Testing Your Projects in The Joy of Coding

For many years, the conventional wisdom in the field of Software Engineering was that software development and software testing were two separate activities that were performed by different people in different roles. While this point of view did allow for specialization, it had many downsides:

- There were long feedback loops between code being authored and code being validated. Developers had to fix bugs in code that they wrote weeks or even months earlier.
- Far too often, there was not clear agreement on how the software ought to behave, resulting in wasted effort.
- There was a perception that the software was "done" before it had been properly vetted.
- In some organizations, people who focused on quality assurance were viewed as second-class citizens. This resulted an unhealthy working atmosphere in which Developers and Testers unnecessarily competed to "be right". As a result, software took longer to build and lower-quality work was tolerated, to the detriment of users.

More recently, the software industry's view towards testing has changed. The introduction of easy-to-use unit testing frameworks and the rise of methologies such as test driven development, have elevated the importance and acceptance of testing software while it is being developed.

Because automated testing is such a vital part of contemporary software development, you are required to author tests[1] for your projects and submit them along with your application source code. When I grade your projects, I will run the following Maven command which will execute your unit and integration tests using the JaCoCo code coverage utility (`https://www.eclemma.org/jacoco/`).

```
mvnw -Pgrader clean verify
```

In this configuration (with the `grader` "profile"), the Maven build will fail if the minimum code coverage requirements are not met. You can view information about which classes were covered by tests in the `target/site/jacoco` and `target/site/jacoco-it` directories.

I expect that every class you write is exercised by automated tests and that at least 75% of the JVM instructions that comprise your project are executed by your automated unit and integration tests. If your unit tests do not provide sufficient coverage, I will deduct up to one point from your project grade[2].

Jacoco modifies the Java class files (bytecode) in order to track code coverage. Once class files have been modified, they may encounter problems when run as plain-old unit tests. Students have observed confusing errors like `TestEngine with ID 'junit-jupiter' failed to discover tests` and `ClassSelector resolution failed` in IntelliJ when classes with modified bytecode are run. They have been able to resolve these errors by executing `mvnw clean` from their command line environment and/or run "Rebuild Project" from the "Build" menu in IntelliJ.

[1] Note that the Maven archetypes create some unit and integration tests to get you started. You will evolve these tests as you implement your projects.

[2] Yeah, I know that measuring code coverage is super-controversial and doesn't directly equate to code quality. However, I want you to gain experience writing unit tests and be rewarded for writing unit tests, beyond just passing all of the functional tests. If you have other ideas for how I can assess the unit tests that you write, I'm very open to other strategies.

Also, note that invoking `System.exit` from unit or integration tests will terminate the test JVM before all tests have executed. Invoking `System.exit` from your program is **not** required for the projects. If you see a message like this from IntelliJ:

```
Process finished with exit code
```

or messages like these when running tests from Maven:

```
[ERROR] The forked VM terminated without properly saying goodbye. VM crash
        or System.exit called?
[ERROR] Process Exit Code: 0
[ERROR] Crashed tests:
```

it's likely that your test invoked code that invoked `System.exit`.