Software Quality: 2D Game Framework Testing

Andrew Graff, Hailee Kiesecker

CS 574 Spring 2020

Abstract

Software Testing is a major field of interest within the computer science community to ensure quality production code. Throughout Boise State University's CS 574 course on software quality, students are taught different methods of software testing. Within this paper three different methods of testing are discussed and implemented on a 2D game generation framework and its results are reported.

1 Introduction

Software frameworks have the ability to deal with ideas for generic functionality in regards to a specific type of software. User code can be added to the framework by a user providing customized application software. More specifically a framework can be described as having, "defined open or unimplemented functions or objects which the user writes to create a custom application" [3]. This differs from a software library in the regard that a software library functions/methods can be called from but generally are not modified.

Within this document we evaluate and test a 2D java game framework. The usage of this specific framework allows the users to create 2D games in the java programming language using its provided classes so that users do not need to start from scratch in the creation process. This can be particularly useful for casual creators who want to get right into implementation rather than having to start from scratch. Uses for this framework include 2D game generation such as *Flappy Bird* or *Space Invaders*, simple testing for already created games, and references for learning.

Throughout the rest of this paper in section 2 we go over related works in regard to our testing and what type of tools and testing methods we used while evaluating our 2D game framework project. Preceding section 2 within section 3 we move on to talk about the framework and what was provided to us by the developers in their own debugging package. In section 4 we talk about this final project and the parts that we have provided in our GitHub repository. Section 5 goes over how to set up this 2D game framework and the running of some tests. Section 6 goes over our testing design and why we chose those testing methods. Section 7 briefly describes issues that we encountered along the way through this project. Section 8 goes over our results and interesting finds, then finally in section 9 we wrap up our paper with a conclusion.

2 Related Works

The java modeling language *(JML)* is used throughout testing of our 2D game framework. JML derived from Contract approach and model-based specification approach is a behavioral interface specification language. Which is used for testing within java programs through specificity of behavior of java modules. [2]

Java Assertions are used to declare an expected condition within a program. These conditions should return Boolean values at run time. If a false value is found it will display to the user an AssertionError. As a programmer or tester modifies the code that they are using Java Assertions on then these Assertions must be taken into account and modified as needed. Through contract a method implementing Assertions has a pre and post condition that the assertion can check to hold true. preconditions describe restrictions on input parameters while post-conditions describe restrictions on the computed values. [5]

EclEmma is a java code coverage tool for Eclipse. Part of white box testing which is covered more in detail in section 6, EclEmma goes through the programs code and takes all the branches that

testing is currently covering. If a branch is not covered EclEmma will show the particular branch in red, if it is it will be green. Their is a trade off between the number of test cases and the thoroughness of those tests depending on the situation. In our framework testing the number of our tests directly affects the thoroughness of our exploration of the 2D game framework code. This is reflected in section 8. [4]

We use JML, Java Assertions and EclEmma throughout our testing and talk about them within the rest of this paper. Out of our fifteen fifteen non-sequential tests, thirteen use Assertions and JML. Later on in this paper we talk about Java MOP, what it is and how it was useful in our testing.

3 Project Providence

The 2D Game Framework GitHub project there contained multiple example test implementations of the framework itself. In the bellow figures you can view some sample GUI's.

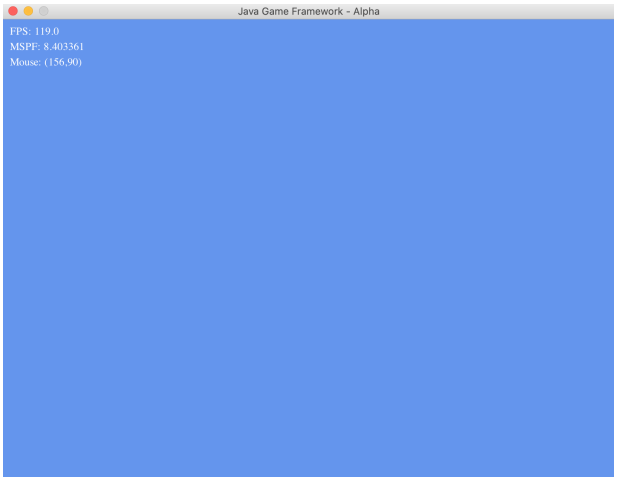


Figure 1: Sample Test Frame

Within figure 1 you can view the mouse positioning as well as the Frame Per Second refresh time. Throughout testing we used this test frame for generic test execution cases of JML and Java Assertions.

