

Automated Testing of Tanaguru

4banger

CSCI 362: Software Engineering

Jim Bowring

College of Charleston

Cass Outlaw, Justin Arends, Scott Stolarski, Drew Bigelow

Table Of Contents

Chapter 1: Introduction.....	3
Chapter 2: Test Plan.....	7
Chapter 3: Testing Framework.....	11
Chapter 4: Implementation of Testing Framework.....	16
Chapter 5: Design Fault Injections.....	16
Chapter 6: Lessons Learned.....	20

Chapter 1: Introduction

Why Tanaguru Contrast-Finder:

We decided to build and test the Tanaguru Contrast Finder project. Tanaguru it is an accessibility tool for websites used to find contrast between font and background colors, aiding people suffering from colorblindness. Some of the key components that led our group to choosing this project were:

- 1) The Tanaguru underlying algorithm is written in Java, which is a language all members of the team are familiar with.
- 2) The Tanaguru GitHub, **at first glance**, looked well documented with an easy to understand file structure and a helpful interactive website.
- 3) Tanaguru included fully built test cases on their GitHub, which appeared to be well organized.

All of the above, were what our team believed to be supporting reasoning for our decision to choose Tanaguru.

Installation, Building and Compiling:

We were able to fork the Tanaguru project easily from GitHub onto our local machines. Once on our local machines we ran into the first problem, Tanaguru's ReadMe file contains no instructions on how to build the project. Only a link to their demo. After some research we were able to find a link to the Tanaguru wiki that had multiple build and installation instructions, one of which was just a blank page. After locating the most correct build instructions page (<https://github.com/Tanaguru/Contrast-Finder/wiki/Howto-build-&-install>) we began to install the dependent software Tanaguru needed to run on a local machine.

To install Tanaguru, these three dependencies are needed.

- Tomcat (version 6 or better)
- Maven
- Git

All of our team already had Git installed and we were familiar with Git operation. First we chose to install Maven, Maven is a set of software management tools for Java based projects. The original developers of Tanaguru implemented these tools.

Installation of Maven was not simple, but their website gave ample information (<https://maven.apache.org/install.html>). However we found an even simpler installation guide on

a public software blog. After Maven and Git were installed we could complete the first steps of the build instructions: `git clone https://github.com/Tanaguru/Contrast-Finder.git` `cd Contrast-finder` `mvn clean install`.

Apache Tomcat is a flexible, powerful, and widely popular application server and servlet container. Apache Tomcat is used to deploy your Java Servlets and JSPs. Which will allow a Java project to create a WAR (short for Web ARchive) file. (Apache).

After installing Maven the “mvn clean install” builds and compiles the project which triggered the built in test to run. The following are screenshots of our results in order as they were displayed over the terminal window:

```
.....
T E S T S
.....
Running org.opens.utils.colorconvertor.ColorConverterTest
getHue
java.awt.Color[r=128,g=128,b=127]
Hue : java.awt.Color[r=128,g=128,b=127]
Rgb2hexBlack
Rgb2hexWhite
Rgb2hexPink
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.09 sec
Running org.opens.utils.contrastchecker.ContrastCheckerTest
getContrastRatio
result :3.6102927852355164
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.011 sec

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

```
.....
T E S T S
.....
Running org.opens.colorfinder.AbstractColorFinderImplTest
findColorsWithValidContrastWithBackgroundTested
findColorsWithValidContrastWithForegroundTested
findColorsWithInvalidContrastWithBackgroundTested
findColorsWithInvalidContrastWithForegroundTested
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.074 sec
Running org.opens.contrast.finder.api.ApiTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

```
.....
T E S T S
.....
Running org.opens.color.finder.hsv.ColorFinderHsvTest
FindColorsWithFgAndBg
```



```
sudo cp color-finder-webapp/target/contrast-finder-webapp-1.0-SNAPSHOT.war /var/lib/  
tomcat6/webapps/ sudo mkdir /var/log/contrast-finder/
```

```
sudo chown -R tomcat6 contrast-finder/ sudo
```

```
invoke-rc.d tomcat6 restart
```

The problem with the commands on Tanaguru's wiki is the directories listed do not exist. Tanaguru has failed to update their wiki since the web app version 1, and the current web app we cloned is version 3. The lack of documentation by Tanaguru lead to hours of research and attempting things on our own. We weren't sure how to get the prerequisites to run properly so that we could compile Tanaguru. The main problem we are facing is that Tanaguru's instructions don't match directories, most likely because they are out of date. The installation instructions were updated last in 2015. We are currently working, blindly, to figure out how to get Tanaguru to compile. The most common error we are getting is that "(blank) directory doesn't exist." Which is due to poor and out of date documentation. Luckily we did not need to completely run apache to run the built in tests of Tanaguru.

