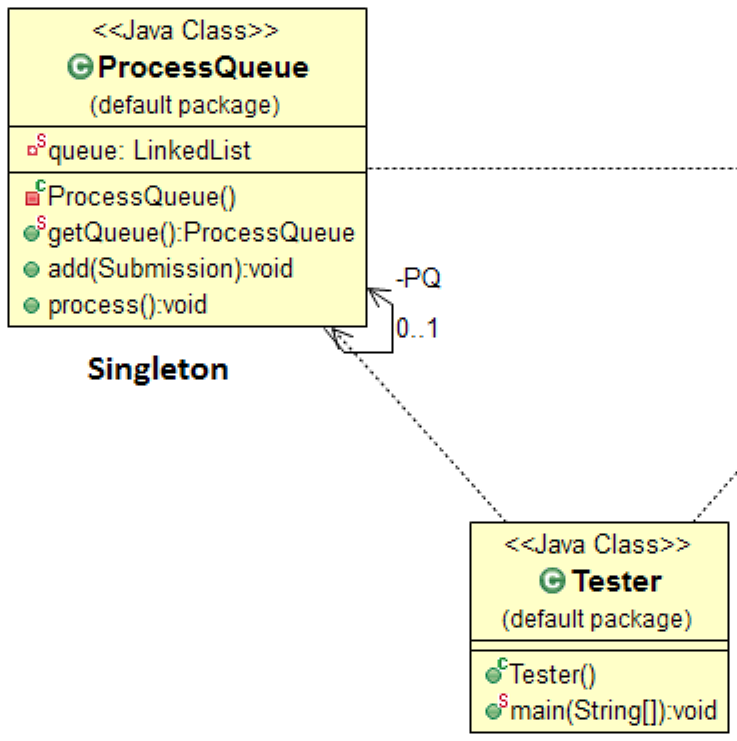
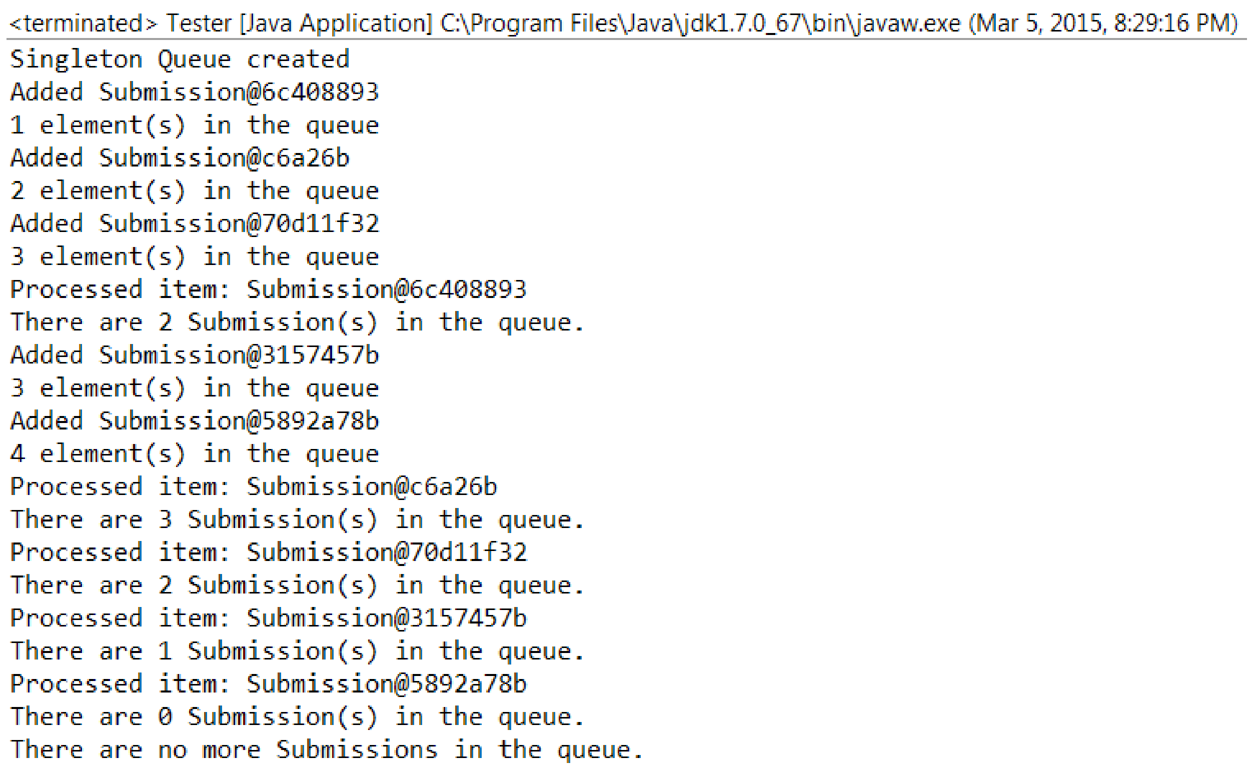
1. For problem 1, we used the Singleton design pattern. This ensured that we only ever have one instance of the ProcessQueue, eliminating scheduling conflicts that could arise from asynchronous entries into multiple queues. Thus, we had each Grader object use the same static queue.

