Developing an Automated Testing Framework for Celestia

Guilherme Costa, Megan Landau, and Tony Tang
Computer Science Department, College of Charleston

Guilherme Costa
costagr@g.cofc.edu
GitHub: gui529

Megan Landau
landaumd@g.cofc.edu
GitHub: landaumd

Tony Tang
tktang@g.cofc.edu
GitHub: ttang725

ABSTRACT

Testing code efficiency and accuracy is one of the most important practices in programming. By building and using an automated testing framework, a user can uncover failures within code quickly. This work shows the benefits of implementing an automated testing framework on the math library of Celestia, an open source space simulation application.

INTRODUCTION

Testing code is a crucial practice for maintaining quality in software. By implementing an automated testing framework, code can be evaluated autonomously and often, ensuring the dependability and functionality of new code and saving time in the process. The benefits of building automated testing has been discussed in recently published works. With this century's increased use of technology and the widespread availability of the internet, there has been a rise in the sharing and distribution of open source projects. Services such as GitHub that provide version control have allowed for many authors to contribute to single projects. With this, however, there is the challenge of maintaining software quality and reliability when so many separate parties are involved. The proliferation of open source projects has led to a discussion of the importance in maintaining code quality in software by using automated testing frameworks [Kochhar et. al., 2013]. There are many benefits associated with using this method over manual testing. Automated testing is considered valuable because of the associated reduction in cost of the testing stage of production, decrease in the length of testing regression cycles, and the overall faster release of the software to market [Day, 2014]. In open source development, implementing an automated testing framework could ensure that new code never harms the working of old code and that developers working in tandem can always check the quality of the program as they make new commits. A framework that is designed to be reusable and incorporate well-defined design patterns will execute smoothly every time and be an invaluable resource for any group of software developers, whether it is for open source project testing or not [Shuangzhou, 2014]. By developing a wide range of test cases with an evaluation of the results, the automated testing framework will be able to demonstrate the overall quality of the code and highlight any weaknesses for future programming.

This work aims to validate the claim that automated testing frameworks increase the visibility of errors in code. The Humanitarian / Free Open Source Software chosen for research was Celestia, a space simulation application that has been in development since the early 2000s. Celestia uses a custom math library to perform calculations during run time. By developing an automated testing framework that utilizes appropriate test cases, the strengths and weaknesses in Celestia's math library can be exposed easily by comparing expected outcomes to a method's actual outcome. Furthermore, the validity of the testing framework can be demonstrated by injecting fault into the code and then running the framework again: the faults in the new code will be immediately apparent as the pass-fail ratio will be altered dramatically.

The practice of implementing automated testing frameworks is not new in the domain of software engineering, but the value in it cannot be overstated.

METHODS

1. Set up the folder hierarchy in the TestAutomation folder as specified in the Team Term Project Specifications document provided and use "git push" to save this set up to the Team repository on GitHub
2. Clone and download the repository to an Ubuntu Linux machine (Virtual Machines can be used as well) using the command line and "git pull"
3. Choose 5 methods from the Celestia math library that will be tested using the automated testing framework. The 5 methods chosen were: square, cube, circleArea, sphereArea, and radToDegree because their outcomes could be calculated using non-Celestia resources
4. Write 25 test cases to evaluate the methods (5 for each method). All test cases text files must be saved in the testCases folder

```
# Test cases are written in the following format:
#       1. Test Case ID
#       2. Requirement being tested
#       3. Component being tested
#       4. Method being tested
#       5. Driver being used
#       6. Test input(s)
#       7. Expected outcome
```

```
/TestAutomation
    /project
        /src
        /bin
        / …
    /scripts
        runAllTests.(some scripting extension)
        … other helper scripts
    /testCases
        testCase1.txt
        testCase2.txt
        …
    /testCasesExecutables
        testCase1(may be folder or file)
        …
    /temp (for output from running tests … be sure to clean at start of runAllTests)
        testCase1results (might be folder or file)
        …
    /oracles
        testCase1Oracle (might be folder or file)
        …
    /docs
        README.txt
    /reports
        testReport.(txt or html)
```

