CSCI4448/5448 : Homework 3
Design Patterns

These are some great practice problems focused on applying a specific design pattern to improve the code.

Problem #1: Auto-grader Submission Management

Scenario

You are performing an update to an auto-grader system. The auto-grader receives submissions asynchronously, runs a test script, and produces a report. Looking closely at the process, you notice that the process could be easily multi-threaded! You implement the multithreaded aspect immediately, however, you still need to figure out a good way to queue submissions. You have a server which accepts submissions via several different paths, each with its own thread, and a set of Grader objects, each running in its own thread.

Problem

Your task is to create a class which consists of a queue (i.e., FIFO list) of Submission objects. For this example, submissions can be a dummy class (see below). This class must implement two methods: *add(Submission s)* and *process(Submission s)*. Additionally, you want to ensure that only one queue exists. Otherwise, every time a submission is submitted, a new queue will be created, and a Grader object will also not know which queue to select the next submission from.

Starter Code

```
/**
* Submission.java
*
* A basic dummy Submission object for use with the auto-grader
*/

import java.util.Random;

public class Submission
{
  private static Random rand = new Random();
  private int id;

  public Submission()
  {
    // Give this submission a unique(ish) id
    id = rand.nextInt(10000000);
  }
}
```

Problem #2: Auto-grader Reporting

Scenario

Once again, you are performing a much needed update to a particular auto-grader for a certain computer science course. This time, however, you are trying to make the reporting mechanism easier to interact with the specific student submissions. In particular, you would like to generate two types of reports for each submission--one report simply involves counting the total number of correctly passed test cases, while the other report determines the number of time-out errors in a submission. Being wary of how the auto-grader has evolved in the past, you expect extra reports to be created in the future.

Problem

Your task is to create a pair of Report classes which maintain a specific type of report for a Submission object. The Submission class has been provided, but you must create the Report classes. After every test case, the submission should inform the Report objects of the results (i.e., pass or fail). If the test case failed, the error report should ask the Submission object if the most recent test failed due to a time-out (using the *wasTimeoutError* method).
You may add methods to the submission class as needed.
Note: Since the Submission object may be running on a separate thread from the Report objects, this is not simply a matter of running a test case, and then asking for the results.

For simplicity, reports can simply involve printing to the console.

```
/**
* Submission.java
* A representation of a Submission
*/
import java.util.Random;

public class Submission
{
    private Random myRandom;
    private boolean lastErrorWasTimeout;
    // You may add attributes to this class if necessary

    public Submission()
    {
        myRandom = new Random();
        lastErrorWasTimeout = false;
    }
    public void runTestCase()
    {
        // For now, randomly pass or fail, possibly due to a timeout
        boolean passed = myRandom.nextBoolean();
        if(!passed)
        {
            lastErrorWasTimeout = myRandom.nextBoolean();
        }
        // You can add to the end of this method for reporting purposes
    }
    public boolean wasTimeoutError()
    {
        return lastErrorWasTimeout;
    }
}
```

Problem #3: Logging System

Fix the following code based on an appropriate design pattern (Where is the 'smelly' code?).

```
interface Logging {
    public void log(String msg);
}
class LogText implements Logging
{
    public LogText()
    {
        System.out.println("Logging: text format");
    }
    public void log(String msg)
    {
        System.out.println("Logging text to file: " + msg);
    }
}
class LogXML implements Logging
{
    public LogXML()
    {
        System.out.println("Logging: <type>XML Format</type>");
    }
    public void log(String msg)
    {
        System.out.println("Logging text to file: log.xml" );
        System.out.println("<xml><msg>"+msg+"</msg></xml>");
    }
}
class LogHTML implements Logging
{
    public LogHTML()
    {
        System.out.println("Logging: HTML format");
    }
    public void log(String msg)
    {
        System.out.println("Logging HTML to file: log.html" );
        System.out.println("<html><body>"+msg+"</body></html>");
    }
}
class Analysis
{
    public static void main(String[] args)
    {
        if (args.length != 1)
         {
            System.out.println("Usage: java Analysis type");
            System.exit(-1);
         }
        String type = args[0];
        Logging logfile;
        if (type.equalsIgnoreCase("text"))
            logfile = new LogText();
        else if (type.equalsIgnoreCase("xml"))
            logfile = new LogXML();
        else if (type.equalsIgnoreCase("html"))
            logfile = new LogHTML();
        else
            logfile = new LogText();
        logfile.log("Starting application...");
   }
}
```

