THE UNIVERSITY OF SYDNEY

SOFT2201/COMP9201 Tutorial 9

Adapter & Observer

Adapter Pattern

Adapter is the first pattern where using it in an initial design is almost guaranteed to be a bad idea - it is an **extension** pattern, used to allow code to be extended in a maintainable way without expending a large amount of refactoring effort. To understand why this is a useful thing you must first understand two fundamental concepts of programming in real world projects - every time someone touches code, it costs time & money, but also they have a chance to introduce bugs into that code. Because of these reasons once code is working there are very good reasons to not want to touch it (there are reasons to want to touch it too, refactoring is often a good thing - but sometimes you want to avoid it).

With Adapter we allow an existing service used by a client to be swapped out for another already existing service that has a different API. We leverage dependency inversion *and* dependency injection in order to do this - if the client code depends on an interface, it is very easy to swap in another implementation of that interface. If the client is *given* that implementation rather than creating it itself then it doesn't even need to know anything has changed.

Question 1: Implementing Adapter

Consider the following client, existing old service, existing new service, and main driving class. The lecture covered the 2 different ways to implement Adapter - both of these methods will allow you to have the client code use the new service where the only existing class that gets even a **single character** modified is the main driving class. Determine the best choice for this problem, and implement this. Consider the difference between the version of adapter you have selected and the alternative. Also consider how you would have to do it without Adapter. The following Java code can be found in a zip folder on canvas module 9.

```
public interface Service{
        int getValue();
        void printDescriptionToSTDOut();
        void toggle();
}

public class Client {
        private Service service;
        public Client(Service service) {
                this.service = service;
        }
        public void useService() {
                service.toggle();
                service.printDescriptionToSTDOut();
                System.out.println(service.getValue());
        }
}

public class OldService implements Service {
        private boolean toggled = false;

        public OldService(){}

        public int getValue(){
                return 42;
        }

        public void printDescriptionToSTDOut() {
                System.out.println("Toggle is currently: " + toggled);
        }

        public void toggle(){
                toggled = !toggled;
        }
}
```

```java
// Note: this deliberately does NOT implement Service
public class NewService {
        public static final int VALUE = 42;

        private boolean toggled = false;

        public NewService(){}

        public String getDesc() {
                return "Hey, this is " + toggled;
        }

        public void setToggled(boolean toggled) {
                this.toggled = toggled;
        }

        public boolean getToggled() {
                return toggled;
        }
}

public class MainDriver {
        public static void main(String[] args) {
                Client client = new Client(new OldService());
                client.useService();
        }
}
```

Observer

Observer is quite different from the structural pattern Adapter because it is something to include 'in advance', rather than as a modification to an existing system like Adapter - though it can be involved in extensions or refactoring as well. Observer handles issues of deep and complex coupling, not by removing the coupling (the line on a UML diagram will still be there) - but by lessening how involved that coupling is (the technical term for this is improving the 'connascence' between the classes).

Question 2: Implementing Observer in a New System

Consider the following requirements of an autonomous vehicle fleet monitoring system. The monitoring system is based on Vehicle objects which track current vehicle speed, fuel, and occupancy. The application should provide a FleetDisplay with each vehicle shown as a separate line or tile, all updated in real time as the Vehicle objects update their associated vehicle's status via the internet (your implementation doesn't need to actually use the internet, you can just get random integers at random intervals from 5 to 10 seconds in length (think 300 to 600 ticks if you use a 60 fps keyframe, don't try anything involving Thread.sleep or similar).

You will notice far less of this system has been provided to you (aka none of it) - use this as practice for including a design pattern in an entirely new system where there is no provided code to start you off.

Because this is quite an involved activity you should ensure you first model it in UML - aside from this being a good idea every time you model a system, that way your tutor has a chance to review your design within the tutorial time. When implementing it you should start with a version that does nothing but print updates to System.out - this should be a model package with a console view package. Once you have this working with the Observer pattern, then swap out the console view package with a JavaFX view.