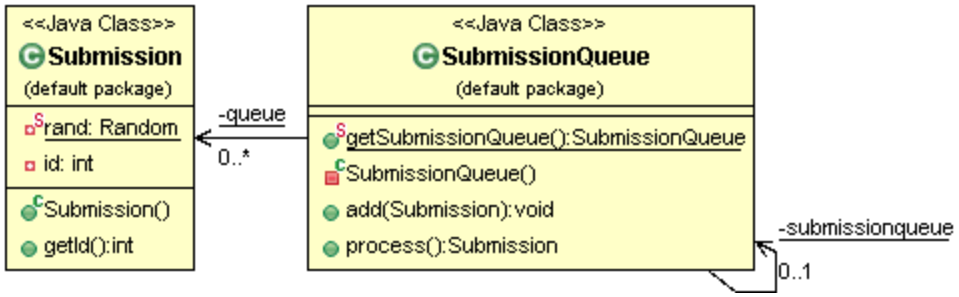
Samuel Volin
Cris Salazar
HW3 Writeup

Problem Set 1

1. **Problem Set 1**
2. The **Singleton** design pattern was used to create a queue that could add submissions and process submissions. To prevent multiple SubmissionQueues from being spawned, the singleton pattern was implemented with a private constructor and public static getter, so that only a single accounted for instance of SubmissionQueue could ever be instantiated and referenced.
3. 

4. **Samuel Volin wrote the classes for this portion**
5.

Work Percentage	Name	Work Done
90%	Sam	Submission, SubmissionQueue, most of the Problem1Driver
10%	Cris	answered a few questions, pair programmed a little

6.

```
//Program Driver
//This is the driver that is used to test the code
//It shows that every instance of the report updates correctly
//Thus showing the observer design pattern
public class SubmissionDriver {

	public static void main(String[] args){
		Submission sub = new Submission();
	for(int i = 0; i < 10; i++){
		sub.runTestCase();
	}

	int incorrect = 10 - sub.getCReport().getNumberCorrect() -
```

```
sub.getTReport().getNumberOfTimeOutErrors();
        System.out.println("Number of Correct: " +
sub.getCReport().getNumberCorrect());
        System.out.println("Number of Incorrect w/o Timeout: " +  incorrect);
        System.out.println("Number of Timeout Errors: " +
sub.getTReport().getNumberOfTimeOutErrors());
   }

}
```

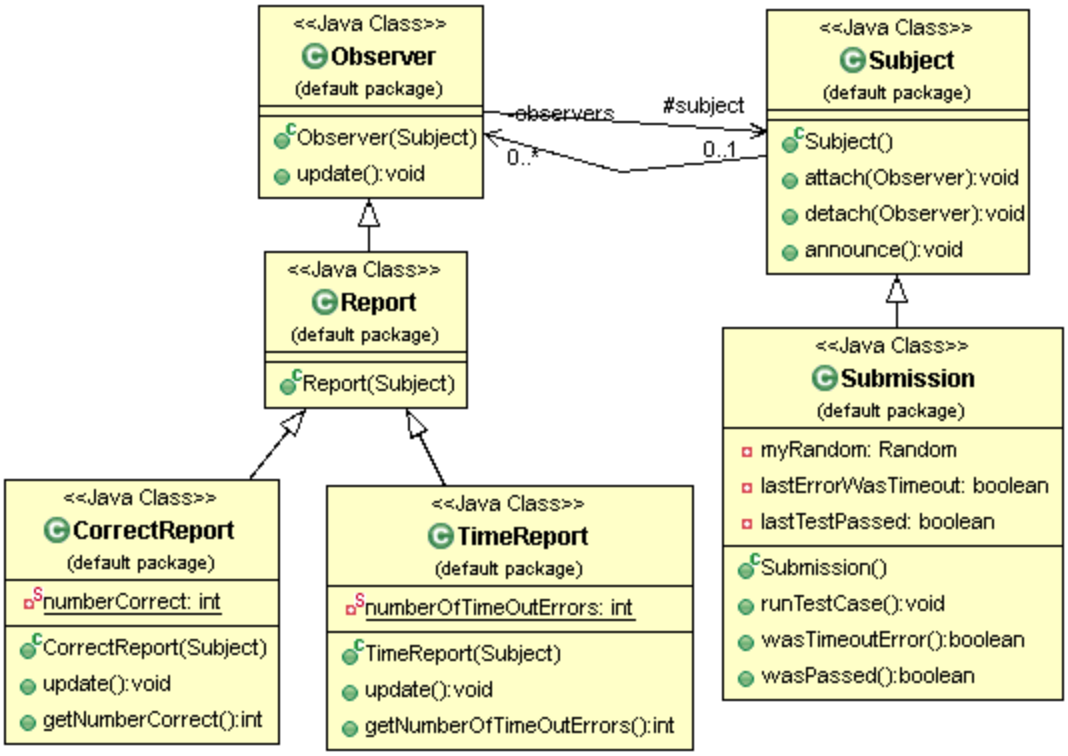
Sample output:

```
Time-out Error Found
Time-out Error Found
Time-out Error Found
Time-out Error Found
Number of Correct: 4
Number of Incorrect w/o Timeout: 2
Number of Timeout Errors: 4
```