Problem #4: Boulder Trapper Company

Scenario

Having just graduated from CU, you feel like you can tackle the world, so you decide to start your own tech startup! You have teamed up with Alice, an expert in mechanical engineering, and Bob, a brilliant friend from high school who just graduated from Rhode Island School of Design. Together, you start the "Boulder Trapper Company", and develop custom intruder deterrent systems. Alice develops several mechanical components, each controlled by a small microcontroller. Alice made sure to build a nice API for every component she created. Bob's job is to provide a specific design for individual clients. You've trained Bob on basic programming, but Bob is complaining that he is spending way too much time writing complicated code, and not enough time refining his designs. Bob has given you three examples of code he has written, and Alice has provided you with the API for her devices. Looking at the three examples, you notice that Bob seems to be writing the portions of the same code in each case.

Problem

Determine a way to make Bob's job simpler, based on the existing code. Bob should be able to use your solution for future projects as well. Your solution should not impede use of Alice's existing API.
Include a re-write of Bob's three examples using your improved system, so that Bob has examples to mimic in the future.

```
/**
* GlueSpot.java
*
* One of Alice's trap components.  A glue spot causes an intruder to stick
* to the floor
*/
public class GlueSpot
{
  public GlueSpot()
  {
  }
  public void stick()
  {
    System.out.println("Haha!  The intruder got stuck to the floor!");
  }
}
```

```
**
* Rope.java
* One of Alice's trap components.  A rope will tie up an intruder.
*/
public class Rope
{
  public Rope()
  {
  }
  public void tie()
  {
    System.out.println("Intruder is tied up!");
  }
}
```

```java
/**
* Net.java
* One of Alice's trap components.  A net will entangle an intruder, making it
* difficult to move
*/
public class Net
{
  public Net()
  {
  }
  public void entangle()
  {
    System.out.println("Haha! The intruder got tangled up in a net!");
  }
}
```

```java
/**
* ExampleTrap1.java
*
* One of Bob's traps
*/

public class ExampleTrap1
{
  public ExampleTrap1()
  {
    // This trap consists of 2 glue spots, followed by 3 nets,
    // and finished with tying up with rope

    GlueSpot glueSpot1 = new GlueSpot();
    GlueSpot glueSpot2 = new GlueSpot();
    Net net1 = new Net();
    Net net2 = new Net();
    Net net3 = new Net();
    Rope rope1 = new Rope();
  }

  public void trigger()
  {
    glueSpot1.stick();
    glueSpot2.stick();
    net1.entangle();
    net2.entangle();
    net3.entangle();
    rope1.tie();
  }
}
```

```java
/**
* ExampleTrap2.java
* One of Bob's traps
*/

public class ExampleTrap2
{
  public ExampleTrap2()
  {
    // This trap consists of four glue spots, followed by two nets,
    // and finished with tying up with three ropes

    GlueSpot glueSpot1 = new GlueSpot();
    GlueSpot glueSpot2 = new GlueSpot();
    GlueSpot glueSpot3 = new GlueSpot();
    GlueSpot glueSpot4 = new GlueSpot();

    Net net1 = new Net();
    Net net2 = new Net();

    Rope rope1 = new Rope();
    Rope rope2 = new Rope();
    Rope rope3 = new Rope();
  }

  public void trigger()
  {
    glueSpot1.stick();
    glueSpot2.stick();
    glueSpot3.stick();
    glueSpot4.stick();

    net1.entangle();
    net2.entangle();

    rope1.tie();
    rope2.tie();
    rope3.tie();
  }
}
```

```java
/**
* ExampleTrap3.java
*
* One of Bob's traps
*/

public class ExampleTrap3
{
  public ExampleTrap3()
  {
    // This trap consists of 2 glue spots and rope to tie up the intruder
    // There's not enough room in the building for nets :(

    GlueSpot glueSpot1 = new GlueSpot();
    GlueSpot glueSpot2 = new GlueSpot();

    Rope rope = new Rope();
  }

  public void trigger()
  {
    glueSpot1.stick();
    glueSpot2.stick();

    rope.tie();
  }
}
```

Problem #5: General Game Playing AI

Scenario

General game playing is a field of artificial intelligence where a game playing agent is programmed to play any type of game. The agent is not aware of which game it will play until run-time, when it is provided the rules to the game, so there is no way to generate a specific algorithm for your player. The basic concept is to generalize the idea of a game, and to create an algorithm for determining the best move for a given type of game based on some criteria, such as timers, game complexity, etc. Note that different algorithms may be suitable for different situations (such as move selection time, board size, etc.)