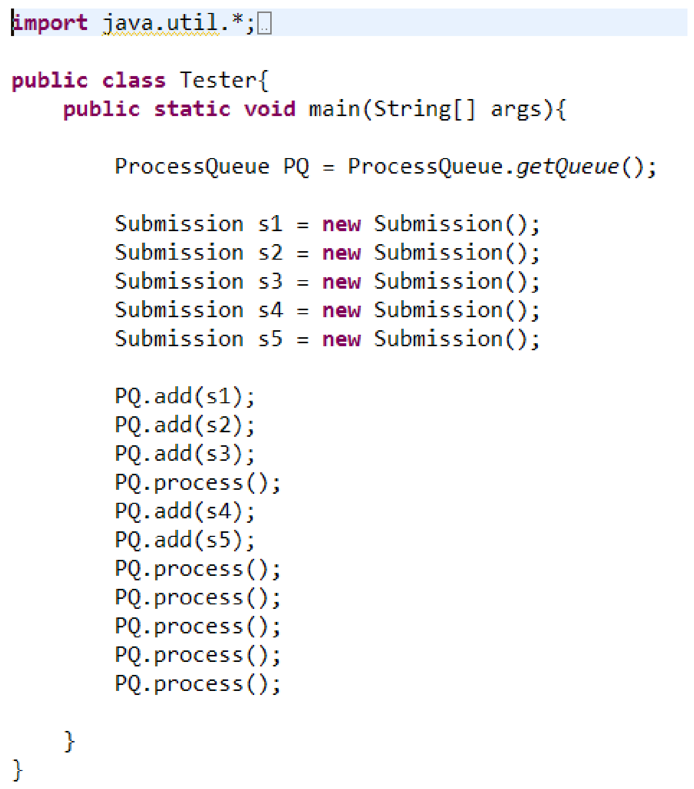


100%	Brian	Implemented singleton
0%	Luke	N/A



```
<terminated> Tester [Java Application] C:\Program Files\Java\jdk1.7.0_67\bin\javaw.exe (Mar 5, 2015, 8:29:16 PM)
Singleton Queue created
Added Submission@6c408893
1 element(s) in the queue
Added Submission@c6a26b
2 element(s) in the queue
Added Submission@70d11f32
3 element(s) in the queue
Processed item: Submission@6c408893
There are 2 Submission(s) in the queue.
Added Submission@3157457b
3 element(s) in the queue
Added Submission@5892a78b
4 element(s) in the queue
Processed item: Submission@c6a26b
There are 3 Submission(s) in the queue.
Processed item: Submission@70d11f32
There are 2 Submission(s) in the queue.
Processed item: Submission@3157457b
There are 1 Submission(s) in the queue.
Processed item: Submission@5892a78b
There are 0 Submission(s) in the queue.
There are no more Submissions in the queue.
```

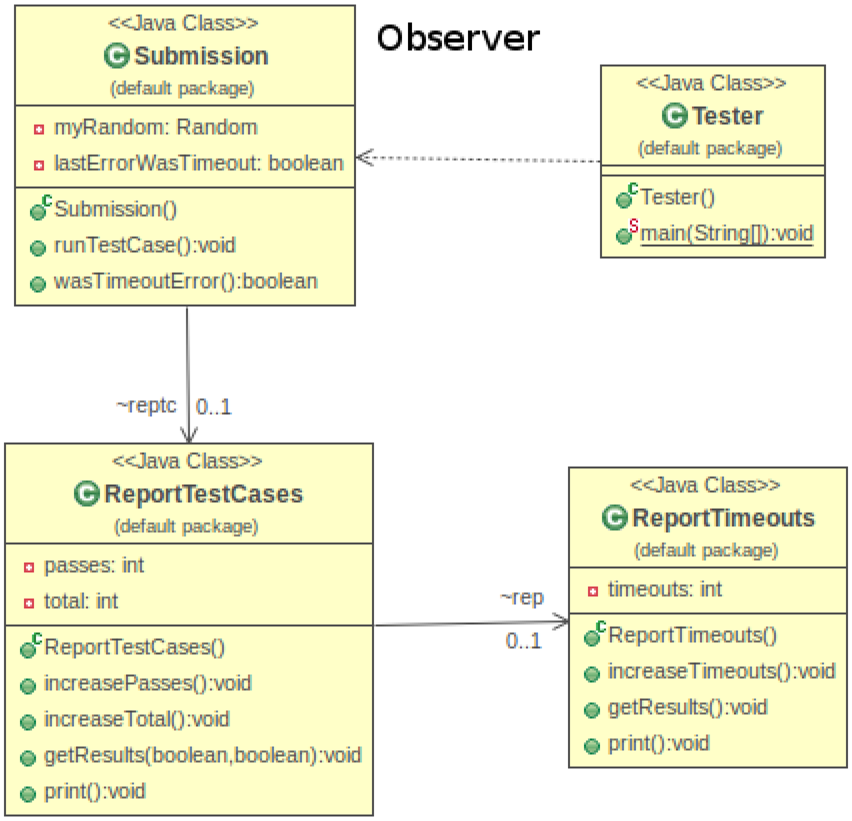
```java
//Brian Gaydon

import java.util.*;

public class ProcessQueue {

    //singleton queue and ProcessQueue object
    private static ProcessQueue PQ;
    private static LinkedList queue;

    private ProcessQueue() {
        queue = new LinkedList();
    }

    public static ProcessQueue getQueue() {
        if(queue == null) {
            PQ = new ProcessQueue();
            System.out.println("Singleton Queue created");
        }
        return PQ;
    }

    public void add(Submission s) {
        queue.addFirst(s);
        System.out.println("Added " + queue.getFirst());
        System.out.println(queue.size() + " Submission(s) in the queue");
    }

    public void process() {
        //This is where actual processing code would go

        if (!queue.isEmpty()) {
            System.out.println("Processed item: " + queue.getLast());
            queue.removeLast();
            System.out.println("There are " + queue.size() + " Submission(s) in the queue.");
        }
        else System.out.println("There are no more Submissions in the queue.");
    }
}
```



