Melissa Jones, Nicholas Benyo, Scott Moser
Seattle University – Professor McKee
CPSC5210-01
June 6, 2019

Milestone #2

I. Project Description

The project chosen by the D'Buggers team is the classic game of Battleship written in Java. In this variant, a user sets their ships on the board and plays against the computer via command-line interactions. Hits and misses are recorded under the hood as state is maintained by the program until a winner is declared. We are using the Eclipse IDE for development and the built-in JUnit5 framework for testing. Our project was forked from GitHub user *ymarcus93*, who committed the code in 2013 with MIT licensing. The following URL is a link to the forked project with our team's updates and tests.

GitHub Project URL: https://github.com/mosers1/Java-Battleship

II. Unit Testing

For the purposes of this project, we are defining a software unit as a class. After inspecting the code and familiarizing ourselves with the application of the game, we determined which classes and methods were and were not testable.

The Battleship project has a total of six classes with a total of 68 methods, of which 56 methods are public. Of these public methods, we determined those belonging to the Battleship class are not testable. This is because they are used to drive the program and interact with the user and the Battleship class is comprised mainly of private methods. We also determined that the Randomizer class is untestable as we cannot accurately determine an expected output based on methods that generate random booleans and integers. Our evaluation of all classes and methods can be found in our documentation of testability found at the following GitHub repository:
https://github.com/mosers1/Java-Battleship/blob/master/docs/UnitMapping.xlsx

For each unit deemed testable, we analyzed its source code and created white box unit test cases to achieve maximum code coverage. Where appropriate, equivalence classes were identified and boundary partition testing was performed. Overall, we achieved high code coverage for the units under test (UUTs). One exception is the generalPrintMethod() function in the Grid class. This function prints to the console and without mocking or redirecting the serial output for analysis, we would be unable to effectively provide test coverage for this function.

III. Build Script

A build shell script was created to automate the building of the source code and the JUnit test cases. The script contains a list of Java source files to build and loops through them. A standalone JUnit JAR file is used to allow the building of JUnit test cases directly from the command-line. Due to constraints on SU's CS1 server, we needed to limit the memory used by the Java Virtual Machine so it could run. Since the Java compiler on CS1 is so out of date, we introduced explicit dependencies on having a specific version of the Java Development Kit (JDK 1.8.0_211) installed at a specific path. This will help ensure a consistent build environment for whoever uses the script to build the project and unit tests.

Usage:

```
./buildTestSuite.sh
```

This script can be run in Windows (using Cygwin), Mac, or a Unix system. It has been run successfully on Windows using Cygwin, SU's CS1 server, and AWS Linux running on an EC2 cloud server.

Link to script on GitHub:
https://github.com/mosers1/Java-Battleship/blob/master/scripts/buildTestSuite.sh

IV. Regression Testing

After creating the build script, we created a regression testing shell script. Regression tests are useful for ensuring changes to source code do not adversely affect existing functionality.

Our regression test script runs our suite of unit tests a specified number of times, calculates and prints the results to the user, and optionally sends an e-mail to a specified recipient containing the results. All console output is directed to a log file in addition to the console to enable debugging and aid in automating test reports. The script measures the execution time to run the test cases and prints timestamps in the log for further analysis.

Usage:

```
./runTestSuite <numIter> [emailRecipient]
   - <numIter> - Number of times to run the test suite. Range: [1,10000)
   - [emailRecipient] - Optional e-mail address to notify with test results
```

This script can be run in Windows (using Cygwin), Mac, or a Unix system. It has been run successfully on Windows using Cygwin, SU's CS1 server, and AWS Linux running on an EC2 cloud server.

Link to script on GitHub:
https://github.com/mosers1/Java-Battleship/blob/master/scripts/runTestSuite.sh

V. Stress Testing

For milestone two, we also created a shell script to perform stress testing. This script kicks of a specified number of background instances of the test suite each executing a specified number of times. Results are then calculated and displayed to the user. All console output is also directed to a log file to aid with test report automation. The fork/join concept is employed in the stress test script to spin off multiple background tasks to load the CPU and then wait until they are all complete prior to analyzing the results.