Problem

Your task is to generate a general game player class, which can easily select a method of playing based on the context of the game. Your game player should be able to play a game using one of three different types of approached: *TreeSearch, Minimax,* and *MonteCarlo*. The game player should implement two public methods:

- *setup*
- *selectMove*

The playing approach to use should be selected based on the following criteria: If the board is small (i.e., size is less than 30), then *TreeSearch* should be used. If the board is large (i.e., size is greater than or equal to 30) and the moveSelectionTime large (i.e., greater than or equal to 60), then *Minimax* should be used. Finally, if board size is large and moveSelectionTime is small, then *MonteCarlo* should be used.

A skeleton class for *GamePlayer* is provided, with the method signatures for the two methods described above. You may add attributes / methods to this class as needed, but the Game will interact with the GamePlayer through these two methods. Your solution should involve creating a set of new classes, each implementing a game playing approach. Your solution should be easily extendible (i.e., adding a new playing technique involves creating a new class). An incorrect solution involves creating a new method in *GamePlayer* for each play approach.

```
/**
* GamePlayer.java
* A simple game player class
*/
public class GamePlayer
{
    public GamePlayer()
    {
        // Initialize attributes as necessary
    }
    public void setup(int boardSize, int moveSelectionTime)
    {
        // Select appropriate method of play based on arguments
    }
    public void selectMove()
    {
        // A dummy method.  Your code simply needs to print to the console
        // "I'm making a move using the _______ AI algorithm."
    }
}
```

Problem #6: Video Game Trees

Scenario

You are working at Video Games Ultra and they have a game that creates a bunch of trees for the terrain. They need to make a huge forest with thousands of trees placed randomly throughout the landscape.

Problem

When they scale the program up for the thousands of trees, the program is slowed down too much and takes up too many resources. Your task is to modify the provided code so that it doesn't slow down from too much overhead of creating tons of image objects.

```
import java.util.*;
import java.io.*;
import java.awt.*;
import javax.swing.*;
import javax.imageio.*;

interface Terrain
{
    void draw(Graphics graphics, int x, int y);
}
class Tree implements Terrain
{
    private int x;
    private int y;
    private Image image;
    public Tree(String type)
    {
        System.out.println("Creating a new instance of a tree of type " + type);
        String filename = "tree" + type + ".png";
        try
        {
            image = ImageIO.read(new File(filename));
        } catch(Exception exc) { }
    }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
    @Override
    public void draw(Graphics graphics, int x, int y)
    {
        graphics.drawImage(image, x, y, null);
    }
}
class TreeFactory
{
    private static final ArrayList<Tree> mylist = new ArrayList<Tree>();
    public static Terrain getTree(String type)
    {
        Tree tree = new Tree(type);
        mylist.add(tree);
        return tree;
   }
}
```

```
/**
 * Don't change anything in TreeDemo
 */
class TreeDemo extends JPanel
{
    private static final int width = 800;
    private static final int height = 700;
    private static final int numTrees = 50;
    private static final String type[] = { "Apple", "Lemon", "Blob", "Elm", "Maple" };

    public void paint(Graphics graphics)
    {
        for(int i=0; i < numTrees; i++)
        {
            Tree tree = (Tree)TreeFactory.getTree(getRandomType());
            tree.draw(graphics, getRandomX(width), getRandomY(height));
        }
    }
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        frame.add(new TreeDemo());
        frame.setSize(width, height);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
    private static String getRandomType()
    {
        return type[(int)(Math.random()*type.length)];
    }
    private static int getRandomX(int max)
    {
        return (int)(Math.random()*max );
    }
    private static int getRandomY(int max)
    {
        return (int)(Math.random()*max);
    }
}
```