1. PSP Time Recording Log

You must set up your PSP **Time Recording Log** before starting work on the exercise (see the instructions in the "PSP Documents" section of Week 4 on Moodle). When you have finished the exercise, go back and calculate your productivity rate in Lines of Code per Hour (LOC/Hour).

If you are running your own Eclipse installation, you can install the Metrics plugin from http://metrics2.sourceforge.net/ that can automate LOC calculations for you.

2. Tutorial Exercise 3

Building on your work from last week, your task this week is to extend and improve the Java University system.

NOTE: Make sure you begin by recording in your Start Time in your Time Recording Log. Remember to keep track of any interrupt time that occurs before you finish.

2.1. Requirements

IMPORTANT: Before you type any code, you must draw a UML class diagram of the final modified system (after all parts are completed) and show it to your tutor.

Part A: Redesign of `Unit`

The designers of the `Unit` class have realised that it has a problem. New methods are expected in future that need to be able to get quick access to a student enrolled in a unit without having to search through an `ArrayList` each time, and there is never a need to access the students in the order in which they were enrolled in the unit. They have decided to replace the students `ArrayList` with a `HashMap` called `enrolledStudents`.

You are to refactor the `Unit` class as described below. You must not change any classes other than `Unit` and `Student`.

- Remove the old `students` `ArrayList` attribute.
- Create an attribute called `enrolledStudents` that is a `HashMap` that stores references to `Student` objects and uses the `studentID` as the key.
- Modify the `enrolStudent(…)` method to use this new data structure.
- Change the method you wrote last week to return an `ArrayList` of the students enrolled in the unit so that it still has the same signature, but gets its data from the `enrolledStudents` `HashMap`. This will allow any existing clients of the `Unit` class that make use of that method (e.g. `University`) to continue to work without change.

Test your system to make sure that everything still works. This is an example of *regression testing*. Demonstrate this to your tutor before proceeding.

- Add a function `isEnrolled(…)` that takes a `Student` as its input parameter and returns a `boolean` indicating whether or not the student is enrolled in the unit.
- Add a routine called `unenrolStudent(…)` that takes a `Student` as its input parameter and removes that student from `enrolledStudents`.
- Add tests to the `University` class to ensure that these two new features work as required.

Part B: Command Line I/O

In your major assignment you will be working on an interactive text application. You will need to read data that the user types on the command line. There is no simple single built-in method to read strings from the command line in Java, so here is an example of a method you can use:

```
private String readString() {

    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String s = null;
        try {
            s = in.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return s;
}
```

Modify `University` so that when each `Student` object is created, the user is prompted to enter the student's first name, last name, and student ID. Read these data from the command line and use them when creating the `Student` object.

Part C: Assessments

There are two types of assessment at Java University: exams and assignments. Every piece of assessment has an associated weight, which must be an integer between 1 and 100. Exams have a duration, which is an integer between 30 and 180 representing the duration of the exam in minutes. Assignments have a title, which is a string giving the assignment name.

You must implement these requirements using inheritance. You need to:

- Create a *abstract* class called `Assessment` that has
  - An `int` attribute `weight`, and getter and setter methods for this attribute.
  - An abstract function `description()` that returns a `String`
- Create a class `Exam` that inherits from `Assessment`. It needs to have:
  - A constructor that takes a weight and a duration as parameters, and sets the corresponding attributes.
  - An `int` attribute `duration`
  - An implementation of `description()` that returns a String such as: "Exam: duration 180 minutes, weight 60%"
- Create an `Assignment` class that inherits from `Assessment`. It needs to have:
  - A constructor that takes a weight and a title as parameters, and sets the corresponding attributes.
  - A `title` attribute
  - An implementation of `description()` that returns a string containing the title and weight, such as: “Assignment: Team Software Project, weight 40%”

The constructors must have executable preconditions that throw exceptions if the rules for valid weights and durations mentioned above are violated (see Appendix).

Part D: Assessment Schemes

We now need to add further functionality to our `Unit` class. Every `Unit` needs to have *assessment scheme* associated with it. An assessment scheme is a collection of assessments. The sum of the weights of the assessments in an assessment scheme must always be equal to 100. An `AssessmentScheme` object must be passed the `Assessment` objects that it consists of when it is created. Design an `AssessmentScheme` class that meets these requirements, and uses assertions and exceptions to check for correctness, and report violations. Modify `Unit` so that the assessment scheme for the unit can be set. Modify `University` to test your new code.

Appendix

Design By Contract (DBC) is an approach to design and coding that bridges the gap between textual specifications for classes and their methods, and the code itself. Some languages (e.g. Eiffel) support DBC natively. In others, some or all of the principles of DBC can be implemented by hand using constructs such as assertions and exceptions.

In Java, we can take a first step by getting methods to check their preconditions, and to throw exceptions if they are violated. We can also use the Java `assert()` statement to specify postconditions. Consider the following two classes:

```java
public class Driver {

    public static void main(String[] args) throws Exception {

        DBCDemo demo = new DBCDemo();
        double arg1 = 2;
        double arg2 = 8;
        System.out.println(
            demo.geometricMean(arg1, arg2)
        );
    }

}
```

Note that this `main(…)` method is declared to throw an `Exception`. This is necessary because it calls the `geometricMean(…)` method of `DBCDemo` that itself can throw an `Exception`, and nothing is done to handle it in `main(…)`.

```java
public class DBCDemo {

     public double geometricMean(double a, double b) throws Exception {
          /*
           * Start Preconditions
           */
          // Precondition: firstArgumentNonNegative
          if (a < 0) {
               throw new Exception("Precondition violated: 
firstArgumentNonNegative");
          }
          // Precondition: secondArgumentNonNegative
          if (b < 0) {
               throw new Exception("Precondition violated: 
secondArgumentNonNegative");
          }
          /*
           * End Preconditions
           */

          double result = Math.sqrt(a*b);
          //double result = (a + b)/2; // WRONG This is the arithmetic mean

          /*
           * Start Postconditions
           */
          // Postcondition: invertingGeoMeanAccurate
          // Note that we should always avoid checking for equality between 
floating point values, due to finite machine precision
          assert(Math.abs(result*result - a*b) < 1e-20) : "Postcondition 
violated: invertingGeoMeanAccurate";
          /*
           * End Postconditions
           */
          return result;
     }

}
```

The method `geometricMean(…)` has two preconditions: both its arguments must be non-negative, as we it needs to compute the square root of their product. It also has a post-condition that checks that the calculation of the geometric mean is correct.

Try this code out. See what happens if you change one or both of the arguments in `Driver` to be negative. See what happens if you comment out the correct geometric mean calculation and uncomment the line below it.

NB. You need to enable assertion checking for your project by passing the argument “-ea” to the virtual machine (VM). You can do this under “Run | Run Configurations | Arguments” in Eclipse. Alternatively, you could use explict exception-throwing for post-conditions too.