Usage:

```
./stressTestApp <numInstances> <numIter>
  - <numInstances> - How many instances of runTestSuite to run in background
  - <numIter> - Number of iterations for each instance to run
```

This script can be run in Windows (using Cygwin), Mac, or a Unix system. It has been run successfully on Windows using Cygwin, SU's CS1 server, and AWS Linux running on an EC2 cloud server.

Link to script on GitHub:
https://github.com/mosers1/Java-Battleship/blob/master/scripts/stressTestApp.sh

VI. Test Results

A total of 34 test cases were created for the Location, Grid, and Ship classes. Upon executing all cases using the JUnit framework within the Eclipse IDE, all ran successfully. These results are identical when running our regression test script. No failures have been observed to date when running the stress test script for up to 10 minutes.

VII. Code Coverage

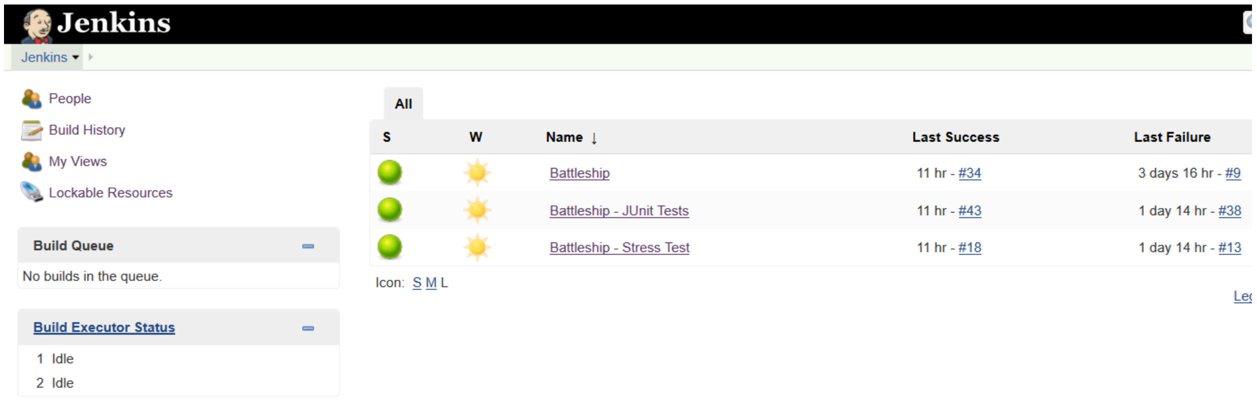
For the software units under test (UUT), 82.4% code coverage was achieved. This was measured using the code coverage tool built into the Eclipse IDE. As noted earlier, the reason our code coverage is not 100% is due to the generalPrintMethod() function in the Grid class. Testing this function would not be trivial and, as a result, was excluded from the scope of this project. Recalculating the code coverage with this function excluded yields 100% code coverage for the UUTs.

VIII. Continuous Integration [Extra Credit]

Rather than focusing on monkey testing or mocking for extra credit, our team determined learning how to setup a continuous integration (CI) server would be most beneficial to the team. This approach was approved by Professor McKee during an after-class discussion.

To set this plan in motion, we created an AWS EC2 server instance at the micro tier level (t2.micro) running AWS Linux. We then configured the security credentials via the use of public and private keys so we could SSH and SCP into the server. Once we had console access, we installed the necessary packages for Java's JDK, e-mail service (mailx), Jenkins, and Git. Once Jenkins was installed, we configured the necessary user accounts and began creating the individual Jenkins jobs.

Since we are using a public source code repository in GitHub, integration into Jenkins was rather straightforward. Each job leverages the scripts developed as part of milestone 2. This allows for source-controlled scripts to control the jobs which is a good practice. For quicker response to changes in the repository, webhooks were enabled so GitHub could notify our CI server whenever a change is made and immediately start a build. E-mail notifications are sent to the project team whenever a breakage occurs so it can be triaged immediately. The results of the failing job are included in the e-mail for maximum transparency. A screenshot of our CI server can be seen in Figure 1.