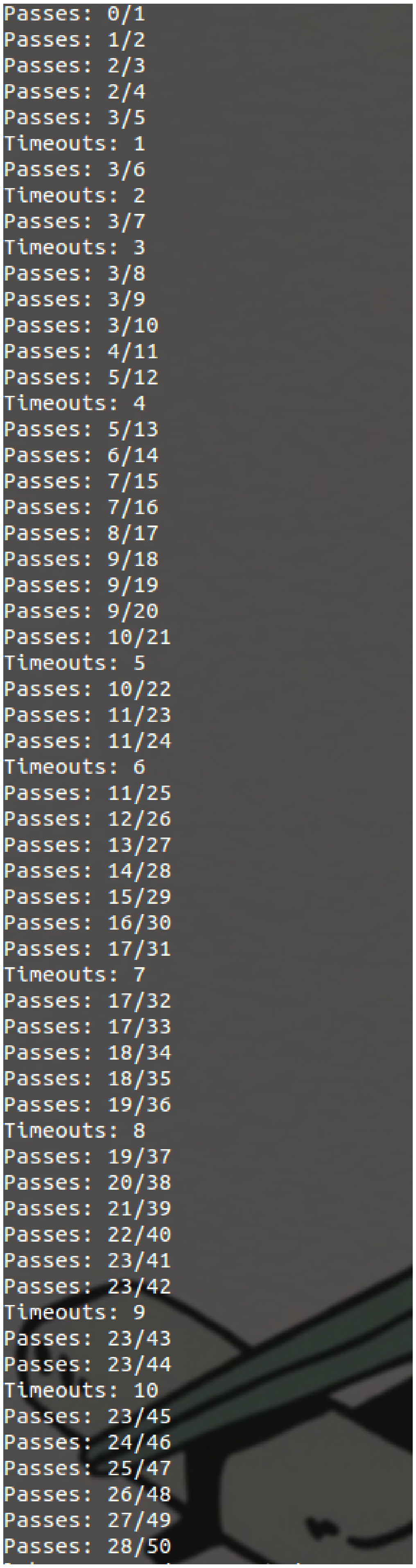
```java
import java.util.*;

public class Tester{
    public static void main(String[] args){

        ProcessQueue PQ = ProcessQueue.getQueue();

        Submission s1 = new Submission();
        Submission s2 = new Submission();
        Submission s3 = new Submission();
        Submission s4 = new Submission();
        Submission s5 = new Submission();

        PQ.add(s1);
        PQ.add(s2);
        PQ.add(s3);
        PQ.process();
        PQ.add(s4);
        PQ.add(s5);
        PQ.process();
        PQ.process();
        PQ.process();
        PQ.process();
        PQ.process();

    }
}
```

2. For problem 2, we used the Observer design pattern. The pattern allows the submission to delegate the reporting of test cases as well as the timing out of the tests to objects which will report back (print to console) when they have finished.

