HW Refactoring

You can work individually or in groups of 2.

Background

You have been provided code for a system to automatically grade student homework submissions written in Java. The system itself consists of four classes:

Class	High-level Description
TestSuite	Contains static methods that test a set of classes (student solution). From a high-level, they are similar to JUnit tests. You have been provided with a *TestSuite* that tests a homework from CS 1302 that utilizes *Martian* classes (which are also provided, but are unimportant for this assignment). You will not modify this class.
TestEngine	Contains *main* and the code to run the test methods in *TestSuite*, grade results, and build a report. You will not modify this class, except possibly to change a Boolean flag from *true* to *false* and back as explained shortly.
Test	Represents an individual test that is run on the student solution. Contains fields: test num, description, points earned, *etc.* You will refactor the *assess* method in this class (explained shortly).
GradeReport	Contains a collection of *Tests* and summary information about the test results. You will not modify this class.

To use, run the system against the instructor's solution which generates an expected results file. Then run against student's solution whose results are compared to the expected results.

Detailed Description

1. *TestEngine* has a flag:

    ```
    final boolean shouldGenerateExpectedResults = true;
    ```

    When the flag is *true*, the system assumes that it is being run against the instructor's (correct) solution. The code runs all the tests in *TestSuite*, collects the results, and saves these expected results in a file: *expectedResults.txt*.

    When the flag is *false*, the system assumes that the student solution is being tested. It then runs the tests, collects the results, and compares them to the expected results producing a report similar to the one below. Note that we show the results for only a few of the 16 tests. Note also that each test can have multiple parts.

    ```
    Test 1-Martian.equals()
    Correct - Expected: r3.equals(g1)= false | Actual: r3.equals(g1)= false
    Correct - Expected: r3.equals(g4)= true | Actual: r3.equals(g4)= true
    Summary: 2 out of 2 answers correct : 5.0 points out of 5.0

    Test 2-Martian.compareTo()
    Correct - Expected: r3.compareTo(g1)= -9 | Actual: r3.compareTo(g1)= -9
    Correct - Expected: r3.compareTo(g4)= 0 | Actual: r3.compareTo(g4)= 0
    Correct - Expected: r2.compareTo(g3)= 8 | Actual: r2.compareTo(g3)= 8
    Summary: 3 out of 3 answers correct : 5.0 points out of 5.0

    ...

    Test 15-MartianManager.Teleport("Orck")
    Correct - Expected: countsIds=[0, 0, 1, 1] | Actual: countsIds=[0, 0, 1, 1]
    ```

```
Correct - Expected: count Orcks=2 | Actual: count Orcks=2
Summary: 2 out of 2 answers correct : 5.0 points out of 5.0

...

Overall Summary: 80.0 points out of 80.0 (100.0%)
```

2. *TestEngine* contains a path variable:

   ```
   final String PATH = "src//hw1a//";
   ```

   The *hw1a* reference is the folder/package where all the classes are, including the *Martian* classes. This should be correct, unless you change the folder/package.

3. I recommend you run do the following:

   a. Open *TestEngine,* set the flag to *true* and run.
   b. It will generate *expectedResults.txt.* Open this file and scan it quickly. It it is a bit cryptic. This file is read and parsed back into memory when running against the student solution.
   c. Set the flag back to *false* and run again. The results will be shown in the console and also saved in *studentReport.txt.* Of course, you are running against the same same code that generated the expected results, so everything will be correct.
   d. For this assignment, it is not necessary to have a student solution. However, if you want to see it detect errors, open *MartianManager* and comment out the lines shown below and add a return of *null.*

   ```
               public Martian getMartianAt(int i) {
       //          if( (i<0) || (i>=martians.size()))
       //                  return null;
       //          return martians.get(i);
                   return null;
               }
   ```

   e. Run *TestEngine* (flag is still *false*). You should see that the first answer in Test 5 is incorrect:

   ```
   Test 5-MartianManager.getMartianAt()
   Incorrect - Expected: mm.getMartianAt(2).getId()=2 | Actual: java.lang.NullPointerException
   Correct - Expected: mm.getMartianAt(99)=null | Actual: mm.getMartianAt(99)=null
   Summary: 1 out of 2 answers correct : 2.5 points out of 5.0
   ```

   f. Change *getMartianAt* so that it is correct, rerun, and verify that both answers in Test 5 are now correct.

4. Doing this assignment, you will generally run the program with the flag set to *false*

   ```
   final boolean shouldGenerateExpectedResults = false;
   ```

   However, if you generate a run-time error that you feel is not in code you modified, try the following immediately: set the flag to *true,* run (hopefully works), then set to *false* again.

5. You will refactor the *assess* method in the *Test* class. This method “grades” an individual test. Remember that each test can have multiple parts (answers). The code uses two lists, *expectedOutput* (expected results) and *actualOutput* (student results). Each list contains a number of strings where each string is an “answer”. For example, if the test had 3 parts, then there would be three strings in each list. There are two types of answers:

    a. String answer – For string answers, each element in *expectedOutput* is directly compared using string *equals* to the corresponding value in *actualOutput* to determine correctness or not. For example, in Test 1 above, the two highlighted strings are compared:

    ```
    Test 1-Martian.equals()
    Correct - Expected: r3.equals(g1)= false | Actual: r3.equals(g1)= false
    ```

    b. String answer that contains a double – We must (usually) provide some flexibility with answers that contain a double value. For example, if the expected answer is: “Total balance=550.27” we may not want to penalize a student’s answer that is close. For example, we may want to consider: “Total balance=550.25” to be correct. The technique we use to handle doubles is considered next.

        i. An answer with a double will have one of two sets of embedded flags:

            1. “%d … %tp …” or
            2. “%d … %ta …”

        ii. For example:

        ```
        "%d 8.834 %tp 2.0 The average num rebounds is=8.834"
        ```

        Let’s dissect the string above:

Token	**Meaning**
%d	Flag denoting that the answer contains a double
8.834	Double value that is found in the answer
%tp	Flag denoting that the tolerance is specified with a percentage
2.0	Tolerance percentage
The average num rebounds is=8.834	Expected answer

        The "%tp 2.0" means the tolerance is 2%. Thus, an actual result of $8.834 \pm 8.834 * 0.02$ is considered correct. Tolerance can also be specified absolutely. For example:

        ```
        "%d 8.834 %ta 0.01 The average num rebounds is=8.834"
        ```

        Which means that an actual result of 8.834 ± 0.01 is considered correct.

        Example output for an answer with an embedded double:

        ```
        Test 16-MartianManager.getAverageId()
        Correct - Expected: mm.getAverageId()=6.5 | Actual: mm.getAverageId()=6.5
        Actual Error=0.0<0.065=Max Error
        Summary: 1 out of 1 answers correct : 5.0 points out of 5.0
        ```

Requirements

1. You will refactor the *assess* method in the *Test* class. The method is long and heavily commented – two code smells!

   Hints:

   a. Readability is the goal you are after.
   b. The initialize code that is in the method is a form of *feature envy.* In other words, why are instance variables being initialized in a method?!?
   c. As noted in the video, you can highlight code, choose: Refactor, and then choose the proper refactoring technique. I believe they are all "extract method". Or, you can just do it manually.
   d. Don't just make *assess* readable, make the extracted methods readable too. In other words, you will need to refactor some (all?) extracted methods.

Deliverables

1. Code – zip the *hw1a* package into a file named: hw_refac_LastName1_LastName2.zip and submit on Blazeview in the dropbox named, *HW Refac*. Only one person should submit.