Figure 2: Accelerated Test Frame

Figure 2 shows a more advanced testing frame provided by the source providers. Within it you can see a bunch of wild multi shapes and colors. These are being called by the game.animations package and being written to the game.gui by the game.framework package. This 2D game framework java project consists of many sequential methods that build off of each other as well as many math helper functions that could cause many issues to users if there is anything wrong with the current values in the code.

In addition to the provided test cases in game.debug we added our own test frame called TestFrameworkMenu.java that is able to implement additional Framework functionality so that we can test out our created test cases with more accurately. This meant writing some of our own "games" to test the features supported by the framework. In this example, we tested out the menu creation to make sure everything displayed properly within the window.

4 Project Overview

For the this project we were to verify the 2D game framework Java program was efficient and reliable to use as a base framework for intro to Java game creation programmers. To do this we implemented 15 non sequential tests that used JML, Java Assertions, and outside of the project requirements a few JavaMOP files, described in section 6. These first 15 tests were used to test value inputs as well as test against output's calculated information to be sure that our frame was operating at a good capacity for it to be usable in an actual game implementation.

After completing our 15 non-sequential tests we then implemented 5 sequential tests. As stated previously our 2D game framework relies on the execution of different classes and methods working together to produce an accurate panel or frame that a user can play with. If anything goes in the wrong order it can cause the program to crash or information to be inaccurate. In section 6 we talk about JavaMOP along with how and why we used it as a major factor in our testing process.

5 Project Setup

In order to get this framework running on our local machine, we first need to clone the repository. Once we have the repository checked out from GitHub, it's time to play around! At the top level, there are two folders: ***nbproject*** (which contains Netbeans project support files) and ***src*** (the folder we care about as we are using Eclipse). Additionally, the author included several build scripts for building the project. In practice, these did not work out of the box, so we ignore them. There is also a README.md that explains how to run the compilation scripts. Under ***src***, there are seven packages: ***game.animation.ease***, ***game.animation.tween***, ***game.debug***, ***game.framework***, ***game.graphics***, ***game.gui***, and ***game.input***. Since the included compilation scripts do not work, we simply use Eclipse to compile and run all three test games.

The package ***game.animation.ease*** contains some classes that the author collected to help with some mathematical calculations necessary for animations in game design. ***game.animation.tween*** is simply a template for a MotionTween class that is unimplemented. We ignore these two packages as none of our testing covers animations.

The ***game.debug*** package is one of the primary packages we look at during our testing. It contains several classes we use heavily in our testing. *AcceleratedGameTest* is a test game that simply displays randomly colored, randomly sized boxes on screen. This test was nice to run our tests on and modify to see what kind of effects the testing had on the program execution. *PerformanceTiming* is a class that stores information about the games performance, such as the number of frames per second that are being rendered, and information about how to display that to the window. *TestFramework* is another class that is a sample game that simply displays a window with the *PerformanceTiming* information. It is also the base game that was used to generate the class *TestFrameworkMenu* in which we created our own game to test menu creation. We also have two JavaMOP files in this directory along with their compiled AspectJ files for the monitors. They are $\text{menu}_b ar_c reations.mop and super_i nitializaiton.mop$.

The ***game.framework*** package is the next big package that we utilize heavily. It has the base classes that are required for a game program. *Game* is the base class that extends *JFrame*, and the building block of this framework. There are several other helper template classes here, like *GameTime* and *GameHelper*. There are also some data structure classes here that are used in typical game development. *Vector2*-*Vector4*, *Rectangle* (for displaying windows), *Matrix*, etc. We primarily use and test *Rectangle* and *Vector2*. There is a unit test file here generated to test *Vector2* called *TestVector2*. There are also three JavaMOP files here and their compiled AspectJ files for the monitors. They are $\text{game}_i nitialization.mop, game_i nput.mop, and start_e nd.mop$.