7. **CODE:**

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

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);
  }

  public int getId()
  {
            return id;
  }
}
```

Problem Set 2

1. **Problem Set 2**
2. The **Observer** design pattern is used to implement a submission report system. The two report classes, TimeReport and CorrectReport each keep track of the number of timeout errors and number of correct answers respectively. They do this through an update function. This design pattern is implemented through the use of Observer and Subject abstract classes.
3. 

4. **Cris Salazar and Samuel Volin wrote the classes for this portion**
5. **Work**

Work Percentage	Name	Work Done
35%	Sam	Discussed potential design patterns, fixed implementation before submission
65%	Cris	Implemented all code, implemented design pattern, written questions

6. **Examples**

```
//This is the test driver for the program
//It shows that the reports classes are updating correctly
//If they are updating properly through the submission class
//We know the design pattern is implemented
```

```
public class SubmissionDriver {

	public static void main(String[] args){
		Submission sub = new Submission();
		TimeReport tReport = new TimeReport(sub);
		CorrectReport cReport = new CorrectReport(sub);
		sub.attach(tReport);
		sub.attach(cReport);

	for(int i = 0; i < 10; i++){
		sub.runTestCase();
	}

	int incorrect = 10 - cReport.getNumberCorrect() - tReport.getNumberOfTimeOutErrors();
	System.out.println("Number of Correct: " + cReport.getNumberCorrect());
	System.out.println("Number of Incorrect w/o Timeout: " +  incorrect);
	System.out.println("Number of Timeout Errors: " +
tReport.getNumberOfTimeOutErrors());
   }
}
```

Sample output:

```
Time-out Error Found
Time-out Error Found
Time-out Error Found
Time-out Error Found
Number of Correct: 4
Number of Incorrect w/o Timeout: 2
Number of Timeout Errors: 4
```

7. CODE:

```
/**
* Submission.java
*
* A representation of a Submission
* Cris Salazar
* Samuel Volin
*/

import java.util.Random;

public class Submission extends Subject
```

```
{
   private Random myRandom;
        private boolean lastErrorWasTimeout;
        private boolean lastTestPassed;

   // 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();
                lastTestPassed = passed;
                if(!passed)
                {
                   lastErrorWasTimeout = myRandom.nextBoolean();
                }

                // You can add to the end of this method for reporting purposes
                announce();
        }

   public boolean wasTimeoutError()
        {
           return lastErrorWasTimeout;
        }
   public boolean wasPassed()
        {
           return lastTestPassed;
        }
}
```

```
public class TimeReport extends Report {

        private static int numberOfTimeOutErrors;

        public TimeReport(Subject s){
                super(s);
                numberOfTimeOutErrors = 0;
        }
```

