
JML UNIT TESTING

1 Contents

In section 2, we discuss what we are trying to do in this lab.

In section 3, we discuss JML and JML Unit Testing.

In section 4, we give instructions to try this out on a few simple examples.

In section 5, we give a program and ask students to try out JML Unit Testing on this program.

2 What are we trying to do?

WHAT ARE WE TRYING TO DO?

Our goal is to use JML specifications to serve as an oracle to help us in creating unit tests.

HOW IS THIS DIFFERENT FROM WHAT WE HAVE BEEN DOING?

We do not have to write test code. We do have to provide input data and we do have to first create JML specifications for each of the methods we want to test.

HOW DOES IT WORK?

It converts the specifications into oracles and uses them to check if the tests passed or failed. It generates test cases from the test data supplied by you.

3 About JML and JML-Unit Testing

3.1 JML

JML or Java Modeling Language allows one to specify the pre-conditions and post-conditions of each method and also to specify invariants that must hold. A variety of tools have been built which make use of these specifications. In our case, we use the JML-JUnit testing tool, which makes use of the specifications to create the oracle for testing. Let us look at an example of JML specifications:

```
//@ requires currentFloor < MAX_FLOOR;  
//@ assignable currentFloor, floorButtonsLit;  
//@ ensures currentFloor == newFloor;  
//@ ensures floorButtonsLit.equals(  
//@     \old(floorButtonsLit.remove(new JMLInteger(newFloor))));  
//@  
public abstract void moveUp();
```

In the above example, the “//@” are special comments that indicate that these are JML specifications. “requires” means that these are pre-conditions. “ensures” means

that these are post-conditions”. “assignable” indicates which variables can be modified (by the programmer). “\old” is used to indicate the value of the variable at the start of the program. The syntax of these specifications use Java syntax as much as possible.

There are complexities such as model variables, hierarchy of specifications, and a host of others – and we will ignore them for the purpose of this lab.

3.2 JML UNIT TESTING

To use JML Unit Testing, there are seven steps to follow:

- 1) First, write specifications for methods of interest (i.e. write pre and post conditions using JML syntax).
- 2) Make sure that your Java code AND `jml-release.jar` are in your classpath.
`setenv CLASSPATH “./opt/JML/bin/jml-release.jar”`
- 3) Use `jmlc` command – this is JML compiler. This compiles the Java program with the JML specifications and checks for syntactic errors.
- 4) Use `jmlunit` command – this is the JMLUnit test code generator. This generates `_JML_TestData.java` and `_JML_Test.java` files (and creates code for all necessary oracles). The first one is used to generate test data. The second one is a test driver.
- 5) At this point, you can edit the `_JML_TestData.java` files to add your own test data.
- 6) Use `javac` to compile the test driver and test data code.
- 7) Use `jmlrac` command – this runs your tests.

4 Practice using JMLUnit testing

- 1) The files you will need for this lab can be found at: www.cs.iastate.edu/~smitra/lab08.zip. It also has papers on JML and JUnit Testing. You will need to unzip `hw.zip` which is in the `lab08` folder.
- 2) ssh to `pyrite.cs.iastate.edu` and login using your COM S username and password. Then, set `CLASSPATH` environment to include `.` and `/opt/JML/bin/jml-release.jar`

4.1 JML/JUNIT TESTING OF CLASS LIGHT

This problem is designed to get you familiar with the JML tools for unit testing, on a very simple problem. In the directory where you copied the files, you will find an interface, `LightType`, and a class, `Light`, that implements that interface. You may want to read over the files `LightType.java` and `Light.java`.

COMPILE AND RUN THE JML/JUNIT TESTS FOR CLASS LIGHT.

To do this, first be sure that both `.` and `jml-release.jar` file are in your CLASSPATH.

Once you are sure that your CLASSPATH is right, execute the following commands at the command line:

```
jmlc -Q LightType.java Light.java // Compile JML specs and Java code.
jmlunit -i Light.java // Create oracle, testdata stub, test driver
javac Light_JML_TestData.java Light_JML_Test.java // compile driver
jmlrac Light_JML_Test // run test driver
```

Details on these commands are available using “man <command-name>”.