Chapter 2: Test Plan

The Testing Process

Testing Key Methods of the Underlying Algorithm: We have chosen specific methods to test the functionality of Tanaguru. The methods we chose were chosen due to their impact on the final product of the Tanaguru web app, which is to specify if two colors have a good ratio of contrast between each other.

Requirements Traceability:

All of these methods are validated with third party sites, cited in each specific test case.

Return a contrast Ratio between two colors.

The first method to test is probably the most important one in the project which is `getContrastRatio(Color color1, Color color2)`. This method is one of the last methods called when executing the project, and this method will take two color values and return the ratio that separates them. We will be testing if Tanaguru's ratio matches that of other contrast ratio tools. We will also be testing if the method can handle unexpected inputs.

Return information about specific color objects, which are used to calculate the overall color ratio.

The second method we are going to test is `getLuminosity(Color color1)`. This method should be able to be given a color object and return a ratio value that describes how much white is in the color. This method should also be able to handle unexpected inputs without crashing.

The third method we are going to test is `getBrightness(Color color1)`. This method will return a ratio that represents the amount of white relative to the amount of black in a single color object. This method will be given unexpected values, as well as expected values to test its behavior.

The fourth method we are testing is `getSaturation(Color color1)`. This method returns a ratio that represents the amount of grey in a specific color. This method will be given unexpected values, as well as expected values to test its behavior.

Convert color objects to different types.

The final method we are testing is `rgb2Hex(Color color1)`. This method should return the hexadecimal equivalent of the input color which will be an RGB value. This method will be given unexpected values, as well as expected values to test its behavior.

Tested Items

The items that are subject to our test cases include the classes Contrast Checker, ColorConverter as well as the above mentioned methods.

Testing Schedule

The first five test cases will be done October 3rd. The automated testing framework will be done by October 31st. The fault injection will be done after the entire framework executes

Test Recording Procedures

The test outputs will be written into a text file. The expected results will be in another text file. The two files will be compared. If an output does not match the expected result, the test will be recorded as a failure. All outputs will be produced into an HTML document that can be viewed then saved after each execution cycle.

Hardware and Software Requirements

Hardware

A computer that is capable of running either ubuntu natively or running a virtual machine. As well as all the software required.

Software

All the prerequisite software (tomcat, maven, java, etc.), Ubuntu operating system, Tanaguru installed

Constraints

Bad documentation of Tanaguru

Tomcat and Maven issues could slow down test development

System Tests (for the first 5 test cases)

Below is screenshots representing the structure and layout of our test cases. Each test with an expected oracle that is not an exception will have the 8th line be a link we used to verify our results.


```

1  TEST CASE FILES
2  4Banger CSCI 362
3
4  THIS DOCUMENT SERVES AS THE TEMPLATE FOR ALL TEST FILES
5
6  Test Template
7  1. Test number
8  2. Class being tested
9  3. Method being tested
10 4. test input(s) to be given to the driver
11 5. expected outcome(s)
12 6. Requirement being tested
13
14 8. Source where expected output was verified
15
16
17
18 // TESTS 0 - 9 tests getContrastRatio() method
19
20 // TESTS 10 - 14 tests getSaturation() method
21
22 // TESTS 15 - 19 tests getLuminosity() method
23
24 // TESTS 20 - 25 tests rgb2Hex() Method

```

```

1 Test Number 000
2 contrastCheckerDriver
3 getContrastRatio()
4 / / / / /
5 Exception: java.lang.NumberFormatException: For input string: "/"
6 Requirement: Given two color objects, return the contrast between
7 them. Handle invalid input exceptions

