Unit Testing and Automatic Build Tools

This recitation introduces you to several tools to help you test—and automate the testing of—your applications as you develop and maintain them. You will first use Apache Ant and Travis CI to build and test your code as you develop it. Then you will write some unit tests and use a tool called EclEmma to measure the coverage of your tests.

Build and test automation

Ant is a build tool (similar to Make or Maven) that coordinates the build process, from creating output directories to compiling the code and running unit tests. Travis CI is a web service that automatically builds your code and runs tests when you push to GitHub, sending you email if the tests fail. These tools enable you to more-easily maintain the integrity of your code, ensuring that new code does not break your project.

To complete this task you should:

1. Install Ant using the instructions posted on Piazza, if you have not already done so.
2. Inspect the `build.xml` file in your `recitation/02` directory.
3. Run `ant test` from the command line inside your `recitation/02` directory.
4. Based on the `build.xml` file, on which commands does `ant test` depend? What command can you run to only compile (rather than compile and test) your code?
5. Edit and commit a minor change to the recitation code and push your repository to GitHub.
6. Log into Travis CI (http://travis-ci.com) and investigate the result of the last build. Travis CI should report that your project passed all test cases.
7. (Optional) Edit a test case so that the test fails and push to GitHub. Again inspect the results of Travis CI.

Unit testing and code coverage

In this part of recitation you will write unit tests for code we've provided and use EclEmma to measure the coverage of your tests.

Two distinct types of unit testing are *black-box testing* and *white-box testing*. Black-box testing is when you write high-level test cases against the specification of a program,

without using (or testing) the specific underlying implementation of the program. In white-box testing, however, your tests are tailored to the specific underlying implementation of the code you are testing.

In the exercise below you will write black-box tests for a `LinkedIntQueue` class and you will write white-box tests for an `ArrayIntQueue` class. The `LinkedIntQueue` tests are an example of black-box testing because the source code for `LinkedIntQueue` is not provided, and your tests are based on the specified functionality of the `IntQueue` interface rather than the features of the underlying `LinkedIntQueue` implementation. Your `ArrayIntQueue` tests should be white-box tests; you should write test cases that specifically test underlying features of the `ArrayIntQueue` class, not just test against the `IntQueue` specification.

As you write tests for this exercise you should use EclEmma to measure the code coverage of your unit tests, i.e., the percentage of code which has been tested by your unit tests.

To complete this task you should:

1. Install EclEmma using the instructions posted on Piazza, if you have not already done so.
2. Read the Javadoc comments for the `IntQueue` interface, and familiarize yourself with the preconditions and postconditions for each method.
3. Write unit tests for the `LinkedIntQueue` class in the `IntQueueTest` class. Each `IntQueueTest` method should test a specific functionality of the `IntQueue`.
4. Modify `IntQueueTest` to run on the `ArrayIntQueue` class instead of the `LinkedIntQueue` class. There are a few bugs in `ArrayIntQueue`, so at least one test should fail.
5. Investigate test coverage with EclEmma.
6. Fix each bug that was found.
7. Continue writing test cases until you achieve 100% line coverage with EclEmma.