DEBUG, FIX, RECOMPILE, RERUN

Using the output of the last command, find the bug in `Light.java` and fix it.

Now working with the corrected version of `Light.java`, compile and run the JML/JUnit tests again, as above, to demonstrate that there are no more failures in the testing.

==>> Put your corrected version into a new folder named 417lab08.

4.2 JML/JUNIT TESTING OF CLASS LIFT

In this problem you will work with tests of the `Lift` class.

COMPILE AND RUN THE JML/JUNIT TESTS FOR CLASS LIFT.

Run the tests for `Lift` by executing the following commands at the command line. (Be sure you have the right CLASSPATH as in problem 1.)

Execute the following commands at the command line:

```
jmlc -Q LightType.java Light.java LiftType.java Lift.java
jmlunit -i Lift.java
javac Lift_JML_TestData.java Lift_JML_Test.java
jmlrac Lift_JML_Test
```

DEBUG, FIX, RECOMPILE, RERUN

a) Create a text file named lift_answers.txt and enter your answers to the questions in this section in this file.

b) In the output from the above command, there should be several failures. Each failure describes where in the source files the specifications are that describe that error. Find one of these specifications and comment out the clause that is detecting the failure. (You can do this by putting // on each line, to the left of the clause in question.) Then run the tests again, by executing the commands above. Briefly describe how that changes the output of the tests, and why it changes.

c) Suppose there were no JML specifications at all in the files you were testing, what do you think would happen when you ran tests? (Give a brief explanation.) What does this tell you about the relationship between testing with jmlunit and the JML specifications?

d) Fix the file Lift.java, and then rerun the tests to demonstrate that there are no errors. Note that you may have to iterate the process of running the tests and fixing errors.

e) There is at least one error in Lift.java, which you may have seen earlier, but which was not uncovered by the tests. This error is not shown by the tests because the test data, which is created by the file Lift_JML_TestData.java, is inadequate. Look in the file Lift_JML_TestData.java and find the declaration of the variable vLiftStrategy (near line 174). In the code that initializes this variable, there is an anonymous class that extends org.jmlspecs.jmlunit.strategies.NewObjectAbstractStrategy(), overriding the method make(int). This method is what generates the data of type Lift that is used in testing. The problem with this method is that it does not generate Lift objects in enough states to reveal some bugs. Add other cases to this switch statement (the new cases should be consecutively numbered) so that the new data reveals bugs in other methods in Lift.java. Show the output from testing that reveals these bugs, then fix them, and run the tests again to show that the bugs really are fixed.

==>> Put the lift_answers.txt and the corrected Lift.java in the folder 417lab08.

==>> Zip up 417lab08 folder and email to smitra@iastate.edu and kslu@iastate.edu with subject line: cs417 Lab08 submission.

5 SPECIFICATION and TESTING Problem (THIS IS A HOMEWORK)

In this section, your mission is to work on writing specifications for BuggleSort, filling in test data, and running and fixing BuggleSort. The steps are:

- 1) Write the JML specifications for BuggleSort (there are two requirements – one is that the final array is sorted and the other is that the final array has the same elements as the original array. Ignore the second requirement).
- 2) Run `jmlc` and `jmlunit`.
- 3) Search for `make` and `addData` in `_JML_TestData.java` file. Enter your test data there (these are data you figured out in lab01, 02 etc for BuggleSort).
- 4) Compile the `_JML_` files using `javac`
- 5) Run `jmlrac` `_JML_Test.java` file.
- 6) Use the failures to fix bugs in BuggleSort.

- 7) You need to zip the following and submit to smitra@iastate.edu and kslu@iastate.edu with the subject line: cs417 HW4
 - a. Your modified BuggleSort.java (with the JML specs and bugs fixed)
 - b. Your BuggleSort_JML_TestData.java file
 - c. Output of step 5)
 - d. Also, write up JML specs for the “second” requirement - i.e. that the final array has the same elements as the original array.

ACKNOWLEDGEMENTS:

This homework is an adaptation of materials provided by Dr. Gary Leavens. Thanks to Dr. Leavens and to Kai-Shin Lu for making this lab/hw possible.