Build Queue

No builds in the queue.

Build Executor Status

1 Idle

2 Idle

S	W	Name ↓	Last Success	Last Failure
●	☀	Battleship	11 hr - #34	3 days 16 hr - #9
●	☀	Battleship - JUnit Tests	11 hr - #43	1 day 14 hr - #38
●	☀	Battleship - Stress Test	11 hr - #18	1 day 14 hr - #13

Figure 1. Jenkins CI Server Screenshot

Our server features the following jobs:

- Battleship: Builds the source code and JUnit test cases
- Battleship – JUnit Tests: Runs the unit test suite (regression test script)
- Battleship – Stress Test: Runs the stress test script

A guest account was created so our professor and other classmates can view our CI projects and get a feel for how Jenkins works. This account provides restricted access to the continuous integration server. They provide view-only access, the ability to download a workspace, and the ability to kick off a build.

Jenkins Credentials:

- URL: http://ec2-54-163-130-1.compute-1.amazonaws.com:8080/
- Username: mckeem
- Password: pickleball_2019

Due to cost constraints, the server will be live through June 19th until grades are posted. At that time, the server will be disabled. The team's GitHub repository is intended to remain public.

IX. Lessons Learned

Over the course of this project, our team was able to apply the software test and debugging theory we learned to a project written by a third party. To recap, we had to search for a testable project, identify the software units it contains, classify each unit and function as testable or untestable, write unit test cases aiming for maximum code coverage, and develop scripts to automate the building and testing of our project. Our team added and achieved a stretch goal of implementing a CI server. Several important lessons were learned during each phase of this project.

At the start of this project, we had initially chosen a different program to test. It was also a Battleship game program, but we quickly learned that not all programs are created equally in terms of testability. The initial program we chose required a significant amount of refactoring and/or mocking in order to produce a reasonable level of test coverage. All of the teammates work fulltime day jobs and it was not going to be realistic to test the first program. We learned a little bit of upfront design can mean the difference between testable and untestable software.

Most team projects, related to software or not, involve multiple team members needing to share and work on the same files. This is especially an issue when it comes to software development and test since we often need to add, delete, change source files and scripts over the course of a project. The important of using a source code management tool, like GitHub, was underscored during this project.

In our experience, most classes that teach programming often skim over or completely omit the topic of software test. While this is likely due to a limitation of how much material a class can cover during a quarter or semester, many students graduate with computer science-related degrees with little knowledge in the area of software test and debugging. This can lead to spaghetti code that is untestable and ends up being thrown away and rewritten. This is time consuming, expensive, and avoidable. We learned that designing software with test in mind can help avoid the need for code rewrites and, when paired with appropriate test strategies, can help deliver high quality software. Not only is this required for regulated industries like aviation and medical, it just makes good business sense regardless of the size of a project.

Through writing build and test scripts and setting up a CI server, we learned the sheer power of automation. Automation, ideally set up at the beginning of a project, allows projects to run tests early and often. In our example, our tests are run anytime a user makes a commit to our code base. Within minutes, they have instant feedback regarding the build status and whether any tests were broken. When a break is detected, the team is notified for quick triage. Additionally, using a CI server helps rule out build environment inconsistencies they can lead to builds working on one PC and not another. Setting up the CI server took less than a day. We learned it does not take much time to set up a CI server which leaves little room for excusing not having it on future team projects.

The overall experience has been positive for Team D'Buggers. The Battleship project we chose proved to be easy to understand, relatively easy to import and build, as well as easy to unit test. While several of our test cases ended up being rather simple, we still gained the benefits of practicing writing test cases, regression testing, and stress testing. Additionally, we were able to exercise our teamwork skills, use source control tools, learn about continuous integration, and debug our program and test code along the way. These skills are critical in the workplace and fundamental to learn in an academic environment.