```

```

1 Test Number 001
2 contrastCheckerDriver
3 getContrastRatio()
4 57 7 130 200 100 250
5 Ratio = 4.360
6 Requirement: Given two color objects, return the contrast between them. Handle invalid input exceptions
7
8 https://webaim.org/resources/contrastchecker/

```

```

1 Test Number 002
2 contrastCheckerDriver
3 getContrastRatio()
4 A 0 0 0 0 0
5 Exception: java.lang.NumberFormatException: For input string: "A"
6 Requirement: Given two color objects, return the contrast between them. Handle invalid input exceptions
7

```

```
1 Test Number 003
2 contrastCheckerDriver
3 getContrastRatio()
4 0 0 0 0 0 0
5 Ratio = 1.000
6 Requirement: Given two color objects, return the contrast between them. Handle invalid input exceptions
7
8 https://webaim.org/resources/contrastchecker/
9
```

```
1 Test Number 004
2 contrastCheckerDriver
3 getContrastRatio()
4 0 0 0 255 255 255
5 Ratio = 21.000
6 Requirement: Given two color objects, return the contrast between them. Handle invalid input exceptions
7
8 https://webaim.org/resources/contrastchecker/
9
```

```
1 Test Number 005
2 contrastCheckerDriver
3 getContrastRatio()
4 256 0 0 0 0 0
5 Exception: java.lang.IllegalArgumentException: Color parameter outside of expected range: Red
6 Requirement: Given two color objects, return the contrast between them. Handle invalid input exceptions
7
8
```

Chapter 3: Testing Framework

Testing Plan implemented with at least 5 test cases.

Framework Description:

Our bash script will initialize to the top level directory where our project is located. Our output HTML page is initialized at the very beginning of the scripts without the closing tags. Then our script will find the desired method we want to test and the driver needed to complete the tests, once these are located the script compiles the Tanaguru method and our driver to the same project directory. Once compiled the bash script reads in the test case(s) correlating to our driver and passes the given arguments to the Java driver. For every test case the script will re-compile the driver and tested method to prevent previous tests contaminating the results. The bash scripts handles the driver outputs to a text file, which is then added to our HTML output page in the form of the table. Once all test text files have been ran through the driver the script adds the closing tags. Then the HTML page, with the CSS style file, will open the output table in a new Firefox tab, the script will wait for the tab to be closed before removing all output files to prevent errors for the next time the script is ran.

How to Run Testing Framework:

1. Install Java on a linux machine.
2. Insure that firefox is installed on the linux machine
3. Clone the project to your machine from our GitHub.
4. The project will clone in a directory named 4Banger
5. Change to the 4banger director, and run the command
sh ./scripts/runAllTests.sh

- The tests will take some time to run, and eventually an output page will be produced that will look like the below image.

The screenshot shows a web browser window with the title '4BANGER TESTING RESULTS'. The browser's address bar shows the file path: file:///home/parallels/Documents/362/4banger/htmlFiles/contrastFinderTest.html. The table below contains the test results.

TEST NUMBER	TESTED METHOD	INPUTS	ORACLES	OUTPUTS	PASS/FAIL
Test Number 000	getContrastRatio()	/////	Exception: java.lang.Number FormatException: For input string: "/"	Exception: java.lang.Number FormatException: For input string: "/"	PASS
Requirement: Given two color objects, return the contrast between them. Handle invalid input exceptions					
Test Number 001	getContrastRatio()	57 7 130 200 100 250	Ratio = 4.360	Ratio = 0.171	FAIL
Requirement: Given two color objects, return the contrast between them. Handle invalid input exceptions					
Test Number 002	getContrastRatio()	A 0 0 0 0 0	Exception: java.lang.Number FormatException: For input string: "A"	Exception: java.lang.Number FormatException: For input string: "A"	PASS
Requirement: Given two color objects, return the contrast between them. Handle invalid input exceptions					
Test Number 003	getContrastRatio()	0 0 0 0 0 0	Ratio = 1.000	Ratio = 1.000	PASS

- Close the browser and the framework will terminate, taking you back to your command line prompt.

Test Cases we implemented:

The test cases we implemented in this deliverable are the 5 we referenced in deliverable 2. All 5 test cases were testing the most important method in our project `getContrastRatio(Color color1, color2)`. Both expected and unexpected inputs were given to the method via our driver.

