (virtual proxy): This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.
- Access control (protection proxy).
- Local execution of a remote service (remote proxy).
- Logging requests (logging proxy).
- Caching request results (caching proxy).