# ECE 322 Lab Assignment 5

*Integration Testing (White-box Testing #2)*

## Overview

The objective of this lab is to become familiar with integration white box testing techniques tools. This lab makes of use of Java and JUnit testing for integration testing.

## Introduction

The following sections will serve as a brief introduction to integration testing.

### Integration Testing:

Integration testing serves as a logical extension of unit testing. There are two general approaches to integration testing:

- *Non-incremental testing (Big Bang):* Test each module **independently** then test the system as a whole
- *Incremental Testing (Top down/Bottom up):* **Combine** the next module to be tested with the set of previously tested modules before running tests. Generally done in either a bottom up or top down method
  - ➢ *Bottom up:* Test the lowest level modules in isolation, then incrementally add higher and higher level modules.
  - ➢ *Top down:* Test the highest level modules in isolation stubbing out lower level functionality, incrementally add lower modules.

Generally, integration testing requires the use of various stubs and drivers.

- *Stubs:* In integration testing a stub is used as a stand in for lower level modules not currently under test. Generally, a stub returns a dummy value or simply makes an assertion so that the test case can ensure it was called.
- *Drivers:* A driver is simply a piece of testing code which makes it possible to call a submodule of an application independently. Driver code may require *stub* setup, object initialization and so on.

### *Mocking Frameworks*

Mocking frameworks can be extremely helpful in abstracting away much of the grunt work involved in creating and maintaining stubs and drivers. These frameworks allow for the easy creation of Mock Objects, which are usable as stubs in integration tests.

Mockito is a popular Java mocking framework, you can install this package using gradle by including:

```
repositories { jcenter() }
dependencies { testCompile "org.mockito:mockito-core:2.+" }
```

In your gradle build file, declare a maven dependency:

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-all</artifactId>
<version>latest</version>
</dependency>
```

Or download the latest jar file from:

http://jcenter.bintray.com/org/mockito/mockito-core/

By creating *mock objects* you will be able to define object interfaces to use as stubs in your test cases, and easily verify the correct behavior of the modules under test with minimal boilerplate code.

Some tutorials on how to use Mockito can be found at:

- http://www.vogella.com/tutorials/Mockito/article.html
- https://dzone.com/articles/getting-started-mocking-java

Or check the official documentation:

- http://static.javadoc.io/org.mockito/mockito-core/2.8.47/org/mockito/Mockito.html

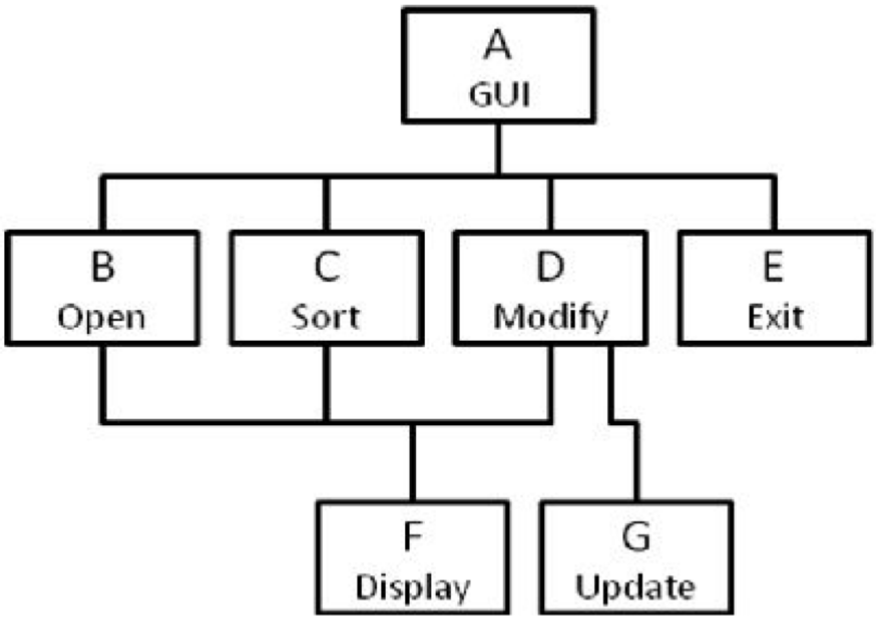
For the ease of marking, you can use Mockito 2.x in your tests.

## Preparation:

Prepare test cases for the tasks you will be working on in the lab session. Make sure you have Eclipse and JUnit installed on your account, or an equivalent Java IDE. The code for this lab is available on the class website and can be imported into eclipse as an existing project.

## Task (100 marks)

Consider a simple database system which has been constructed in a modular fashion: Module A invokes Module B Module C, Module D, and Module E. Module D invokes Module F, and Module G. Module's B and C also invoke Module F.



Entries in the database are composed of two elements, a String *name* and a String *Phone number*. Module A is a Command Line Interface which processes command strings and delegates functionality to sub modules:

1. Open a data file (Module B)
2. Sort records (Module C)
3. Modify a record (Module D)
4. Exit (Module E)

Files for this database should contain one entry per line, and elements of an entry should be comma separated. Sorting sorts records by first name.

The modify function (Module D) additionally uses the Display function (Module F) and the update function (Module G).

**Your task** is to prepare and run test cases, stubs, and drivers for:

- Big Bang Integration
- Bottom Up testing
- Top Down testing

**Unit tests should cover the full functionality of each module;** there should be at least one test for each piece of functionality and method. Your test cases should strive for, at a bare minimum, statement coverage. You **do no**t need to main function of the provided code as it implements a simple command loop.

Only the **public** methods of Module A need to be tested directly. Use the provided printstream accessor to test String message outputs from the module.

Your test cases **must be broken into three distinct test suites,** one for each type of integration testing (3 test suites total). The **reuse** of individual tests is **encouraged;** however, be aware that *test order and setup is key* to correct integration testing. Your test suites should be easily run against the provided application code for verification.

Run the test cases and record your observations and results in your report, as in previous labs *test cases must be presented in your report as a test case table* with meaningful descriptions, expected and actual results. Failed test cases must be highlighted.

Comment on the effectiveness of integration testing in isolating individual modules. Would this type of testing be helpful in a large scale system? Why? Are drivers and stubs an effective method of isolating modules? Which type of integration testing do you think is best? Which would be the most appropriate for a test driven development environment? Which would be the most appropriate for library development? Which is the easiest to maintain? Consider these and similar questions in your discussion of integration testing to show you have a strong understanding of the underlying concepts and motivations of this testing technique.

Your code should include:

- Unit tests for all **public functionality** of Modules A through F
- Unit tests testing module **integration** with proper use of **stubs**
- **Three test suites** , one for each type of integration testing, with test cases in the **correct order**
- You **do not** need to submit correct code for this assignment.

Your report should include

- Three **test case tables,** one for each type of integration testing. Test case reuse is okay where applicable; however results should be **fully** listed in each table for clarity.
- A discussion of **your results** regarding the application under test. This should include a discussion of any errors you found in the code, and **fixes** to make your tests pass
- A discussion of the above questions regarding integration testing techniques.

## Lab report:

Must be typed, no handwritten versions will be accepted. Follow the general format as included in the lab guideline. Submit all code via email. Your report should include:

- Your write-up
- The results of your test cases, in the form of test case tables
- Any conclusions you have drawn from these test cases regarding the applications under test. This should include your discussion focusing on the questions raised in each section
- Your test code as JUnit Java files to be easily executed against the provided application code

Clearly indicate all failed test cases, and explain the cause of these failures.