SDP Week 3 - Exercises on Design Patterns

Mark Channer

Patterns: Adapter, Decorator, Factory Method, Observer (MVC) and Singleton

Short form questions

1. **Write down three differences between abstract classes and interfaces in Java 8. Provide examples to illustrate your answer.**

    1. A class can implement multiple interfaces, but can only extend one abstract class.

    2. An abstract class can have instance fields, but an interface cannot because it does not have 'state', although it can create static final constants.

    3. While abstract classes are from Object (and therefore come with Object's methods), interfaces come from anything

    4. Any methods in an interface that are not declared to be default or static are implicitly abstract, whereas unimplemented methods in an abstract class must be declared to be abstract with the abstract keyword.

    5. An abstract class can have a constructor, an interface cannot.

    6. Access modifiers possible with abstract class methods are public, protected, package-private and private, whereas methods in an interface are implicitly public.

2. **Are the following true or false? (I've deliberately chosen not to type out examples of invalid code as feel it would do me more harm than good)**

    (a) Every interface must have at least one method.
    False. Marker interfaces, such as Cloneable and Serializable, do not have any methods and exists only to show that any class implementing this interface has some type of behaviour or property associated with it

    (b) An interface can declare instance fields that an implementing class must also declare.
    False. An interface does not hold state and any variables must be static.

    (c) Although you can't instantiate an interface, an interface definition can declare constructor methods that require an implementing class to provide constructors with given signatures.
    False. An interface cannot contain constructor methods.

3. **Provide an example of an interface with methods that do not imply responsibility on the part of the implementing class to take action on behalf of the caller or to return a value.**

   When an interface is used to register for updates, such as in the Observer pattern, any class implementing it may need to take some action when one of the implemented methods is called, but it is generally not performing the action for the caller. One example of this is the MouseMotionListener interface.

4. **What is the value of a stub class like WindowAdapter which is composed of methods that do nothing?**

```
package stub;

public interface WindowListener {

        void windowOpened();

        void windowClosing();

        void windowClosed();

        void windowIconified();

        void windowDeiconified();

        void windowActivated();

        void windowDeactivated();

}

package stub;

public class WindowAdapter implements WindowListener {

        @Override
        public void windowOpened() {}

        @Override
        public void windowClosing() {}

        @Override
        public void windowClosed() {}

        @Override
        public void windowIconified() {}

        @Override
        public void windowDeiconified() {}
        @Override
        public void windowActivated() {}

        @Override
        public void windowDeactivated() {}

}
```

If a concrete class implements WindowListener, all of the methods defined within it must be implemented whether they have a use or not. However, if someone were to instead extend WindowAdapter, they could still use WindowListener as a reference type, but would only need to implement the methods that they have a use for. Since Java 8, if the methods in the WindowListener were to be made default, there would not be any need for the WindowAdapter stub.

5. **How can you prevent other developers from constructing new instances of your class? Provide appropriate examples to illustrate your answer.**
   This would be a job for the Singleton design pattern. The main characteristics of which are:
   - A private constructor to prevent unwanted instantiation
   - A private static reference variable of the same class that references the only instantiation of the object
   - A public static method that returns a reference to the above object and is the only point of access to it

   Example:

```
public class Singleton {

    private static Singleton instance;

    private Singleton() {}

    static {
        instance = new Singleton();
    }

    public static Singleton getInstance() {
        return instance;
    }

}
```

6. **Why might you decide to lazy-initialise a singleton instance rather than initialise it in its field declaration? Provide examples of both approaches to illustrate your answer.**
   If you wished to conserve memory, lazy initialisation will ensure that the singleton object is not created until it is called for. However, care must be taken with this approach if the application is concurrent.

