Professional Skills
Agile Fundamentals
Jira
Git
Databases Introduction
Java Beginner
Maven
Testing (Foundation)

Java Intermediate

- Optionals
- JDBC CRUD
- Exceptions
- SOLID Principles
- Single Responsibility
- Open/Closed
- Liskov Substituiton
- Interface Segregation
- Dependency Inversion
- Best Practice
- Design Patterns
- Creational Design Patterns
- Structural Design Patterns
- Behavioural Design Patterns
- Collection & Map
- HashSets
- HashMaps
- Enums
- Logging
- Generics
- Lambda Expressions
- Streams
- Complexity
- Input and Output
- Local Type Inference

HTML

Structural Design Patterns

Contents

- Overview
- Tutorial
  - Adapter Pattern
    - Adapter Pattern Advantages
    - Adapter Pattern Disadvantages
  - Façade Pattern
    - Façade Pattern Advantages
    - Façade Pattern Disadvantages
- Exercises

Overview

Structural design patterns are designed to structure your classes in objects in a meaningful way, usually through inheritance and interfaces.

The structural patterns are:

- Flyweight
- Adapter
- Composite
- Decorator
- Façade
- Proxy
- Bridge
- Filter

Tutorial

Adapter Pattern

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

This is a special object that converts the interface of one object so that another object can understand it.

Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate:

- The adapter gets an interface, compatible with one of the existing objects.
- Using this interface, the existing object can safely call the adapter's methods.
- Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Sometimes it is even possible to create a two-way adapter that can convert the calls in both directions.

For instance, let's say we have two classes, `RoundHole` and `RoundPeg`, with compatible interfaces:

▶ Round Hole
▶ Round Peg

We also have an incompatible class called `SquarePeg`:

▶ Square Peg

CSS
Javascript
Spring Boot
Selenium
Sonarqube
Advanced Testing (Theory)
Cucumber
MongoDB
Express
NodeJS
React
Express-Testing
Networking
Security
Cloud Fundamentals
AWS Foundations
AWS Intermediate
Linux
DevOps
Jenkins Introduction
Jenkins Pipeline
Markdown
IDE Cheatsheet

We can then introduce an Adapter class to allow us to fit the square pegs into the round hole.

We extend `RoundPeg` class to let the Adapter objects act as round pegs:

▶ Adapter Pattern

Then, in the main class, we can now use the Adapter:

▶ Using Adapter

The output we get is:

```
Round peg r5 fits round hole r5.
Square peg w2 fits round hole r5.
Square peg w20 does not fit into round hole r5.
```

We use the Adapter pattern when we want to use some existing class, but its interface isn't compatible with the rest of your code.

We can use the Adapter pattern when we want to reuse several existing subclasses that lack some common functionality and can't be added to the superclass.

Adapter Pattern Advantages

- Adheres to the Single Responsibility Principle - you can separate the interface or data conversion code from the primary business logic of the program.
- Adheres to the Open/Closed Principle - you can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

Adapter Pattern Disadvantages

- The overall complexity of the code increases, because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

Façade Pattern

The Façade pattern is a structural design pattern that provides a simplified interface to a library, a framework or other complex set of classes.

A Façade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts.

A Façade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients care about.

Having a Façade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

Below is an example of how the Façade pattern is used. We have a video converter project which contains many classes which return some statements:

▶ Façade Pattern

With the use of the Façade pattern, we can bring together functionality from other complex classes to achieve a simple goal.

- Use the Façade pattern when you need to have a limited but straightforward interface to a complex subsystem.
- Use the Façade when you want to structure a subsystem into layers.

Façade Pattern Advantages

- You can isolate your code from the complexity of a subsystem.

Façade Pattern Disadvantages

- A Façade can become a god-object, which is coupled to all classes of an app.

Exercises

Here, we have a `Mallard` and a `WildTurkey`:

```
public class Mallard implements Duck {

    @Override
    public void quack() {
        System.out.println("wak");
    }

    @Override
    public void fly() {
        System.out.println("i am doing a fly, i am");
    }
}
```

```
public class WildTurkey implements Turkey {

    @Override
    public void gobble(){
        System.out.println("gobblé");
    }

    @Override
    public void fly(){
        System.out.println("i am doing a sm0l fly, because i'm bige");
    }
}
```

1. Reverse-engineer the `Duck` and `Turkey` interfaces using what you already know.
2. Write a `TurkeyAdapter` which allows us to use `Turkey` objects in place of `Duck` objects.
3. Write a `DuckAdapter` which allows us to use `Duck` objects in place of `Turkey` objects.
4. Test that both your Adapters work with a `Runner` class containing a `main()` method.

▶ Solution for (1)
▶ Solution for (2)
▶ Solution for (3)
▶ Solution for (4)