The last really big package that we primarily deal with in our testing of this framework is ***game.gui***. This package contains classes that support creating menus and some templates for other GUI components. *MenuBar*, *Menu*, *MenuItem*, and *Anchor* are the classes used for menu creation.

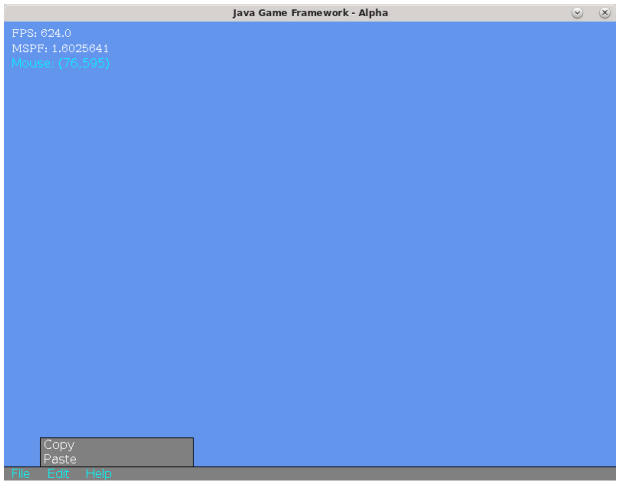


Figure 3: Sample Test Frame with several menus

The final package in this framework deals with inputs to the game. ***game.input*** contains classes that handle keyboard and mouse inputs. This package includes *Keyboard*, *KeyboardInput*, *Mouse*, *MouseInput*, *MouseAction*, and *MouseKeys*. These classes are used in *TestFramework*, *TextFrameworkMenu*, and *AcceleratedGameTest*.

6 Testing Design

Due to high amounts of calls from the Game classes the best course of testing correctness on the main functionality of the framework was decided to be mainly within the three classes game.framework.Game, game.framework.GameHelper, game.framework.GameTime, shown in figure **??**. With additional test cases created inside of game.input and game.framework. The main objective of our testing was to use as few tests as possible while maximizing the amount of code coverage of our test cases. A fair amount of the testing was around Java Monitoring Oriented Programming (MOP).

JavaMOP is an abstraction of program methods that allows the user to test sequential properties through Regular Expressions or other languages such as context free grammars. For our testing we used JavaMOP along with Extended Regular Expressions (ERE). This allowed us to take sequential properties of the game.framework package and express them through ERE into AspectJ code. [6] slides on JavaMOP

AspectJ is a aspect-oriented programming (AOP) extension that can be used in Eclipse to add AOP capabilities within a project. AspectJ is a great plug in for testing since it is able to add constructs to java that help in dealing with crosscutting concerns within a java project. crosscutting concerns are issues spread across a system and are not just bound to one module [1].

To test some of these classes, we used the category partition method to create some JUnit tests for one of the classes, *Vector2*. We created the file, Vector2.tsl that describes the categories. After executing TSL on this file, we execute a script to parse this file and generate JUnit tests.

```
Parameters:
        Function:
                add.    [property Add]
                addX.   [property AddX, SingleVal]
                addY.   [property AddY, SingleVal]
                barycentric. [property Bary]
                clamp.  [property Clamp]
                divide. [property Divide, SingleVal]
                dotProduct.     [property DotProduct]
        NumberOfVectors:
                1.      [if AddX || AddY || Add || Divide]
                2.      [if DotProduct]
                3.      [if Clamp || Bary]
        TypeOfInput:
                min.
                max.
                rand.
        NumberOfValues:
                one.    [if SingleVal]
                two.    [if !SingleVal]
```

Figure 4: TSL for Vector2

Through our JavaMOP testing we discovered that with the current game.debug package there is no sequencing errors being found however, in the future someone using the framework could potentially do a wrong set of calls if they do not know the framework and its code fully. While testing we had the advantage of having full access to all of the frameworks packages and test code. This is extremely helpful in white box testing the math helper functions in game.frameworks package. White box testing allows the users and testers to take sections of known code and base tests of it from knowing the parameter values that are allowed for a method or function.