Issues Encountered / Lessons Learned:

- When compiling our test cases and comparing them to the oracles we received from other sites, there were some rounding issues that caused our test cases to fail. This makes sense due to the comparison we are doing in bash that only passes the test if there is a 100% match. The rounding errors were usually not more than a 1/100th difference in the two ratios. We attribute this error to a combination of rounding by our driver, and by

Tanaguru's methods. We rounded to 2 decimal places to match what other contrast websites did in their comparisons.

Ratio = 4.360

Ratio = 4.362

FAIL

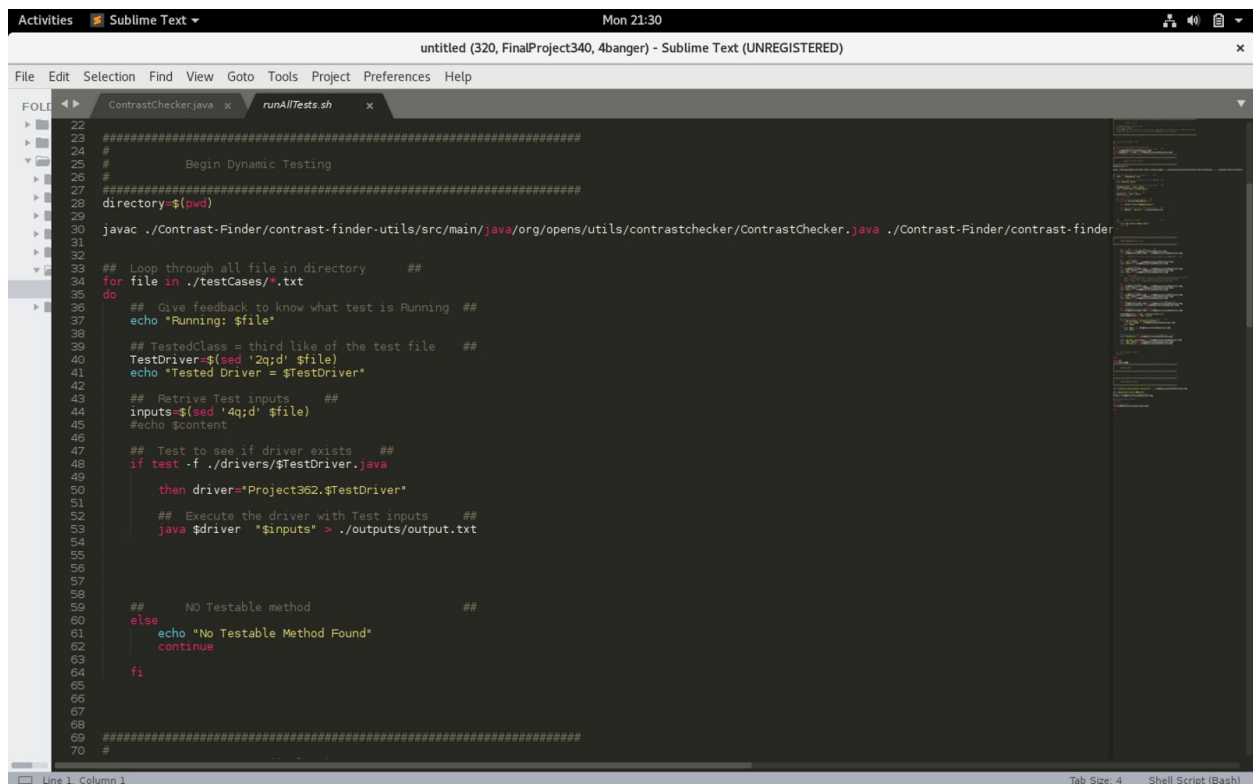
2. To run the method from the Tanaguru code in our driver, we were required to compile both java files in the same bash line and output their class files to one directory. This required learning arguments of the javac command and how to control where they files are output to.
3. Getting the test.txt files to be read into the java driver required learning new bash commands to take specific lines of text files and pass them during the java bash command to run our driver. This process is what lead to our test files being undocumented and instead having a template file that will describe what is happening in each test.

