CS355 Software Engineering Process

Lab Week 03

Design Pattern – Adapter Pattern

In this lab, we are going to look at a software design pattern: adapter pattern. In general, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. Software design patterns typically show relationships and interactions between classes and objects.

The Adapter Pattern

The *adapter pattern* converts the interface of a class into another interface the clients expects. This pattern mainly adapts one object to another one. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. The adapter pattern is widely known in software development and used in many programming languages, e.g., Java.

Adapter allows to reuse existing coding without changing it, as the adapter ensures the conversion between the different interfaces. The adapter pattern only converts interfaces, while the other patterns such as decorator pattern adds new functionality to an existing interface.

Example of an Adapter

Let's implement a real world adapter like a power adapter. Different countries sometimes have different electrical sockets. In order to make different electrical sockets work together with different plugs the use of adapters is necessary.

German socket:

```
public class GermanElectricalSocket {
    public void plugIn(GermanPlugConnector plug) {
        plug.giveElectricity();
    }
}
```

German plug: any class that implements this interface can be plugged into a German socket.

```
Public interface GermanPlugConnector {
    public void giveElectricity();
}
```

For example, brand zest is a German plug. **Note that any class that implements GermanPlugConnector has to implement giveElectricity() method**

```
public final class ZestPlug implements GermanPlugConnector{
      public void giveElectricity(){
            System.out.println("giving electricity to a zest plug.");
      }
}
```

UK socket:

```
public class UKElectricalSocket {
    public void plugIn(UKPlugConnector plug) {
        plug.provideElectricity();
    }
}
```

UK plug interface: any class that implements this interface can be plugged into a UK socket.

```
public interface UKPlugConnector {
    public void provideElectricity();
}
```

These classes make clear that only UKPlugConnectors can be plugged into a UKElectricalSocket and only GermanPlugConnectors can be plugged into a GermanElectricalSocket.

We can even create a new brand of plug called: ZestPlug. This Zest brand implements GermanPlugConnector.

Fortunately an UKElectricalSocket can also be used with a GermanPlugConnector by using an adapter. This can be archived by wrapping a GermanPlugConnector in a UKPlugConnector.

Creating an adapter for plug connectors

To use the plugIn of UKElectricalSocket an `UKPlugConnector has to be used. Therefore, the GermanPlugConnector is wrapped in a new class, which implements the UKPlugConnector interface.

```
public class GermanToUKPlugConnectorAdapter implements UKPlugConnector {
    private GermanPlugConnector plug;
    public GermanToUKAdapter(GermanPlugConnector plug) {
        this.plug = plug;
    }
    @Override
    public void provideElectricity() {
        plug.giveElectricity();
    }
}
```

The adapter can then be used like this:

```
GermanPlugConnector plug = new ZestPlug();
UKElectricalSocket socket = new UKElectricalSocket();
UKPlugConnector ukAdapter = new GermanToUKPlugConnectorAdapter(plug);
socket.plugIn(ukAdapter);
```

Exercise 1: Download the Java files from moodle, compile using Eclipse and run the code (TestPlugs.java). Try to understand the relationships among different classes.

Exercise 2: Create a new brand plug (called Furutech) that implements UKPlugConnector. You may use ZestPlug.java as a template.

Exercise 3: Create an UKToGerman adapter that allows a UK plug such as Furutech to be plugged in a German socket.