Lab Exercise 1
Using EGit and JUnit

This lab exercise will get you familiar with following:

- **EGit**, an Eclipse plug-in to use to a distributed version control system called Git.
- **JUnit**, a unit testing framework for Java programming language.
- **Eclipse Debugger**, Eclipse has a built-in debug feature with its own perspective to help you resolve bugs in your software.

Throughout this course, you will use EGit to import local copies of your labs and assignments. You will make modifications to the project and then commit (record the changes) and push (update the remote repository) those modifications. After the deadline, the Grader will pull your remote repository for the assignment to grade. This means that you will need to remember to ensure that the remote repository has all of your final changes.

To also help you with your assignments, you will be supplied JUnit test cases, that will help ensure that your program is working as it should. These tests may or may not be used to dictate your grade for the assignment, in that the Grader may write separate test cases or may not use some of the supplied JUnit test cases. These are supplied for your convenience.

Debugging is an important part of programming and a crucial skill to master in order to be successful in this course. Eclipse provides a set of tools to help you navigate through the program and resolve bugs.

In this lab exercise, you will be asked to perform actions to familiarize yourself with EGit, JUnit tests and debugging.

1  Importing a Git Repository

Follow the steps below to import a Git Repository

1. Open Eclipse. *Tip:* If you are using a lab computer, you can preserve your workspace between logins by putting it on the I:\ drive.

2. Import a Git project by doing one of the following methods.
   - In the Eclipse Menu bar, select **File -> Import**.
   - In the **Package Explore View**, right-click and select **Import**.

   Regardless of which method you choose a **Import** dialog should display.

3. Expand the **Git** folder by double clicking and highlight the **Projects from Git** and click **Next**.
4. Select Clone URI and click Next.

5. Enter the following information, substituting ⟨pantherId⟩ with your own.

   • Host: andrew.cs.uwm.edu
   • Repository path: 
     /afs/cs.uwm.edu/users/classes/cs351/401/⟨pantherId⟩/git/lab1.git
   • Protocol: ssh
   • Port: 53211
   • User: ⟨pantherId⟩
   • Password: ⟨your panther password⟩

Do not write in the URI box, it should be automatically generated from the above information you entered. Also the choice of whether you want to store the authentication information in Secure Store is up to you. Despite being convenient, the storage is (somewhat) unsafe. An example image is given below. Click Next

![Import Projects from Git dialog](image)

Figure 1: Import dialog

6. At the next Dialog ensure that master branch is selected. Click Next.
7. The Dialog should now prompt for location for the local project. You may change this to what you are comfortable with. Click Next.

8. The next Dialog will be a radio selection that may be defaulted to Use the New Project wizard. Change the selection to Import existing projects and click Finish. The project should now show up in your Package Explorer View.

2 Coin Dispenser Problem

In this lab you are given CoinDispenser.java located in the edu.uwm.cs351 package. The Coin Dispenser simply takes a value in terms of pennies, i.e. $1.52 would be 152, and dispense that value into the amount of quarters, dimes, nickels and pennies.

3 Testing with JUnit

We also provided a simple test suite TestCoinDispenser.java for you. To run the test suite in Eclipse, you can do one of the following:

- In the Eclipse Menu bar, select Run -> Run As -> JUnit Test.
- Right-Click on the project or Test Suite in the Package Explorer View and select Run As -> JUnit Test.
- Click on the green button with an arrow in the Eclipse toolbar below the Menu bar.

Note: The Run Button will run the previously run configuration, but should prompt for a run configuration if none was previously stated and multiple run types for the project exist.

4 Locked Tests

Upon running the test, a dialog named “Unlock Test Case” should pop up. Look at the last line above the input box, three question marks (???) take the place of expected value in the assertion. Try to input a 4 and click “Check”. It should tell you it is not the correct value. Try again, how many quarters should be dispensed out of 78 cents? Notice that it tells you this test is unlocked after the correct value is given. Now it jumps to the next locked test, look back and notice the correct value has replaced the ??? in the previous test (in the dialog only, the source code cannot be modified of course). This time pretend you do not know the answer and click “Cancel”. It jumps to the next locked test in testDimesDispense. Cancel all remaining dialogs, later we will show a preferable way to unlock all the tests.

Go back to the Package Explorer, right click on lab1, and select Refresh. A new file named TestCoinDispenser.tst should appear at the bottom. This automatically generated file contains the keys so next time the test is run, you do not need to answer all the questions (that you had correct answers for) again. This is a new file created in the project,
which means you must check this file manually the first time you push otherwise it will not be included. Refer to the section on Committing and Pushing for more information.

Now let us take a closer look at test cases and locked tests. Open the TestCoinDispenser.java file, in the testQuarterDispense method, the first line sets the initial value to be 100 cents. In the next line, the assertEquals method asserts that the expected value (i.e. 4) should be equal to the actual returning result from the dispenseQuarters method call. That is, if implemented correctly, it should dispense 4 quarters out of 100 cents. Now look at the next test, we set the initial value to be 78 cents. In the next assertEquals call, the expected value is locked (hidden) from you, instead, a hash code (480802330) is given. This is a locked test. There are a few more locked tests in this test suite. The purpose of having locked tests is to make sure that you understand the expected behaviour of the methods you are going to implement.

In most cases, you should try to unlock the tests before writing any code. For this purpose, a separate driver class UnlockTest is provided in the default package. This driver will not run the tests, it only unlocks them (so you do not need to implement anything before running this driver). Right click on it and run as Java Application. The unlock test dialog should pop up again. Notice that it remembers the correct answers you provided so you do not need to unlock the tests more than once. Try to unlock all the tests by providing correct expected values. Notice the text in front of the input box, it indicates the expected data type. For Boolean, the only legal inputs are true and false. For String, you do NOT need to include any quotes.