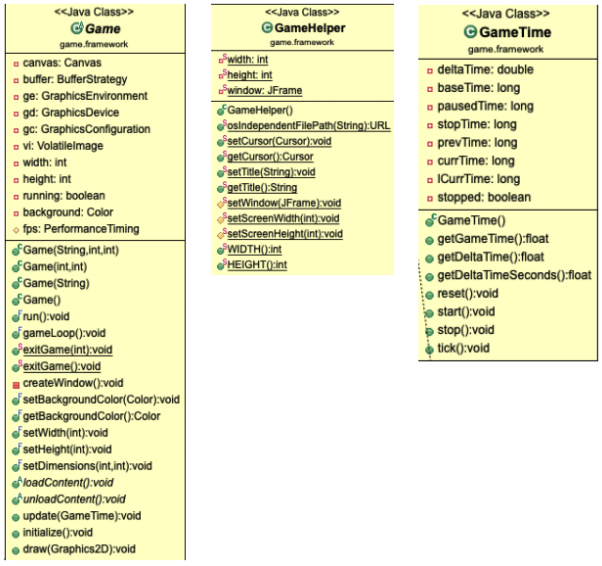


Figure 5: Game Class Outlines

7 Issues Encountered

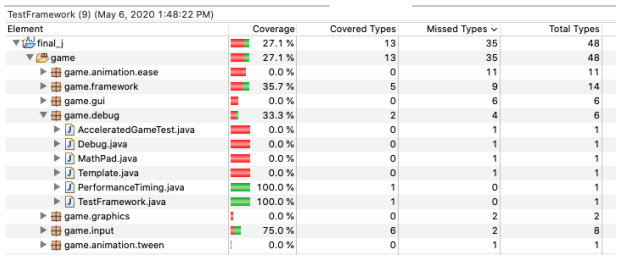
There were several issues we ran into when trying to implement this framework for testing. The first issue we ran into, as we alluded to earlier, was simply compiling the program from the command line. The build scripts did not work as specified and we could not configure them to work properly. We tried downloading and installing ***ant*** to compile using the *build.xml* file in the main directory which uses the Netbeans compile XML script in ***nbproject***/*build-impl.xml*, but this still didn't work. As a result we could not run CoverityScan on the project (it requires a successful compilation on the command line). We spent a lot of time fixing Doxygen comments due to compilation errors. There were many Doxygen comments that had incorrect argument names or weren't using up-to-date Doxygen syntax. Another issue we ran into was that it wasn't easy to test the *PerformanceTiming* class. It has a member *fps* that has no getter function. We had to create one in order to do some check on *fps*.

Some additional issues we ran into is that it is not easy to simply test a framework that is largely not implemented and only provides the infrastructure for you to implement yourself. It was good that the project included several test games to play with. It provided a "learn on your own" example, but we had to implement our own programs. We would have much more preferred to test a project that was already implemented, and spent more time on black-box testing rather than white-box testing.

8 Results

With most of or JML, and Java assertions tests on the 2D game framework there were no issues present. However, for testing out the frames per second to be sure that our game never dropped bellow an optimal level of performance we needed to create a helper function within the original code. This newly created getFps() function allowed us to return the actual float valued fps. Which allowed our JML assertion to be functional.

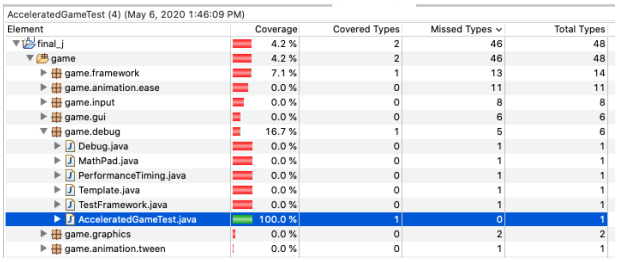
It was discovered that our currently given game.debug frame templates do not exercise the extent of the functionality of the 2D game framework. If you look at the code coverage displayed in figure 6 and 7 bellow you can see that it is never able to get into a majority of our functionality.