Chapter 4: Implementation of Testing Framework

Changes from deliverable 3:

We were instructed to change the behavior of our bash script from deliverable 3. Previously our bash script had logic that would compare the tested method to methods that were expected. This allowed our script to build each driver as needed per use.

Our current bash script does not have this logic and instead is a fully dynamic approach. Our script will now build all existing drivers in the ./4Banger/Drivers folder, as well as Tanguru's classes for testing. Our new logic will attempt to locate a matching driver for each test case, then execute the driver with the given inputs. Below we have included an excerpt screenshot of this new script:

A screenshot of a Sublime Text editor window. The title bar shows 'untitled (320, FinalProject340, 4banger) - Sublime Text (UNREGISTERED)'. The menu bar includes 'File', 'Edit', 'Selection', 'Find', 'View', 'Goto', 'Tools', 'Project', 'Preferences', and 'Help'. The editor has two tabs open: 'ContrastChecker.java' and 'runAllTests.sh'. The 'runAllTests.sh' tab is active, showing a bash script. The script starts with a comment 'Begin Dynamic Testing' and sets 'directory=\$(pwd)'. It then runs 'javac ./Contrast-Finder/contrast-finder-utils/src/main/java/org/opens/utis/contrastchecker/ContrastChecker.java ./Contrast-Finder/contrast-finder'. A loop 'for file in ./testCases/*.txt' follows. Inside the loop, it echoes 'Running: \$file', sets 'TestDriver=\$(sed '2q;d' \$file)', and echoes 'Tested Driver = \$TestDriver'. It then retrieves test inputs with 'inputs=\$(sed '4q;d' \$file)' and echoes 'content'. A conditional 'if test -f ./drivers/\$TestDriver.java' checks for a driver. If found, it sets 'driver="Project362.\$TestDriver"' and runs 'java \$driver "\$inputs" > ./outputs/output.txt'. If not found, it echoes 'No Testable Method Found' and continues. The script ends with a 'fi' statement and a final comment line.

```
#####
22
23
24 #
25 #   Begin Dynamic Testing
26 #
27 #####
28 directory=$(pwd)
29
30 javac ./Contrast-Finder/contrast-finder-utils/src/main/java/org/opens/utis/contrastchecker/ContrastChecker.java ./Contrast-Finder/contrast-finder
31
32
33 ## Loop through all file in directory ##
34 for file in ./testCases/*.txt
35 do
36     ## Give feedback to know what test is Running ##
37     echo "Running: $file"
38
39     ## TestedClass = third line of the test file ##
40     TestDriver=$(sed '2q;d' $file)
41     echo "Tested Driver = $TestDriver"
42
43     ## Retrieve Test inputs ##
44     inputs=$(sed '4q;d' $file)
45     #echo $content
46
47     ## Test to see if driver exists ##
48     if test -f ./drivers/$TestDriver.java
49     then driver="Project362.$TestDriver"
50
51         ## Execute the driver with Test inputs ##
52         java $driver "$inputs" > ./outputs/output.txt
53
54     else
55         NO Testable method
56     fi
57
58     ## NO Testable method ##
59     else
60         echo "No Testable Method Found"
61         continue
62     fi
63
64
65
66
67
68
69
70
71 #####
72 #
```

Framework Description (Updated):

The updated version of our bash script changes the logic slightly. The script still initializes, clears the terminal, and creates the header of the HTML output page. Next our script builds everything in the ./4Banger/Drivers folder with a .java extension, simultaneously compiling all the Tanaguru classes that are needed for our project. The script will then loop through each .txt file in the ./4Banger/testCases folder reading each line, matching the input method with its driver, passing inputs to that driver, running the driver, and collecting outputs. All outputs are passed to an output text file, which is compared to the oracle line of our test case file, the comparison result is added to the HTML file for final output display. The preceding process is repeated for every test case file, and our script will provide feedback to any test case file that is not properly formatted. After all the tests have been ran, the HTML page is produced in firefox and the program will terminate once the window is closed.