Observer

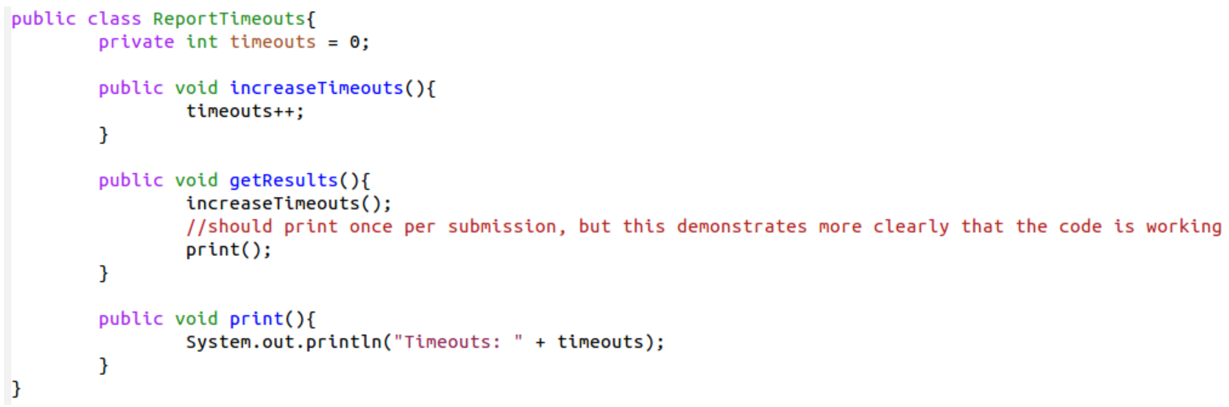


100%	Luke	ReportTestCases and Timeouts
0%	Bryan	N/A

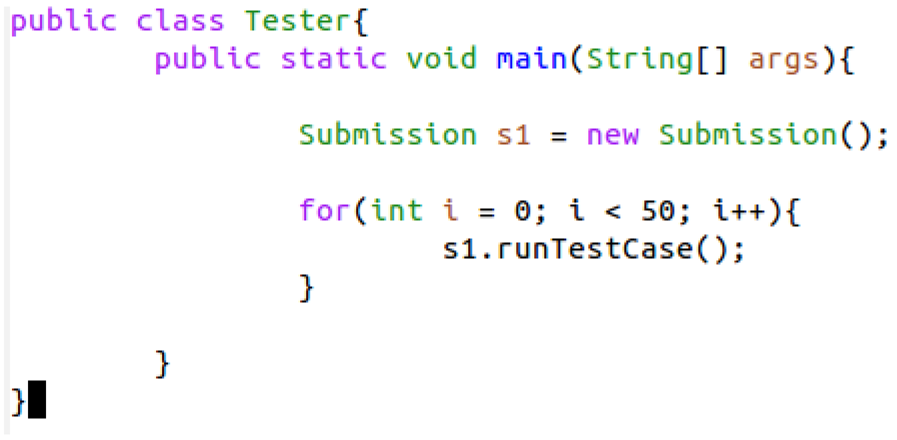


```
Passes: 0/1
Passes: 1/2
Passes: 2/3
Passes: 2/4
Passes: 3/5
Timeouts: 1
Passes: 3/6
Timeouts: 2
Passes: 3/7
Timeouts: 3
Passes: 3/8
Passes: 3/9
Passes: 3/10
Passes: 4/11
Passes: 5/12
Timeouts: 4
Passes: 5/13
Passes: 6/14
Passes: 7/15
Passes: 7/16
Passes: 8/17
Passes: 9/18
Passes: 9/19
Passes: 9/20
Passes: 10/21
Timeouts: 5
Passes: 10/22
Passes: 11/23
Passes: 11/24
Timeouts: 6
Passes: 11/25
Passes: 12/26
Passes: 13/27
Passes: 14/28
Passes: 15/29
Passes: 16/30
Passes: 17/31
Timeouts: 7
Passes: 17/32
Passes: 17/33
Passes: 18/34
Passes: 18/35
Passes: 19/36
Timeouts: 8
Passes: 19/37
Passes: 20/38
Passes: 21/39
Passes: 22/40
Passes: 23/41
Passes: 23/42
Timeouts: 9
Passes: 23/43
Passes: 23/44
Timeouts: 10
Passes: 23/45
Passes: 24/46
Passes: 25/47
Passes: 26/48
Passes: 27/49
Passes: 28/50
```

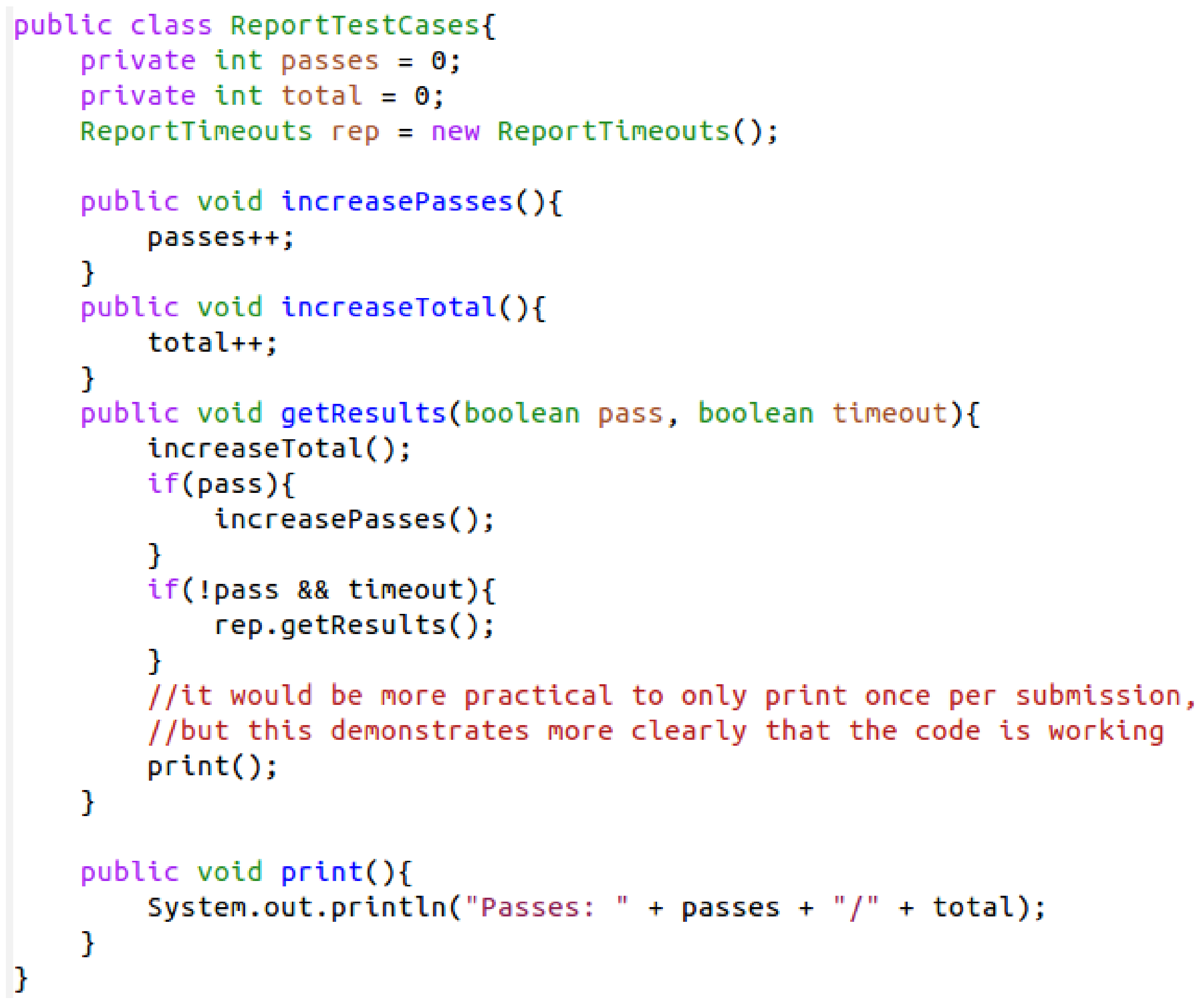
```java
import java.util.Random;

public class Submission
{
    private Random myRandom;
    private boolean lastErrorWasTimeout;

    // You may add attributes to this class if necessary
    ReportTestCases repTests =  new ReportTestCases();
    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
        repTests.getResults(passed, wasTimeoutError());
    }

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



