

Milestone 2

I. Project Description:

The source project chosen by our group is a classic game called Tetris written in Java. In this game, a random block of different shapes will appear from time to time, allowing the user to rotate the blocks, move left, move right or move down. The goal is to fill as many lines of the grid as possible to earn points. Everytime the user makes a move, the program will actively check to make sure it's not blocked by the current state of the grid before executing the user's action.

We use Eclipse and IntelliJ IDEs as well as J-Unit framework for our test suite.

Github URL for Project Milestone2: <https://github.com/IrenaNiu/Java-TetrisTest>

II. Test Instructions (general):

1. Requirements:

- Eclipse IDE for Java
- JUnit 4 (built in Eclipse IDE)
- JDK

2. Project clone and import in Eclipse:

- Open IDE of Eclipse and import the project into the IDE:

File → Import → Git → Project from Git → Clone URL → URL:
<https://github.com/IrenaNiu/Java-TetrisTest.git> → follow the next steps until finish.

You will see the project loaded in the left navigation bar with all subfolders.

Steps for running unit tests and code coverage:

Open the package of "tested" and right click the any of the three test java files

● Run as "JUnit Test" → You will see the tests successfully pass within a second.

● Right click package "tested", click "Coverage as" → "JUnit test".
You will see code coverage results in the coverage panel.

III. Build Script:

In order to automate the building of the Tetris source code and the Junit tests and to simulate an environment conducive for regression testing, a shell script that compiles all the source code

and their corresponding tests was designed. The script was tested and successfully run on the CS1 server with the Java Development Kit (version 11.0.6) already available there. Additionally, JUnit (version 4.13) and hamcrest-all (version 1.3) JAR files were installed and placed in a “lib” folder within the directory that contained all the codes. The build script further sets up the Classpath variable to point to the locations of these JAR files. This is a dependency and it is crucial for these JAR files to be installed on the server, for the tests to be compiled and run successfully. Since there was a great deal of dependency between the Tetris source codes, the * wildcard was used in the script to compile all the source codes and Junit tests simultaneously. We further took inspiration from the Battleship Project to counter the issue with CS1’s memory constraints and used the memory flag “-J-Xmx512m” to limit the memory used by the Java Virtual Machine and aid in execution of the codes.

Usage:

- ☐ Use an FTP client like WinSCP to upload “Java-TetrisTest-master.zip” to any location on SU’s CS1 server
- ☐ unzip Java-TetrisTest-master.zip
- ☐ cd Java-TetrisTest-master
- ☐ ./buildTestSuite.sh

Link to Zip on GitHub:

<https://github.com/IrenaNiu/Java-TetrisTest/Java-TetrisTest-master.zip>

Link to script on GitHub:

<https://github.com/IrenaNiu/Java-TetrisTest/buildTestSuite.sh>

IV. Regression Testing:

The runTestSuite script was written to automate the execution of the JUnit test suite and simulate regression testing. The objective of this script is to ensure that whenever changes are made to the source code, the existing functionality can be tested in an automated manner against the predesigned unit tests and their outcome can be evaluated and assessed with minimum effort. The run script simply executes all the JUnit tests and presents their results, i.e. whether these passed or not. This script enables the tests to be run multiple number of times and additionally send their results as an email to a specified recipient. The script further logs

the time taken to execute the tests and prints these timestamps in a log file for further analysis. This script was also successfully run on the CS1 servers.

Usage:

```
./runTestSuite.sh <numIter> [emailRecipient]
```

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

Link to script on GitHub:

<https://github.com/IrenaNiu/Java-TetrisTest/blob/master/runTestSuite.sh>

V. Stress Testing:

We create a shell script to perform Stress Testing for our program. This script requires the two parameter inputs. The first one is a specified number of instances that will run on the background in parallel, and the second input is how many loop runTestSuit will do. After all the background instances of Test Suits completed, the script will be terminated. The result of how much time it took to complete all instances of Test Suite is logged to a file as well as showing the start time and end time on the shell.

Usage:

```
./StressTest.sh <NumBackgroundInstances> <NumLoops>
```

- <NumBackgroundInstances> : how many background instances the test will run
- <NumLoops> : how many loops runTestSuit will do

(e.g.: ./stressTest.sh 2 3. **CS1 has limitation of instances, better choose num < 5**)

Link to script on GitHub:

<https://github.com/IrenaNiu/Java-TetrisTest/blob/master/stressTest.sh>

VI. Test Result:










Unit test: A total of 34 test cases were performed to test BlockGrid, TetrisGrid and TetrisBlock classes. All passed without failures.

Regression Test: regression test scripts always passed, and can be run in cs1, windows, and mac OS.

Stress Test: no failure has been discovered when running the test for up to 200 instances

VII. Code Coverage:

For the all three tested classes, 91.0% code coverage was achieved. The main reason for not achieving 100% code coverage was due to the draw() method (in both BlockGrid and TetrisGrid classes), which utilized java.awt.Graphics module components for GUI of the game. The testing related to this method was not trivial and excluded from the project scope. Other than that, almost 100% code was covered for unit tests.

▼ tested		91.0 %	2,112	210	2,322
▶ TetrisGrid.java		61.2 %	229	145	374
▶ BlockGrid.java		74.4 %	157	54	211
▶ TetrisBlock.java		98.8 %	663	8	671
▶ AllTests.java		0.0 %	0	3	3
▶ BlockGridTest.java		100.0 %	320	0	320
▶ TetrisBlockTest.java		100.0 %	402	0	402
▶ TetrisGridTest.java		100.0 %	341	0	341
▶ (default package)		0.0 %	0	184	184

VIII. Lessons Learned:

This project helps our team to have a better understanding of the materials that we've learned in this class, as well as apply them to a real world example.

The very first thing that we did was looking for a public program that is testable. After carefully considering between several programs, we went with this Tetris Game application. Programs that we've tried to avoid are the ones that have complex codes that are written in such a way it is hard to write unit tests. Those are codes that have bad practices, or code smells such that having too many dependencies within a method. This project taught us not only how to perform unit tests but also how to write beautiful and functional codes that are testable. We've also learned that good unit tests need to be readable, which means that the intent of the tests is clear and describes the story of the behavior aspects that we are testing. As we worked as a team, readable tests would help the collaboration easier and more effectively.

Although manual testing processes identify many unique defects by utilizing exploratory methods; building, verifying, and testing the SUT repeatedly can be tedious and expensive and prone to human errors. This is where automation comes into the picture. By developing scripts that build and execute the SUT and the pre-written JUnit tests as a test suite with hardly any additional effort on the developer's end, we were able to automate the process particularly for regression and stress testing. Thus, whenever changes are made to the source code now, these scripts can be executed to verify and validate whether the functionalities still work fine or not. Developing tests for the source code and analyzing and refining our tests for better code coverage made us realize the importance of unit tests, and how these serve as a less expensive yet tremendously effective way of finding bugs and errors at a preliminary stage of the software life cycle. Moreover, by discovering bugs and defects, and by debugging early on, aided by some automation processes in place, we might be able to incorporate continuous integration concepts propagated by agile and

test-driven development methodologies. Our shared experience with the class project made us recognize that automated and manual testing processes do not stand as substitutes for one another but should be done simultaneously enabling developers and testers alike to automate the more tedious aspects of testing all the while benefiting from the application of human rationale and critical thinking, which lets us discover exceptional problems within any code.