```
	public void update() {
		Submission sub = (Submission) subject;
			if(sub.wasTimeoutError()){
				System.out.println("Time-out Error Found");
				numberOfTimeOutErrors++;
			}

	}

	public int getNumberOfTimeOutErrors(){
		return numberOfTimeOutErrors;
	}
}
```

```
public class CorrectReport extends Report{

	private static int numberCorrect;

	public CorrectReport(Subject s){
		super(s);
		numberCorrect = 0;
	}

	public void update(){
		Submission sub = (Submission) subject;
		if(sub.wasPassed())
			numberCorrect++;
	}

	public int getNumberCorrect(){
		return numberCorrect;
	}
}
```

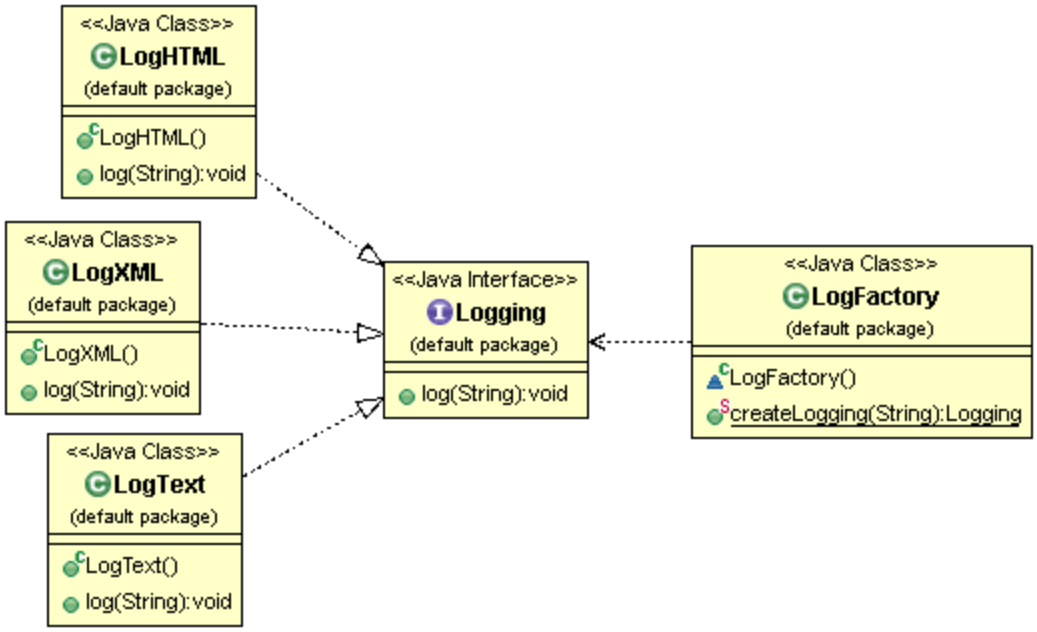
```
public class Report extends Observer
{
	public Report(Subject s) {
		super(s);
	}
}
```

```
public abstract class Observer {

	protected Subject subject;

	public Observer(Subject s)
	{
		subject = s;
	}

	public void update()
	{
		return;
	}
}
```

```
import java.util.LinkedList;

public abstract class Subject {

	private LinkedList<Observer> observers = new LinkedList<Observer>();

	public void attach(Observer o)
	{
		observers.add(o);
	}
	public void detach(Observer o)
	{
		observers.remove(o);
	}
	public void announce()
	{
		for(Observer o : observers)
		{
			o.update();
		}
	}
}
```

Problem Set 3

8. **Problem Set 3**
9. The **Factory** design pattern was used to provide the appropriate logging class in the analysis class without creating an unnecessary number of dependencies on various logging classes.



10.

11. Samuel Volin wrote the classes for this portion

12.

Work Percentage	Name	Work Done
95%	Sam	Found the smell, designed the factory, made the class diagram, did the writeup
5%	Cris	Got sam coffee

13.

