Exercises on Design Patterns - Part I

Week 07

February 18, 2017

In these exercises we will be examining the following design patterns:

- Adapter,
- Decorator,
- Factory Method,
- Observer, and
- Singleton.

The source code examples can be found on the repository under the `exercises/week07` folder.

1. The Factory Method pattern gives us a way to encapsulate the instantiations of concrete types; it encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a *factory method*. The factory method selects an appropriate class from a class hierarchy based on the application context and other contributing factors and it then instantiates the selected class and returns it as an instance of the parent class type.

   The advantage of this approach is that the application objects can make use of the factory method to gain access to the appropriate class instance. This eliminates the need for an application object to deal explicitly with the varying class selection criteria.

   You are required to implement the following classes:

   `Product` defines the interface of objects the factory method creates.

   `ConcreteProduct` implements the `Product` interface.

   `Creator` declares the factory method, which returns an object of type `Product`. `Creator` may also define a default implementation of the factory method that returns a default `ConcreteProduct` object. We may call the factory method to create a `Product` object.

   `ConcreteCreator` overrides the factory method to return an instance of a `ConcreteProduct`.

Factory methods therefore eliminate the need to bind application-specific classes into your code. The code only deals with the `Product` interface (in this case); therefore it can work with any user-defined `ConcreteProduct` classes.

2. In this question we examine the SINGLETON design pattern.

   (a) Why might you decide to *lazy-initialise* a singleton instance rather than initialise it in its field declaration? Provide code examples of both approaches to illustrate your answer.

   (b) There are many ways to break the singleton pattern. One is in a multi-threaded environment but others include:

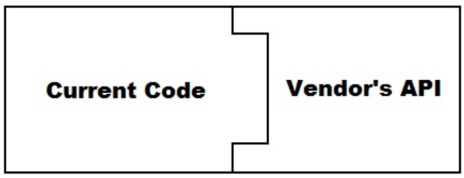
   - If the class is `Serializable`.
   - If it is `Cloneable`.
   - It can be broken by reflection.
   - If the class is loaded by multiple *class loaders*.

   Try and write a class `SingletonProtected` that addresses some (all?) of these issues.

3. We now consider the ADAPTER design pattern.

   A software developer, Max, has worked on an e-commerce website. The website allows users to shop and pay online. The site is integrated with a third party payment gateway, through which users can pay their bills using their credit card. Everything was going well, until his manager called him for a change in the project.

   The manager has told him that they are planning to change the payment gateway vendor, and Max has to implement that in the code. The problem that arises here is that the site is attached to the `Xpay` payment gateway which takes an `Xpay` type of object. The new vendor, `PayD`, only allows the `PayD` type of objects to allow the process. Max doesn't want to change the whole set of a hundred classes which have reference to an object of type `XPay`. He cannot change the third party tool provided by the payment gateway. The problem arises due to the incompatible interfaces between the two different parts of the code. To get the process to work, Max needs to find a way to make the code compatible with the vendor's provided API.



   The current code interface is not compatible with the new vendor's interface. What Max needs here is an ADAPTER which can sit in between the code and the vendor's API, enabling the transaction to proceed.

   (See `xpay/Xpay.scala`

   The `Xpay` interface contains setter and getter methods to get the information about the credit card and customer name. The interface is implemented in the following code which is used to instantiate an object of this type, and exposes the object to the vendor's API.

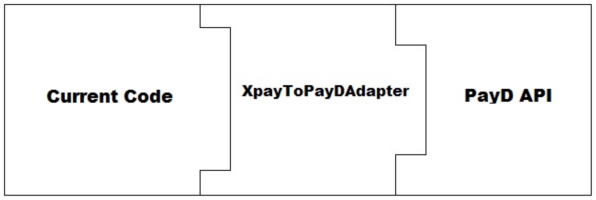
   (See `xpay/XpayImpl.scala`)

New vendor's interface looks like this:

(See `xpay/PayD.scala`)

As you can see, this interface has a set of different methods which need to be implemented in the code. However, `Xpay` objects are created by most parts of the code, and it is difficult (and risky) to change the entire set of classes. We need some way, that's able to fulfil the vendor's requirement to process the payment and also make less or no change to the current code base.