TestFramework (9) (May 6, 2020 1:48:22 PM)

Element	Coverage	Covered Types	Missed Types	Total Types
final_j	27.1 %	13	35	48
game	27.1 %	13	35	48
game.animation.ease	0.0 %	0	11	11
game.framework	35.7 %	5	9	14
game.gui	0.0 %	0	6	6
game.debug	33.3 %	2	4	6
AcceleratedGameTest.java	0.0 %	0	1	1
Debug.java	0.0 %	0	1	1
MathPad.java	0.0 %	0	1	1
Template.java	0.0 %	0	1	1
PerformanceTiming.java	100.0 %	1	0	1
TestFramework.java	100.0 %	1	0	1
game.graphics	0.0 %	0	2	2
game.input	75.0 %	6	2	8
game.animation.tween	0.0 %	0	1	1

Figure 6: TestFramework.java coverage

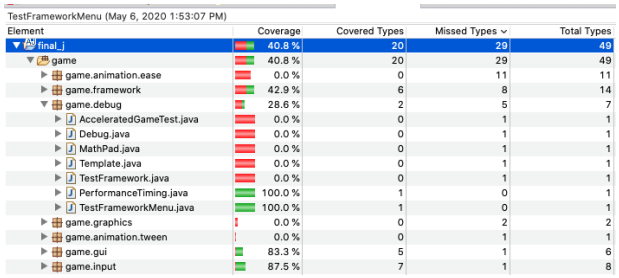


AcceleratedGameTest (4) (May 6, 2020 1:46:09 PM)

Element	Coverage	Covered Types	Missed Types	Total Types
final_j	4.2 %	2	46	48
game	4.2 %	2	46	48
game.framework	7.1 %	1	13	14
game.animation.ease	0.0 %	0	11	11
game.input	0.0 %	0	8	8
game.gui	0.0 %	0	6	6
game.debug	16.7 %	1	5	6
Debug.java	0.0 %	0	1	1
MathPad.java	0.0 %	0	1	1
PerformanceTiming.java	0.0 %	0	1	1
Template.java	0.0 %	0	1	1
TestFramework.java	0.0 %	0	1	1
AcceleratedGameTest.java	100.0 %	1	0	1
game.graphics	0.0 %	0	2	2
game.animation.tween	0.0 %	0	1	1

Figure 7: AcceleratedGameTest.java coverage

For this reason we developed our own debug test frameworks that allowed us to get a better overall coverage of the framework 8. If a user of this framework were to create a fully functional game there is no doubt that the code coverage would begin to reach 100 percent.



TestFrameworkMenu (May 6, 2020 1:53:07 PM)

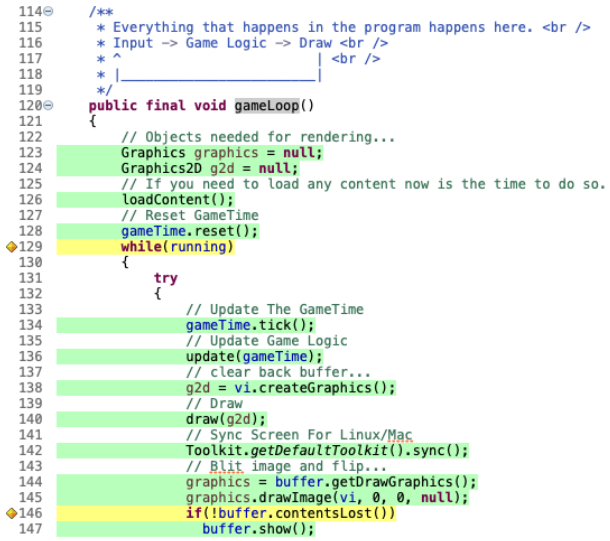
Element	Coverage	Covered Types	Missed Types	Total Types
final_j	40.8 %	20	29	49
game	40.8 %	20	29	49
game.animation.ease	0.0 %	0	11	11
game.framework	42.9 %	6	8	14
game.debug	28.6 %	2	5	7
AcceleratedGameTest.java	0.0 %	0	1	1
Debug.java	0.0 %	0	1	1
MathPad.java	0.0 %	0	1	1
Template.java	0.0 %	0	1	1
TestFramework.java	0.0 %	0	1	1
PerformanceTiming.java	100.0 %	1	0	1
TestFrameworkMenu.java	100.0 %	1	0	1
game.graphics	0.0 %	0	2	2
game.animation.tween	0.0 %	0	1	1
game.gui	83.3 %	5	1	6
game.input	87.5 %	7	1	8