   Eager initialisation:

```java
public class EagerSingleton {

    private static EagerSingleton instance;

    private EagerSingleton() {}

    static {
        instance = new EagerSingleton();
    }

    public static EagerSingleton getInstance() {
        return instance;
    }

}
```

Lazy initialisation:

```java
public class LazySingleton {

    private static LazySingleton instance;

    private LazySingleton() {}

    public synchronized static LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

7. **Using the java.util.Observable and java.util.Observer classes/interfaces show how one object can be informed of updates to another object.**

```java
import java.util.Observable;

public class PowerStation extends Observable {

    private String state = "YELLOW";

    public void changeState(String state) {
        this.state = state;
        setChanged();
        notifyObservers();
    }

    public String getState() {
        return state;
    }
}
```

```java
import java.util.Observable;
import java.util.Observer;

public class StationMonitor implements Observer {

    String monitorName;

    public StationMonitor(String name) {
        monitorName = name;
    }

    @Override
    public void update(Observable observable, Object obj) {
        PowerStation powerStation = (PowerStation) observable;
        System.out.println(monitorName + " state changed to: " +
                powerStation.getState());
    }
}
```

```java
public class ObserverExample {

    public static void main(String[] args) {
        PowerStation powerStation = new PowerStation();
        powerStation.addObserver(new StationMonitor("1"));
        powerStation.addObserver(new StationMonitor("2"));
        powerStation.addObserver(new StationMonitor("3"));

        powerStation.changeState("YELLOW");
        powerStation.changeState("RED");
        powerStation.changeState("BLUE");
    }
}
```

8. **“The Observer pattern supports the MVC pattern”. State if this statement is true or false and support your answer by use of an appropriate example.**
   It is true that the Observer pattern supports the MVC pattern. For example, the model is the (observable) subject that, when its state changes, updates any views/controllers that have registered with it as an observer. In this way, the model is able to remain independent of the views and controllers, and more than one type of view is able to be used.

9. **Provide examples of two commonly used Java methods that return a new object.**
   toString() returns a string representation of an object.
   clone() returns a copy of the object from which it is called.

10. **What are the signs that a Factory Method is at work?**
    Signs that a factory method is at work include the following:
    - A new object is created
    - An abstract class or interface type is returned
    - It is implemented by several classes (although it is possible that only one class is implementing it)

11. **If you want to direct output to System.out instead of to a file, you can create a Writer object that directs its output to System.out:**

    Writer out = new PrintWriter(System.out);

    Write a code example to define a Writer object that wraps text at 15 characters, centres the text, sets the text to random casing, and directs the output to System.out.
    Which design pattern are you using?
    Although I could see that the Decorator pattern could be used here, I had some trouble with what to implement for this question. The below code is taken from the example that Keith posted to Moodle on 24th January.

```java
import java.io.BufferedWriter;
import java.io.PrintWriter;

public class WrapFilterExample {

    public static void main(String[] args) {
        WrapFilter out = new WrapFilter(
                new BufferedWriter(
                        new RandomCaseFilter(
                                new PrintWriter(System.out))), 15);
        out.setCenter(true);
    }
}
```

Long form questions

1. The Factory Method design pattern.

   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.

```
interface Product { }

class ConcreteProduct implements Product { }

abstract class Creator {

    public void doSomething() {
        Product product = factoryMethod();
        // then call relevant method on product
    }

    protected abstract Product factoryMethod(); // or implements a default

}

class ConcreteCreator extends Creator {

    @Override
    protected Product factoryMethod() {
        return new ConcreteProduct();
    }
}

public class Client {
    public static void main(String[] args) {
        Creator creator = new ConcreteCreator();
        creator.doSomething();
    }
}
```

2. The Singleton design pattern.

If you didn't provide implementations of a *lazy* and *eager singleton pattern* in Question 6 do so now. (You should provide a static *getInstance* method.)

Imagine that we now wish to use the code in a *multi-threaded environment*. Two threads concurrently access the class, thread t1 gives the first call to the *getInstance()* method, it will check if the static variable that holds the reference to the singleton instance is null and then gets interrupted due to some reason. Another thread t2 calls the *getInstance()* method, successfully passes the instance check and instantiates the object. Then, thread t1 wakes and it also creates the object. At this time, there would be two objects of this class which was supposedly a singleton.

(a) How could we use the *synchronized* keyword to the *getInstance()* method to operate correctly.

One option would be to use the *synchronized* keyword in the getInstance() method signature as in the code below. This would ensure that no other thread would be able to enter getInstance() while another thread is in there, even if it is sleeping. This would prevent the detailed scenario from happening and ensure that there was no more than one instantiation of the object.