Your are required to use the Adapter pattern to implement a `XpayToPayDAdapter` class to meet the requirements.



4. This question examines the OBSERVER design pattern.

   Sports Lobby is a sports website targeted at sport lovers. They cover almost all kinds of sports and provide the latest news, information, matches scheduled dates, information about a particular player or a team. Now, they are planning to provide live commentary or scores of matches as an SMS service, but only for their premium users. Their aim is to SMS the live score, match situation, and important events after short intervals. As a user, you need to subscribe to the package and when there is a live match you will get an SMS to the live commentary. The site also provides an option to unsubscribe from the package whenever a user wants to.

   As a developer, the Sport Lobby has asked you to provide this new feature for them. The reporters of the Sport Lobby will sit in the commentary box in the match, and they will update live commentary to a commentary object. As a developer your job is to provide the commentary to the registered users by fetching it from the commentary object when it's available. When there is an update, the system should update the subscribed users by sending them the SMS.

   This situation clearly indicates a *one-to-many* mapping between the match and the users, as there could be many users subscribed to a single match. The OBSERVER design pattern is best suited to this situation — you should implement this feature for Sport Lobby using the OBSERVER pattern.

   Remember that there are four participants in the OBSERVER pattern:

   `Subject` which is used to register observers. Objects use this interface to register as observers and also to remove themselves from being observers.

   `Observer` defines an updating interface for objects that should be notified of changes in a subject. All observers need to implement the `Observer` interface. This interface has a method `update()`, which gets called when the `Subject`'s state changes.

   `ConcreteSubject` stores the state of interest to `ConcreteObserver` objects. It sends a notification to its observers when its state changes. A concrete subject always implements the `Subject` interface. The `notifyObservers()` method is used to update all the current observers whenever the state changes.

`ConcreateObserver` maintains a reference to a `ConcreteSubject` object and implements the `Observer` interface. Each observer registers with a concrete subject to receive updates.

(See `observer/Observer.scala`, `observer/Subject.scala`, `observer/Commentary.scala`, and `observer/TestObserver.scala`.)

5. This question considers the DECORATOR design pattern.

   You are commissioned by a pizza company make an extra topping calculator. A user can ask to add extra topping to a pizza and our job is to add toppings and increase its price using our classes.

   Please note: the main aim of the DECORATOR design pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality. The Decorator prevents the proliferation of subclasses leading to less complexity and confusion.

   For simplicity, let's create a simple `Pizza` interface which contains only two methods:

   (See `decorator/Pizza.scala`).

   The `getDesc` method is used to obtain the pizza's description whereas the `getPrice` is used to obtain the price.

   Provide two implementations of the `Pizza` interface:

   - `SimplyVegPizza`
   - `SimplyNonVegPizza`

   The decorator wraps the object whose functionality needs to be increased, so it needs to implement the same interface. Provide an abstract decorator class which will be extended by all the concrete decorators.

   ```
   public abstract class PizzaDecorator implements Pizza
   ```

   Now provide several implementations of `PizzaDecorator` and exercise your classes with the given test class.

   - `Ham extends PizzaDecorator`
   - `Cheese extends PizzaDecorator`
   - `Chicken extends PizzaDecorator`
   - `FetaCheese extends PizzaDecorator`
   - ...

   (See `decorator/TestDecoratorPattern.scala`)

   The code will result in the following output:

```
Desc: SimplyVegPizza (230), Roma Tomatoes (5.20), Green Olives (5.47),
        Spinach (7.92)
Price: 248.59
Desc: SimplyNonVegPizza (350), Meat (14.25), Cheese (20.72),
        Cheese (20.72), Ham (18.12)
Price: 423.81
```