Completion of 25 Test Cases:

All of our test cases were implemented with no major issues with our framework. We were able to use the same test case template and driver template to implement all of the methods, since all the methods dealt with the same type of objects. Outstanding planning ahead of time was credited to the ease of expanding our framework. By selecting methods that all handled similar objects, we were able to simplify the majority of our newly added methods. The only issues that were encountered were rounding issues similar to the ones with our getContrastRatio(Color color1, color2) method.

Chapter 5: Design Fault Injections

Updated How to Run:

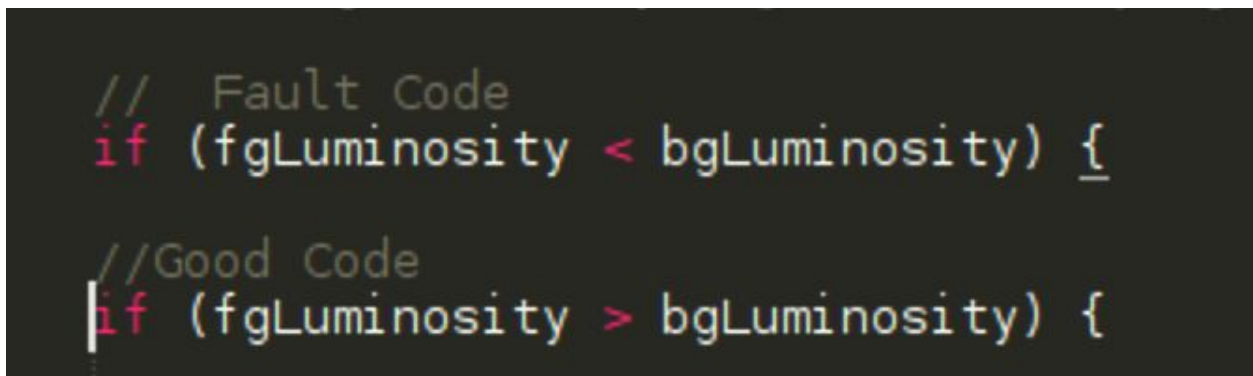
1. Install Java and Git on any debian based linux (Ubuntu) machine.
2. Clone the project to your machine from our GitHub.
3. Select which faults you want to enable in the source code of Contrast-Finder, commenting out the good code (See pictures in Fault Injection).
4. From the 4Banger directory run the command
Sh ./scripts/runAllTests.sh
5. Once the outputs are displayed, close the browser to terminate the framework.

Fault Injection:

1: First method with fault injection was `getContrastRatio(Color color1 color2)`

To trigger an error, we swapped the less than sign to a greater than sign which should invert the ratio between the two colors. The method does not crash due to the fault and behaves as expected.

Notes: The method will pass test cases where all color objects are both the same, and still perform normal exception handling.



```
// Fault Code
if (fgLuminosity < bgLuminosity) {

//Good Code
if (fgLuminosity > bgLuminosity) {
```

2: The next method with injected fault was `getLuminosity(Color color1)`

To create fault in the code, we changed the values of the color object received by `getLuminosity(Color color1)`, by switching the expected Red value with the Expected green value. This should change the ratio of all the colors that are valid passed into the function. The fault code did not crash our framework, and our test notified us that the ratios did not match the oracles.

Notes: The fault code would produce flawed outputs compared to our oracles, and exceptions would still be handled.

```
/**
// Fault code method: Uncomment this method to create fault
// instead of RGB its GRB

public static double getLuminosity(Color color) {
    double luminosity =
        getComposantValue(color.getGreen()) * RED_FACTOR
        + getComposantValue(color.getRed()) * GREEN_FACTOR
        + getComposantValue(color.getBlue()) * BLUE_FACTOR;

    return luminosity;
}
/**/

//Good Code Method
public static double getLuminosity(Color color) {
    double luminosity =
        getComposantValue(color.getRed()) * RED_FACTOR
        + getComposantValue(color.getGreen()) * GREEN_FACTOR
        + getComposantValue(color.getBlue()) * BLUE_FACTOR;

    return luminosity;
}
```

