SOEN 387
WEB-BASED ENTERPRISE APPLICATIONS DESIGN

TUTORIAL – 8

Software Design Patterns

Agenda

- Design patterns: what we cover
- Singleton Pattern
- Example: enum singleton
- Strategy Pattern
- Example: Strategy for file compression
- Observer Pattern
- Example: Stock Observer

Design patterns: what we cover

- Singleton
- Strategy
- Observer

Singleton Pattern

What: The singleton pattern is a design pattern that restricts the instantiation of a class to one object.

Why: Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly.

```
class Singleton
{
    private volatile static Singleton obj;

    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null)
        {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple
threads
                // can reach above step
                if (obj==null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}
```

volatile which ensures that multiple threads offer the obj variable correctly when it is being initialized to Singleton instance.

getInstance method ensures that only one instance of the class is created.

Example: enum singleton

An Enum is singleton by design. All the enum values are initialized only once at the time of class loading.

```java
/** An Enum value is initialized only once at the time of class loading.
   It is singleton by design and is also thread-safe.
 */
enum EnumSingleton {
   WEEKDAY("Monday", "Tuesday", "Wednesday", "Thursday", "Friday"),
   WEEKEND("Saturday", "Sunday");
   private String[] days;
   EnumSingleton(String ...days) {
      System.out.println("Initializing enum with " + Arrays.toString(days));
      this.days = days;
   }
   public String[] getDays() {
      return this.days;
   }
  @Override
   public String toString() {
      return "EnumSingleton{" +
            "days=" + Arrays.toString(days) +
            '}';
   }
}
```

```java
public class EnumSingletonExample {
   public static void main(String[] args) {
      System.out.println(EnumSingleton.WEEKDAY);
      System.out.println(EnumSingleton.WEEKEND);
   }
}
```

```
# Output

Initializing enum with [Monday, Tuesday, Wednesday,
Thursday, Friday]

Initializing enum with [Saturday, Sunday]

EnumSingleton{days=[Monday, Tuesday, Wednesday, Thursday,
Friday]}

EnumSingleton{days=[Saturday, Sunday]}
```

Strategy Pattern

What: It is a software design pattern that enables an algorithm's behavior to be selected at runtime.

Why: A good use of the Strategy pattern would be saving files in different formats, running various sorting algorithms, or file compression. When you have different strategies and at runtime application decides the correct strategy to fire.

In class exercise : Strategy Implementation

AIM: Saving files in different file format (zip and rar) and use strategy pattern to provide the implementation of each.

Step 1: create interface

```
public interface CompressionStrategy {
                /* Interface code goes here */
}
```

Step 2: provide two implementations, one for zip and one for rar

```
public class ZipCompressionStrategy implements INTERFACE_NAME {
                  /* Implementation goes here */ {
                   //using ZIP approach
                  }
                }
public class RarCompressionStrategy implements INTERFACE_NAME {
                 /* Implementation goes here */ {
                   //using RAR approach
                  }
}
```

Step 3: create context class to provide a way to compress the files.

```
public class CompressionContext {
  private CompressionStrategy strategy;
  //this can be set at runtime by the application preferences
  /* your code for setting compression strategy goes here */
  }
  //use the strategy
  public void createArchive(ArrayList<File> files) {
    strategy.compressFiles(files);
  }
}
```

context will provide a way to compress the files.

change in strategy can be done using setComrpression method inside context

Step 4: main class

```
public class Client {
  public static void main(String[] args) {
    CompressionContext ctx = new CompressionContext();
    //we could assume context is already set by preferences
    ctx.setCompressionStrategy(new ZipCompressionStrategy());
    //get a list of files...
    ctx.createArchive(fileList);
  }
}
```

Refer here

Observer Pattern

What: The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. The object which is being watched is called the subject. The objects which are watching the state changes are called observers or listeners.

Why: Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change.

In class exercise : Observer Implemenatation

AIM: *StockMarket* is the **subject** and *StockBroker* is the **Observer**.
StockBroker has two concrete implementations:***StockBuyer*** and ***StockViewer***.
Whenever the *stockMarket* object is changed, these two concrete observers get an update via their function ***update***().

Step 1: create interface StockBroker

```
import java.util.Map;
/**
 * Observer interface
 */

interface StockBroker {

    /* code for updating the stock list goes here */
}
```

Step 2: create implementation of abstract observer

```
import java.util.Iterator;
import java.util.Map;
/**
 * Here, StockBuyer and StockViewer are concrete Observers
 */
public class StockBuyer implements INTERFACE_NAME {
   public void update(Map<String, Double> stocks) {

      /* Implementation goes here */

   }
}

public class StockViewer implements StockBroker {
   public void update(Map<String, Double> stocks) {

      /* Implementation goes here */

   }
}
```

Step 3: create subject class that will notify observers in case of any change

```
/**
 * Subject
 */
public abstract class AbstractStockMarket {
   private List<StockBroker> stockBrokers = new
ArrayList<StockBroker>();

   public void addStockBroker(StockBroker
stockBroker) {
      /* code for adding the stockBroker goes here */
   }

   public void notifyStockBroker(Map<String, Double>
stockList) {
      /* notifying observer code goes here */
      }
   }

 /* abstract method addStock goes here */
 /* abstract method update goes here */

}
```

Step 4: create main class

```
/**
 * Client
 */
public class Application {
   public static void main(String[] args) {
      AbstractStockMarket market = new StockMarket();
      StockBroker buyer = new StockBuyer();
      StockBroker viewer = new StockViewer();
      market.addStockBroker(buyer);
      market.addStockBroker(viewer);
      market.addStock("ORC", 12.23);
      market.addStock("MSC", 45.78);
      System.out.println("===== Updating ORC =====");
      market.update("ORC", 12.34);
      System.out.println("===== Updating MSC =====");
      market.update("MSC", 44.68);
   }
}
```

Refer here

Consequence of observer pattern in Servlet

Assume a Java HttpServlet that contains a set of objects that make use of the observer pattern in order to return data through the servlet's Response object.

```
protected void doGet(final HttpServletRequest request, final HttpServletResponse response) {
        MyProcess process = new MyProcess();
        // This following method spawns a few threads, so I use a listener to receive a completion event.
        process.performAsynchronousMethod(request, new MyListener() {
                                public void processComplete(data) {
                                response.getWriter().print(data.toString());
                                }
        }
}
```

Problem: as the doGet() method completes, the response object becomes null. When processComplete() is called, the response object will be null - thus preventing writing any data out. Here, servlet is closes the connection as soon as the asynchronous method is called.

Solution: The servlet response will be sent back to the client when the doGet method terminates, it won't wait for asynchronous call to finish as well. Need to block until all asynchronous tasks have been completed, and only then allow the doGet() method to return.

- Minimal coupling between Subject and Observer.The subject does not require knowledge of the observer. The observer only needs to know how to get new data.
- Support for broadcast communication.
- An update() triggers a broadcast communication across all observers.
- Unexpected updates.
- The subject is blind to its observer. Thus, the cost of an update() is unknown.
- Observers have no control over when they will receive updates.