```java
public class ReportTimeouts{
        private int timeouts = 0;

        public void increaseTimeouts(){
                timeouts++;
        }

        public void getResults(){
                increaseTimeouts();
                //should print once per submission, but this demonstrates more clearly that the code is working
                print();
        }

        public void print(){
                System.out.println("Timeouts: " + timeouts);
        }
}
```



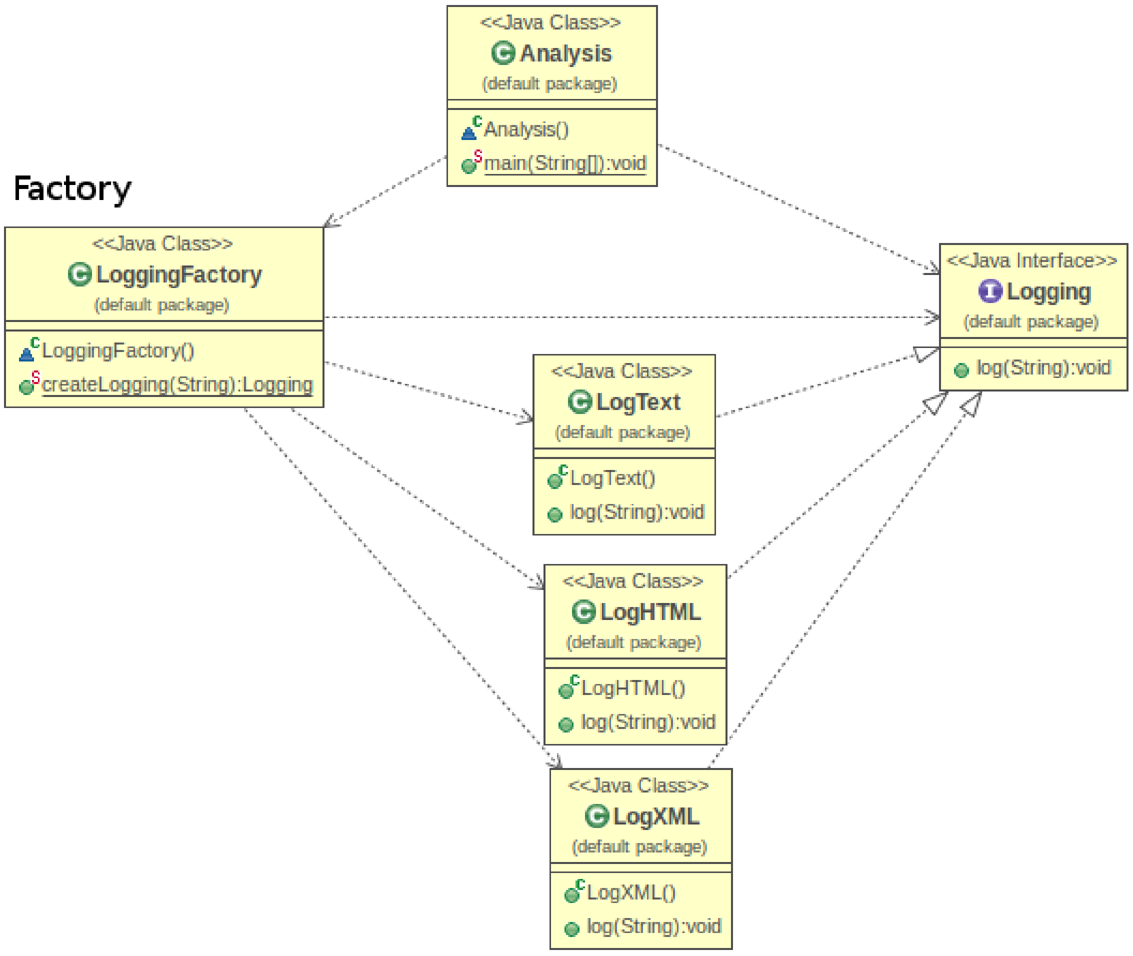
```java
public class Tester{
        public static void main(String[] args){

                Submission s1 = new Submission();

                for(int i = 0; i < 50; i++){
                        s1.runTestCase();
                }

        }
}
```



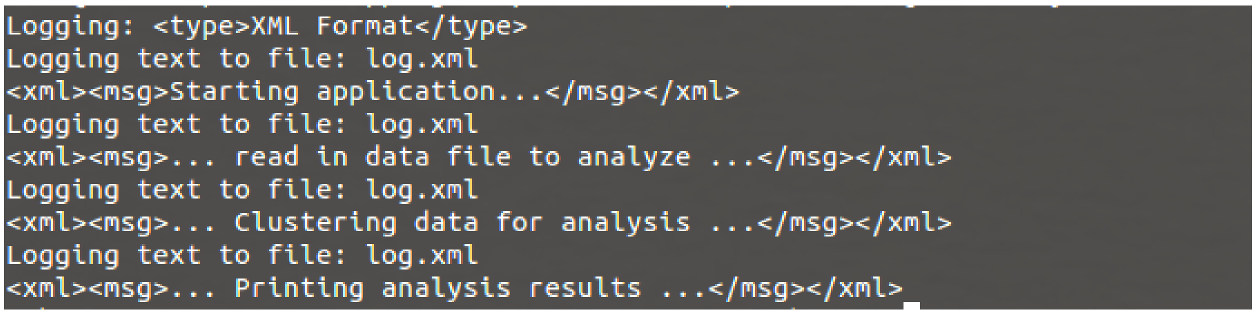
```java
public class ReportTestCases{
    private int passes = 0;
    private int total = 0;
    ReportTimeouts rep = new ReportTimeouts();