3: The next method with injected fault was the `getBrightness(Color color1)`

To get the code to produce incorrect outputs, we altered the order that the color values for RGB were read into the function, similar to get luminosity we replaced the expected Red value with the expected Green value. This method still ran through our testing framework and produced unexpected outcomes that did not match our desired outcomes. Still this did not break

our system, and the changes were monitored.

```
public static Float getBrightness(Color color) {
    float[] hsbValues = new float[MAX_COMPONENT];
    Float brightness;

    //Fault Code
    Color.RGBtoHSB(color.getGreen(), color.getRed(), color.getBlue(), hsbValues);

    //Good Code
    Color.RGBtoHSB(color.getRed(), color.getGreen(), color.getBlue(), hsbValues);

    brightness = hsbValues[BRIGHTNESS];
    return brightness;
}
```

4: The next method with injected fault was getSaturation(Color Obj)

To get this method producing unexpected outputs we repeated similar steps from the getBrightness() method and altered the positions that it receives the RGB values from the color object, again switching the place of the Red and Green values. Similar results were produced and did not match our oracles. This did not crash our framework, and our test cases were able to pick up on the unexpected outputs during run time.

```
public static Float getSaturation(Color color) {
    float[] hsbValues = new float[MAX_COMPONENT];
    Float saturation;

    //Fault Code
    Color.RGBtoHSB(color.getGreen(), color.getRed(), color.getBlue(), hsbValues);

    //Good Code
    Color.RGBtoHSB(color.getRed(), color.getGreen(), color.getBlue(), hsbValues);

    saturation = hsbValues[SATURATION];
    return saturation;
}
```

5: The final method with injected fault was rgb2Hex(Color Obj)

To fault this method we changed the order in which the RGB values are passed into the function. Similar to other fault injection this did not crash our framework and produced outputs that did not match our oracles.

```
public static String rgb2Hex(Color color) {  
    //Fault Code  
    //return (String.format("#%02x%02x%02x", color.getGreen(), color.getRed(), color.getBlue())).toUpperCase();  
  
    //Good Code  
    return (String.format("#%02x%02x%02x", color.getRed(), color.getGreen(), color.getBlue())).toUpperCase();  
}
```

Fault Injection Results:

The fault injection verified that our system is robust and can operate without the Color-Contrast-Finder functioning the way it is supposed to. As long as the tested methods accept and return valid objects then our testing framework will execute as expected. This experiment confirmed that our script and drivers were constructed properly and independently function from Tanaguru's project. As a team we were able to construct a robust testing framework that focused on the functional requirements of Tanaguru's Contrast-Finder web application.

Chapter 6: Lessons Learned

Dealing open source projects is difficult due to lack of instructions and updated readme files. We almost decided against Tanaguru due to our early difficulties simply trying to compile the code. The instructions for compiling and running the code were outdated. It took some a few days of trial and error to run the code they way Tanaguru had intended.

Once we got Tanaguru working, the main problem we had was the bash script. It took a long time to get initially working, and then it was harder to transition from a static to a dynamic style script, meaning with the static script we had hard coded values of which drivers exist, with the dynamic it couldn't know anything about the project at all. Although the dynamic did make it easier in the long run, it was the coding challenge. After the got the script working, which took a while, it was a simple matter of tweaking the test cases , following the correct test case format, and doing the error injections.

The value of robust, validated, and specifically designed tests. While testing Tanaguru we immediately noticed the importance of specifying tests that should behave in a particular manner. The reason we noticed this is that when we passed inputs to our fault code the behavior of our framework did not change, however the results became vastly different from what we were expecting. If anywhere in our project we had not prepared our oracles for the behavior of the code then many more of our test cases would have failed. We were very fortunate that the logic of our framework did not interfere or inhibit the logic of Tanaguru's methods, and our test results behaved as expected.

Our team worked together well, but having a group of four made it very hard to meet. There was a miscommunication and doing all the work seperate did give us problems. We learned having a big group is not always better, and hopefully it will be easier in a workplace environment. A team going to the same office every day. We feel, we gained some valuable experience in software engineering. As well as learned new skills, such bash scripting and refined others like git. All in all, we have taken a lot away from this course and hope to be able to apply it in the future to real world testing applications.

In Conclusion this group project has been beneficial to everyone involved and our understanding of the software development process, specifically testing software, has grown exponentially. This project has given members of our team insights into the structure of open source projects and allowed us to use and test a large scale industry application. The experiences we have gained have been invaluable and we are more effective software engineers because of it.