```
public synchronized static LazySingleton getInstance() {
    if (instance == null) {
        instance = new LazySingleton();
    }
    return instance;
}
```

(b) The synchronised version comes with a price as it will decrease the performance of the code — why?
Synchronization is only useful up until an instance has been created. After this, it is no longer needed as the instance variable will no longer have a null reference. Even so, any other threads that have been spawned to get a reference to a singleton instance would still have to wait until there is no other thread in the getInstance() method before entering, reducing concurrent capabilities.

(c) If the call to the *getInstance()* method isn't causing a substantial overhead for your application, then you can forget about it.
True.

(d) If you want to use synchronisation (or need to), then there is another technique known as double-checked locking which reduces the use of synchronisation. With double-checked locking, we first check to see if an instance is created, and if not, then we *synchronise*.

Provide a sample implementation of this technique.

```
class Singleton {

    private volatile static Singleton instance;

    private Singleton() { }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

There are some other ways to break the singleton pattern:

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

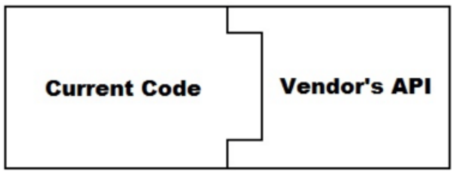
Try and write a class *SingletonProtected* that addresses some (all?) of these issues.
* Idea for below code from p.57 of '*Java Design Pattern Essentials'* by T.Bevis

```
public enum Singleton {

    INSTANCE;

    public void doSomething() {
        // call with Singleton.INSTANCE.doSomething();
    }

}
```

3. 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.

package xpay;

```
public interface Xpay {
    String getCreditCardNo();
    void setCreditCardNo(String creditCardNo);
    String getCustomerName();
    void setCustomerName(String customerName);
    String getCardExpMonth();
    void setCardExpMonth(String cardExpMonth);
    String getCardExpYear();
    void setCardExpYear(String cardExpYear);
    Short getCardCVVNo();
    void setCardCVVNo(Short cardCVVNo);
    Double getAmount();
    void setAmount(Double amount);
}
```

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.

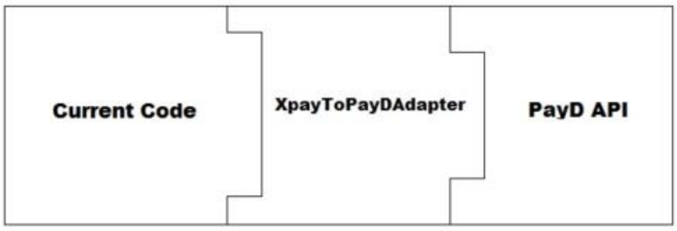
```
package xpay;

@lombok.Data
public class XpayImpl implements Xpay {
    private String creditCardNo;
    private String customerName;
    private String cardExpMonth;
    private String cardExpYear;
    private Short cardCVVNo;
    private Double amount;
}
```

New vendor's interface looks like this:

```
package xpay;

public interface PayD {
    String getCustCardNo();
    void setCustCardNo(String custCardNo);
    String getCardOwnerName();
    void setCardOwnerName(String cardOwnerName);
    String getCardExpMonthDate();
    void setCardExpMonthDate(String cardExpMonthDate);
    Integer getCVVNo();
    void setCVVNo(Integer cVVNo);
    Double getTotalAmount();
    void setTotalAmount(Double totalAmount);
}
```

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.

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



To be completed….