After unlocking all the tests, the JUnit test should finish shortly. If your program passes all the tests, you will see a long green bar at upper side of the view. Otherwise, you will see a red bar instead, along with number of errors and failures caused by running your program. For any test that did not pass, there will be either a blue or a red cross mark next to its name. A Blue cross signifies a failed assertion, and a red one signifies an error in the code ran for the test.

You will notice that you have both blue and red crosses. Click on the testPenniesDispense on the JUnit View and you can see from the Failure Trace that dispensePennies caused a StackOverflowError. Double click on a trace line, it should bring you to the line of code that caused the error. Take a look at the code and you can see that there is a dispensePennies call within itself. This was done on purpose to show you a failed test. Go ahead and remove the call from the code.

5 Debugging

When debugging it may be helpful to know the line number of the code that you are viewing. Enable line numbers by selecting Window > Preferences from the Menu Bar, expand General > Editors in the Preference dialog, highlight Text Editors, and check the box Show line numbers.
Double click on `testQuartersDispense` and you should be pointed to the line where the assertion failed. In the Failure Trace, you can see that we expected a value of 1 to be returned from `dispenseQuarters()` but it was 0. Set a breakpoint at the line where the assertion was failed so that we can debug the code and see what is misbehaving. To do this, in the Editor View double click on the line number of the line you want to set the breakpoint. A blue dot should appear like the image below. Now you can run the Debugger by one of the following method:

- In the Eclipse Menu bar, select Run -> Debug As -> JUnit Test.
- Right-Click on the project or Test Suite in the Package Explorer View and select Debug As -> JUnit Test.
- Click on the green bug button in the Eclipse toolbar below the Menu bar. Note: The Debug Button will run the previously ran configuration, but should prompt for a debug configuration if none was previously stated and multiple debug types for the project exist.
- In the JUnit View, right click on the test that you want to debug on, then choose Debug in the drop-down menu. Tip: this is the preferred way to start debugging in most cases since it allows you to run a single test.

Eclipse should prompt to change to Debug Perspective. Click Yes to change the perspective. The program will now stop at the first breakpoint you have set and allow you to view the current status of the program and step through your code. At this point you can view the variables at the current breakpoint and do any of the following

1. **Step Into (F5)**, will step into the current method call.

2. **Step Over (F6)**, will step over the current line.

3. **Step Return (F7)**, will continue stepping until the current method returns

At this point we want to step into the method call `dispenseQuarters` to see what is going wrong. Click on Step Into, you should run into a page saying “source not found”. This is because there are several method calls on the line you set a breakpoint on so step into tries to bring you into the first one (which is the locked test `Ti(1891568934)` but the source code
of this method is not provided. In this case, click on **Step Return** to get back to the line we start with. Now the first method has been executed, click on **Step Into** again, it should bring you to the `dispenser.dispenseQuarters()` call. And do **Step Into** one more time so we reach `dispenseCoin`. In this method we are dispensing as many coins of the given value from the remaining values. From the **Variables View**, note the values of the following variables:

1. remainingValue
2. coinValue

Step to the **if** statement and consider the following questions:

1. With the remaining value above, if we were dispensing quarters would we expect at least one quarter to be dispensed?
2. The if statement serves the purpose to reduce code in the event that the remaining value is less than the coinValue, should we execute the if block?
3. Step into the block, did it behave as you would have expected?
4. What is wrong with the **if** statement?

Correct the **if** statement and save. You can resume the code by clicking the green arrow in the **Debug View** toolbar, or pressing **F9** or terminate the **debugger** by clicking the red box in the **Debug View** toolbar.

Rerun the tests and ensure that you pass all of the test cases. If you would like, you are supplied with a small program to test the code yourself. Simply run `CoinDispenser.java` as a Java Application.

## 6 Adding New Files via EGit

You will be required throughout this semester to add new files, such as Java classes or answers to questions, to the git repository. Follow carefully the section on Committing and Pushing to ensure that all new files are checked when you commit and push. Be sure well before the deadline to make note of any uncommitted (> icon) and unpushed (up arrow icon) files as shown in figures 3 and 4 below. More EGit icon descriptions may be found on the EGit wiki at [https://wiki.eclipse.org/EGit/User_Guide/State](https://wiki.eclipse.org/EGit/User_Guide/State). Create a file under lab1 called `answers.txt` and write your answers to questions 1-4 from the Debugging section.

## 7 Committing and Pushing

To commit and push your changes to the remote repository, perform the following actions:

1. Right-click on the program and select **Team -> Commit**. A **Commit Changes** dialog like the first example on the next page should display.
2. Enter a detailed message about your changes.

3. Ensure the items you have made changes to or added are staged (drag from “Unstaged changes” to “Staged changes”).

4. Click **Commit and Push**. If you do not see this, you maybe using an older version.

5. If you unchecked **Use Secure Storage**, you will be prompted again for your panther password. Enter it and again you can uncheck **Use Secure Storage** to avoid storing your password.

6. A **Push Results** dialog should display stating something along the lines of “Pushed to lab1 origin”. There shouldn’t be a problem in this lab, but you should read the dialog to see if the push succeeded just to be safe. If you get a message about “*fast-forward*” errors, it didn’t push. Click **Ok**.

Congratulations you have successfully pushed your changes.

After you finished all the above steps, see your TA to get credit for this lab. Or if you are doing this lab remotely, send a confirmation email to **compsci-351@uwm.edu** along with your answers to the **Debugging** section.
Figure 5: Commit Dialog with staged changes