Figure 1: Framework Directory Structure

5. For each method, create a .cpp driver which will take the input from the test case file and return the result of the math library method. Save this driver in the testCasesExecutables folder
6. Save the source file of the "mathlib.h" file from the Celestia repository in the testCasesExecutables folder as well
   Ensure that each driver can be compiled and run from the command line and successfully return an output value
7. Write a script in BASH that parses a testCase.txt file and compiles the specified method's driver. For each test case, the script must make a variable out of the input value and then pass this variable to the compiled driver. Save this script in the scripts folder
8. The script will receive the return value back from the driver and store this value into a variable
9. The script will compare the received actual output with the testCase.txt file's expected output and create a new variable with the assigned value of "Pass" or "Fail" with that test case
10. The script will use a loop to iterate through all of the testCase.txt files found in the testCases folder of the framework, performing steps 5 - 7 for each test case
11. The script will output all the stored variables read from the testCase.txt file and the results of the test to an .html report, which is saved in the reports folder
12. The script will run a loop that appends multiple rows to a table in the .html report
13. The script will populate the table with actual values ascertained from running the test case
14. The results of the tests will be appended to the .html report.
15. ./TestAutomation/scripts/runAllTests.sh will open a browser window displaying the following in columns:
    - Test Case ID
    - Method
    - Input
    - Expected Output
    - Actual Output
    - Pass or Fail Outcome
16. Note the ratio or Pass outcomes to Fail outcomes
17. Inject faults into the existing "mathlib.h" source code. Change the code in the methods being evaluated. This can be done by simply changing a multiplication operator to a division operator and vice versa. Change any operator to another operator to "break" the code
18. Run the script again with the fault injections
19. Note the changes in the testing results table before and after the fault injections

RESULTS

Below are the test results tables as the are displayed in the page after being generated by running the runAllTests.sh Bash script. The final column indicates the Pass or Fail status of the test case.

Test Case ID	Method	Input	Expected Output	Actual Output	Pass / Fail
001	square	2	4	4	Pass
002	square	10	100	100	Pass
003	sphereArea	5	314.159	314.159	Pass
004	cube	8	512	512	Pass
005	radToDeg	0	0	0	Pass
006	sphereArea	2.05	52.8102	52.8102	Pass
007	cube	-2	-8	-8	Pass
008	circleArea	-5.0	error	78.5398	Fail
009	radToDeg	5	286.479	286.479	Pass
010	cube	876	672221376	6.72221e+08	Fail
011	circleArea	0.0	0	0	Pass
012	sphereArea	0	0	0	Pass
013	square	-98209	9645007681	1055073089	Fail
014	square	800000	640000000000	49872896	Fail
015	square	0	0	0	Pass
016	cube	-1024	-1073741824	-1.07374e+09	Fail
017	cube	2.1111	9.4088	9.40863	Fail
018	sphereArea	100000	1.25664e+11	1.25664e+11	Pass
019	sphereArea	-300.05	error	1.13135e+06	Fail
020	circleArea	10	314.159	314.159	Pass
021	circleArea	155555555555	7.6e+22	7.60188e+22	Fail
022	radToDeg	1	57.2958	57.2958	Pass
023	radToDeg	-57	-3265.86	-3265.86	Pass
024	circleArea	500	785398	785398	Pass
025	radToDeg	125000000	7161972439.15	7.16197e+09	Fail

Figure 2: Test results before fault injections. The ratio of Pass to Fail is 16:9, 16 test cases resulted in Pass and 9 test cases resulted in Fail before the fault injections were placed. 64% of test cases passed before fault injections.