```
public 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;
```

```
                // Analysis is now only dependant on LogFactory as opposed to
various
                // Log formats
                logfile = LogFactory.createLogging(type);
                logfile.log("Starting application...");
        }
}
```

14. 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);
        }
}
```

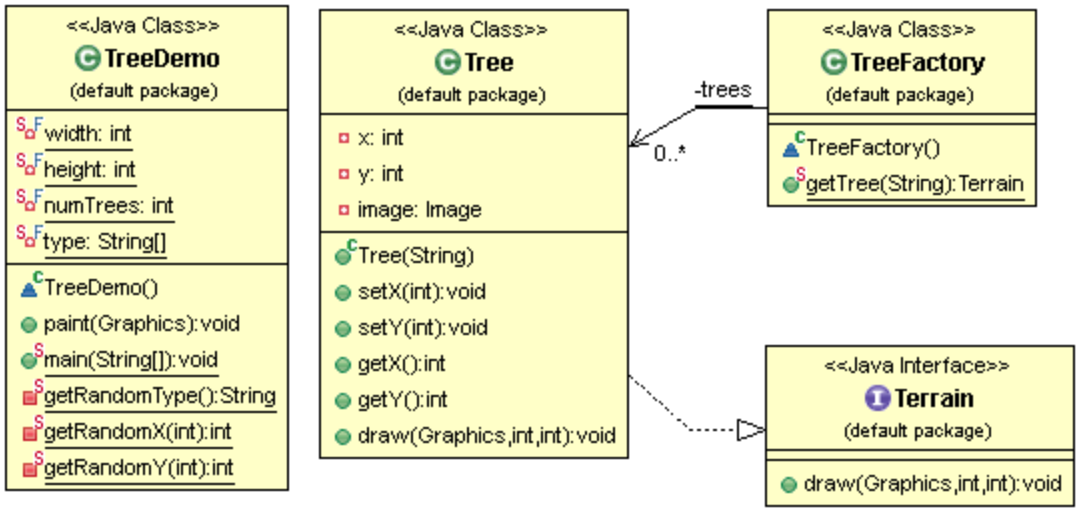
```
class LogXML implements Logging {
        public LogXML() {
                System.out.println("Logging: <type>XML Format</type>");
        }

        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 LogFactory {
        public static Logging createLogging(String type) {
                if (type.equalsIgnoreCase("XML"))
                        return new LogXML();
                else if (type.equalsIgnoreCase("HTML"))
                        return new LogHTML();
                else if (type.equalsIgnoreCase("Text"))
                        return new LogText();
                else
                        return new LogText();
        }
}
```

Problem Set 4

15. **Problem Set 4**
16. The **Flyweight** design pattern was used to make the program run much more efficiently. By storing only one instance of each type of object, the space and time complexity drop noticeably. The intrinsic state is the types of trees and the extrinsic states are the colors, shapes, and tree objects stored.
17. 

18. **Cris Salazar wrote the classes for this portion**
19.

Work Percentage	Name	Work Done
95%	Cris	Designed the flyweight design pattern, made the class diagram, did the write up
5%	Sam	Discussed performance discrepancies

20.

```
private static final int width = 800;
        private static final int height = 700;
        private static final int numTrees = 1000;
        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));
                }
        }
```

Sample output:

```
Creating a new instance of a tree of type Blob
Creating a new instance of a tree of type Maple
Creating a new instance of a tree of type Apple
Creating a new instance of a tree of type Lemon
Creating a new instance of a tree of type Elm

/*-------------------------------------------------------*/
```

I changed the number of trees created to 1000. The program runs much more quickly and only creates one instance of each type of tree. Thus the map is much smaller than if it were creating 1000 different trees.

21. CODE:

```
/**
* Submission.java
*
* A representation of a Submission
* Cris Salazar
* Samuel Volin
*/

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>();
	private static final Map<String, Tree> trees = new HashMap<String, Tree>();
	public static Terrain getTree(String type)
	{
		Tree tempTree = trees.get(type);
		if(tempTree == null){
			tempTree = new Tree(type);
			trees.put(type, tempTree);
		}
		return tempTree;
//		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 = 1000;
	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);
	}
}
```