Figure 8: TestFrameworkMenu.java coverage

What was interesting was where EMMA was able to find usage and not able to find it. Take for example 9 inside of Game after running our TestFrameworkMenu.java inside of the debug package.

```
90      */
91     public final void run()
92     {
93         try
94         {
95             // Creates the game window and double buffer
96             createWindow();
97             // Initialize anything if you need too
98             initialize();
99             // Start the game loop
100            gameLoop();
101        }
102        catch(Exception ex)
103        {
104            ex.printStackTrace();
105        }
106        finally
107        {
108            System.exit(0);
109        }
110    }
```

Figure 9: game.framework.Game- run() coverage
Our gameLoop() is never started however in figure 10 you can see that we did indeed access it and run our game loop. The two shown areas of code are the only way that gameLoop() is ever called.



```
114    /**
115     * Everything that happens in the program happens here. <br />
116     * Input -> Game Logic -> Draw <br />
117     * ^                       | <br />
118     * |_______________________|
119     */
120    public final void gameLoop()
121    {
122        // Objects needed for rendering...
123        Graphics graphics = null;
124        Graphics2D g2d = null;
125        // If you need to load any content now is the time to do so.
126        loadContent();
127        // Reset GameTime
128        gameTime.reset();
129        while(running)
130        {
131            try
132            {
133                // Update The GameTime
134                gameTime.tick();
135                // Update Game Logic
136                update(gameTime);
137                // clear back buffer...
138                g2d = vi.createGraphics();
139                // Draw
140                draw(g2d);
141                // Sync Screen For Linux/Mac
142                Toolkit.getDefaultToolkit().sync();
143                // Blit image and flip...
144                graphics = buffer.getDrawGraphics();
145                graphics.drawImage(vi, 0, 0, null);
146                if(!buffer.contentsLost())
147                  buffer.show();
```

Figure 10: game.framework.Game- gameLoop() coverage

Even though we attempted to test the sequence of menu creation, menu creation is actually quite flexible. In general, the Java graphical framework doesn't care much about the order in which GuiComponents are created or manipulated. It only matters if they are all put together completely before they are actually draw. So you can create the MenuBar, Menu(s), and MenuItems(s) in any order. Additionally, you can add MenuItem(s) to Menu(s), and Menu(s) to the MenuBar in any order. It only matter for the creator which items or menus get added in what order to properly display them in the order the creator intends.

9 Conclusion

In conclusion it was discovered that the GitHub project of a 2D Game Framework had minimal overall issues and would be good for intro to game development programmers with java coding experience. As a project targeted for software quality testing, however, we were quite disappointed in the available testability of the project itself. It required a lot of manual creation of programs to test. The 2D Game Framework is a template for game creation, most of the methods are left to the programmer to implement. If we had to do it all over again, we certainly would have chosen a different GitHub repository to use for exercising our software quality muscles!

References

[1] Frequently asked questions about aspectj.

[2] The java modeling language (jml).

[3] Software framework vs library, May 2019.

[4] Sherman Elena. Lecture 12 - white box testing and eclemma. 2020.

[5] Sherman Elena. Lecture 2 - java assertions. 2020.

[6] Sherman Elena. Lectures 6 and 7 - monitoring-oriented programming and javamop. 2020.