    public void increasePasses(){
        passes++;
    }
    public void increaseTotal(){
        total++;
    }
    public void getResults(boolean pass, boolean timeout){
        increaseTotal();
        if(pass){
            increasePasses();
        }
        if(!pass && timeout){
            rep.getResults();
        }
        //it would be more practical to only print once per submission,
        //but this demonstrates more clearly that the code is working
        print();
    }

    public void print(){
        System.out.println("Passes: " + passes + "/" + total);
    }
}
```

3. For problem 3, we used the factory design pattern. This allowed us to create specific instances of loggers easily. This way one can create a logger and pass it a type and get an appropriate logger for the time of text they are seeking to log

Factory



100%	Luke	Logging factory
0%	Bryan	N/A

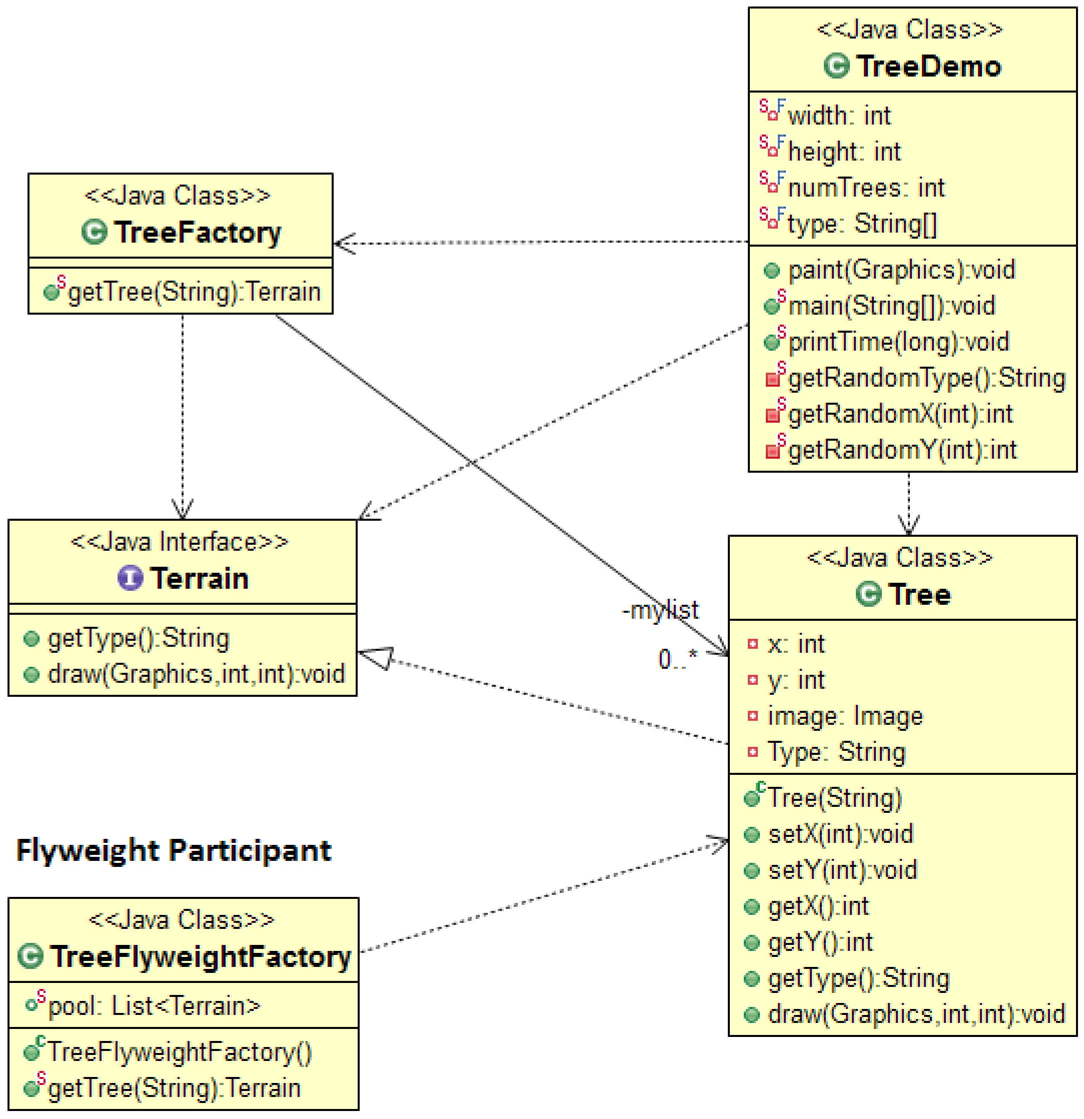


```
Logging: <type>XML Format</type>
Logging text to file: log.xml
<xml><msg>Starting application...</msg></xml>
Logging text to file: log.xml
<xml><msg>... read in data file to analyze ...</msg></xml>
Logging text to file: log.xml
<xml><msg>... Clustering data for analysis ...</msg></xml>
Logging text to file: log.xml
<xml><msg>... Printing analysis results ...</msg></xml>
```

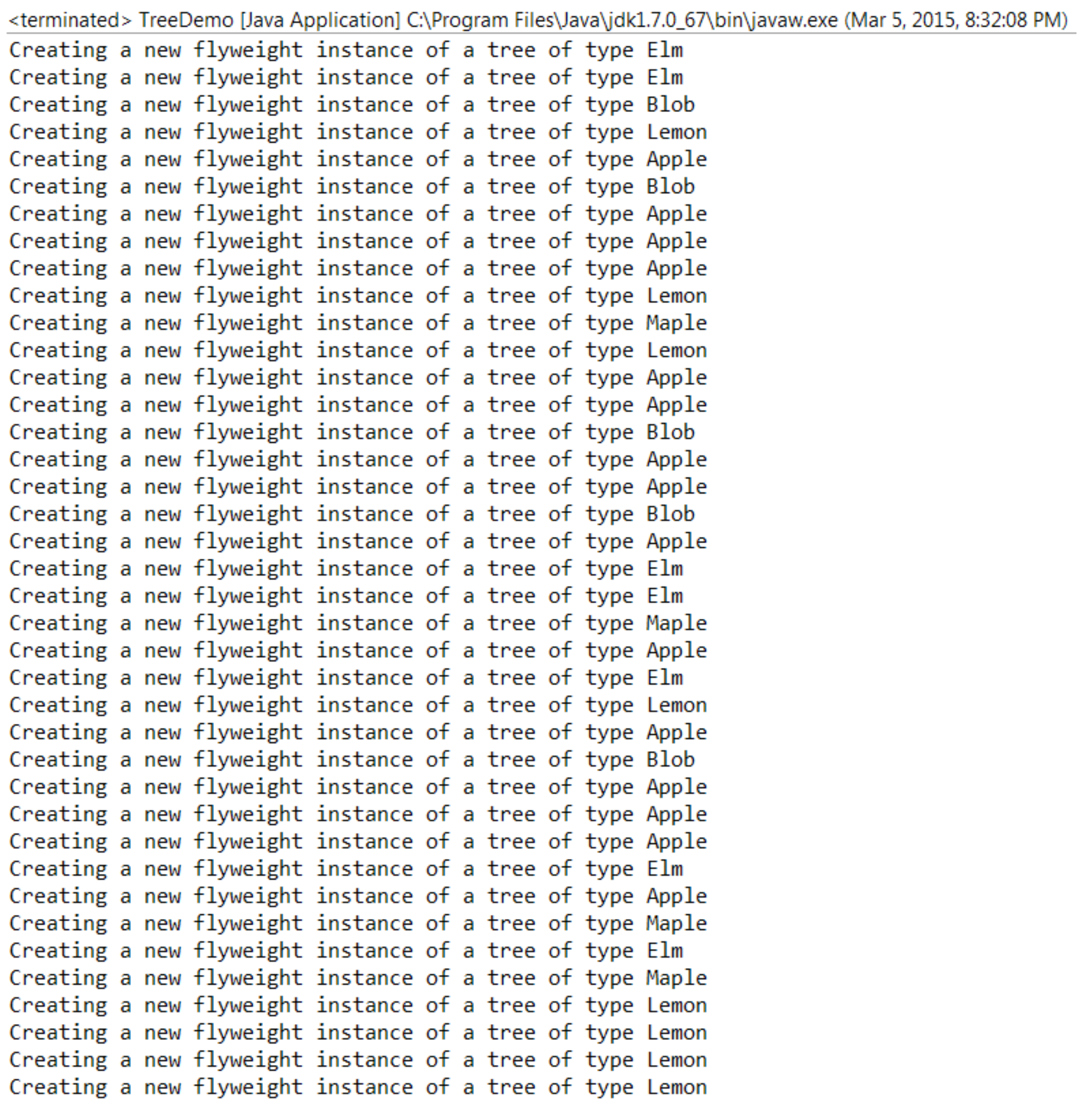
```java
class Analysis
{
        public static void main(String[] args)
        {
                if (args.length != 1)
                {
                        System.out.println("Usage: java Analysis type");
                        System.exit(-1);
                }
                String type = args[0];
                Logging logfile = LoggingFactory.createLogging(type);
                logfile.log("Starting application...");

                logfile.log("... read in data file to analyze ...");
                // code...
                logfile.log("... Clustering data for analysis ...");
                // code...
                logfile.log("... Printing analysis results ...");
                // code...
        }
}
```

```java
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 LoggingFactory
{

        public static Logging createLogging(String type){
                if (type.equalsIgnoreCase("xml"))
                        return new LogXML();
                else if (type.equalsIgnoreCase("html"))
                        return new LogHTML();
                else
                        return new LogText();
        }

}
```

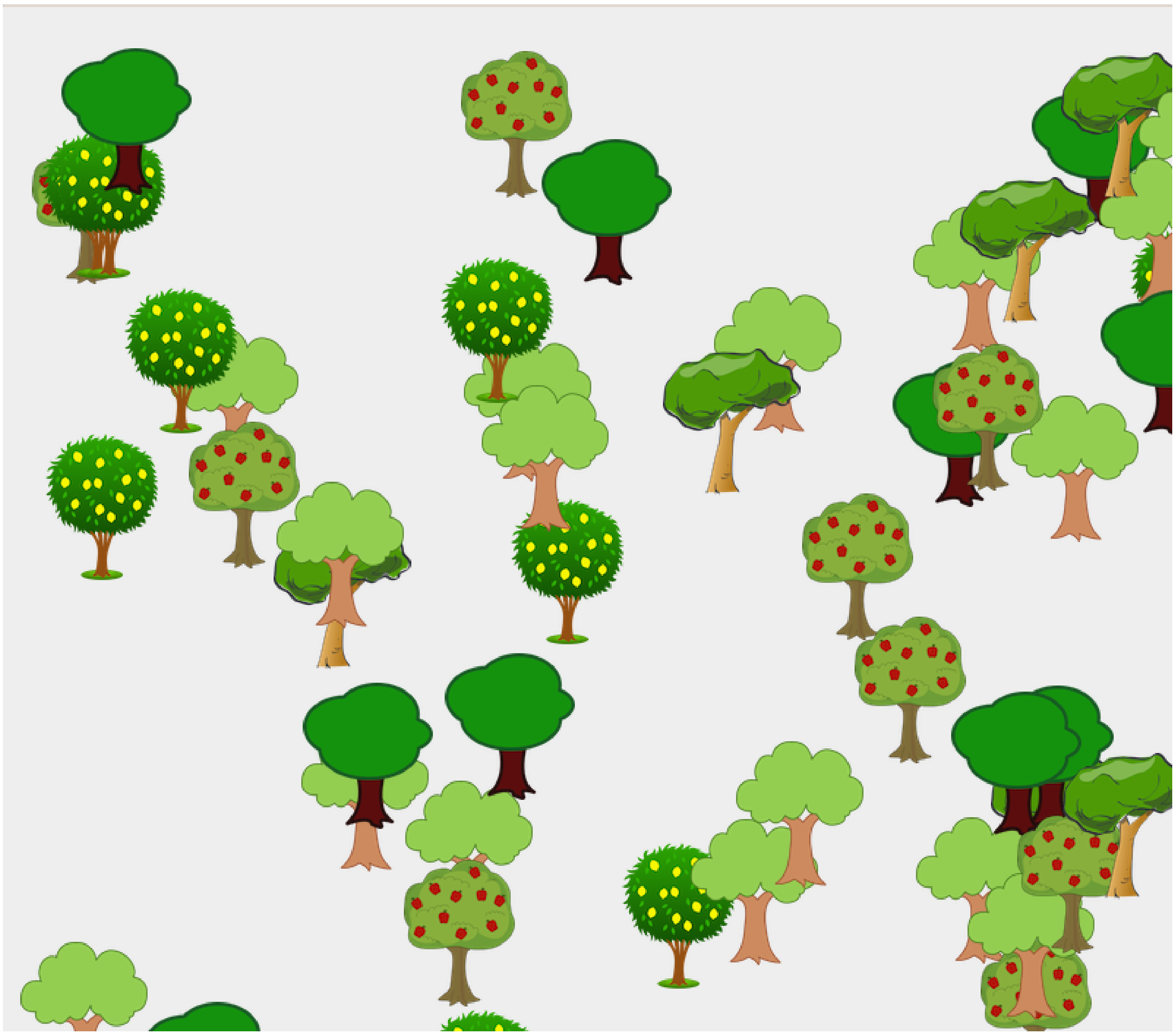
4. For problem 4, we used the Flyweight design pattern. This pattern allowed us to quickly and efficiently create trees that reused the same data for each instance of the same type of tree. The original implementation required the same trees to be loaded many times and took forever; our method reduced the runtime of the draw operation to a fraction of the original.



100%	Brian	Implemented flyweight
0%	Luke	N/A



```
<terminated> TreeDemo [Java Application] C:\Program Files\Java\jdk1.7.0_67\bin\javaw.exe (Mar 5, 2015, 8:32:08 PM)
Creating a new flyweight instance of a tree of type Elm
Creating a new flyweight instance of a tree of type Elm
Creating a new flyweight instance of a tree of type Blob
Creating a new flyweight instance of a tree of type Lemon
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Blob
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Lemon
Creating a new flyweight instance of a tree of type Maple
Creating a new flyweight instance of a tree of type Lemon
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Blob
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Blob
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Elm
Creating a new flyweight instance of a tree of type Elm
Creating a new flyweight instance of a tree of type Maple
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Elm
Creating a new flyweight instance of a tree of type Lemon
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Blob
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Elm
Creating a new flyweight instance of a tree of type Apple
Creating a new flyweight instance of a tree of type Maple
Creating a new flyweight instance of a tree of type Elm
Creating a new flyweight instance of a tree of type Maple
Creating a new flyweight instance of a tree of type Lemon
Creating a new flyweight instance of a tree of type Lemon
Creating a new flyweight instance of a tree of type Lemon
Creating a new flyweight instance of a tree of type Lemon
```



```java
import java.util.*;

interface Terrain
{
    String getType();
    void draw(Graphics graphics, int x, int y);
}

//Flyweight factory
class TreeFlyweightFactory {
    //instantiation needed?
    public static List<Terrain> treePool = new ArrayList<Terrain>();

    public TreeFlyweightFactory() { treePool = new ArrayList<Terrain>(); }

    public static Terrain getTree(String type) {
        for(Terrain tree: treePool) {
            if (tree.getType().equals(type)) {
                return tree;
            }
        }
        System.out.println("Creating new " + type + "tree");
        Terrain newTree = new Tree(type);
        treePool.add(newTree);
        return newTree;
    }
}

class Tree implements Terrain
{
    private int x;
    private int y;
    private Image image;
    private String Type;
    public Tree(String type)
    {
        System.out.println("Creating a new flyweight 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; }
     public String getType() { return Type; }
     @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;
    }
}
```