Test Case ID	Method	Input	Expected Output	Actual Output	Pass / Fail
001	square	2	4	4	Pass
002	square	10	100	100	Pass
003	sphereArea	5	314.159	314.159	Pass
004	cube	8	512	512	Pass
005	radToDeg	0	0	0	Pass
006	sphereArea	2.05	52.8102	52.8102	Pass
007	cube	-2	-8	-8	Pass
008	circleArea	-5.0	error	78.5398	Fail
009	radToDeg	5	286.479	0.00884194	Fail
010	cube	876	672221376	6.72221e+08	Fail
011	circleArea	0.0	0	0	Pass
012	sphereArea	0	0	0	Pass
013	square	-98209	9645007681	1055073089	Fail
014	square	800000	640000000000	49872896	Fail
015	square	0	0	0	Pass
016	cube	-1024	-1073741824	-1.07374e+09	Fail
017	cube	2.1111	9.4088	9.40863	Fail
018	sphereArea	100000	1.25664e+11	1.25664e+11	Pass
019	sphereArea	-300.05	error	1.13135e+06	Fail
020	circleArea	10	314.159	314.159	Pass
021	circleArea	155555555555	7.6e+22	7.60188e+22	Fail
022	radToDeg	1	57.2958	0.00176839	Fail
023	radToDeg	-57	-3265.86	-0.100798	Fail
024	circleArea	500	785398	785398	Pass
025	radToDeg	125000000	7161972439.15	221049	Fail

Figure 3: Test results after a fault injection where the radToDegrees method was altered. The ratio of Pass to Fail is 13:12. 52% of test cases passed after this fault injection, which demonstrates a 12% decrease in successful functionality of the Celestia math library.

Test Case ID	Method	Input	Expected Output	Actual Output	Pass / Fail
001	square	2	4	4	Pass
002	square	10	100	100	Pass
003	sphereArea	5	314.159	31.831	Fail
004	cube	8	512	512	Pass
005	radToDeg	0	0	0	Pass
006	sphereArea	2.05	52.8102	5.35079	Fail
007	cube	-2	-8	-8	Pass
008	circleArea	-5.0	error	78.5398	Fail
009	radToDeg	5	286.479	286.479	Pass
010	cube	876	672221376	6.72221e+08	Fail
011	circleArea	0.0	0	0	Pass
012	sphereArea	0	0	0	Pass
013	square	-98209	9645007681	1055073089	Fail
014	square	800000	640000000000	49872896	Fail
015	square	0	0	0	Pass
016	cube	-1024	-1073741824	-1.07374e+09	Fail
017	cube	2.1111	9.4088	9.40863	Fail
018	sphereArea	100000	1.25664e+11	1.27324e+10	Fail
019	sphereArea	-300.05	error	114630	Fail
020	circleArea	10	314.159	314.159	Pass
021	circleArea	155555555555	7.6e+22	7.60188e+22	Fail
022	radToDeg	1	57.2958	57.2958	Pass
023	radToDeg	-57	-3265.86	-3265.86	Pass
024	circleArea	500	785398	785398	Pass
025	radToDeg	125000000	7161972439.15	7.16197e+09	Fail

Figure 4: Test results after a fault injection where the sphereArea method was altered. The ratio of Pass to Fail is 13:12. 52% of test cases passed after this fault injection, which demonstrates a 12% decrease in successful functionality of the Celestia math library.

CONCLUSION

Using automated testing is an expedient way to determine defects in Celestia's original source code. Celestia's math library uses a specified value of Pi, four-point decimal precision, and an uncertain determination of when to use exponential notation. All of these factors could have contributed to any resulting failures during testing. Small changes made to Celestia's code resulted in a decrease in pass rates for the math library. This work demonstrates the importance of repeatedly running automated tests on a program to determine if any changes in the code will produce unexpected results in the functionality of the program. The automated testing framework developed here can decrease the amount of time spent debugging or questioning programming decisions and can inevitably increase the visibility of program failures when defective code is injected into the program.

REFERENCES

1. Day, P. (2014). n-Tiered Test Automation Architecture for Agile Software Systems. *Procedia Computer Science*, 28 (2014 Conference on Systems Engineering Research), 332-339.
2. Kochhar, P. S., Bissyande, T. F., Lo, D., & Jiang, L. (2013). Adoption of Software Testing in Open Source Projects - A Preliminary Study on 50,000 Projects. *2013 17th European Conference On Software Maintenance & Reengineering*, 353.
3. Shuangzhou, G. (2014). Research on Automatic Testing Framework. *Applied Mechanics & Materials*, (635-637), 1594.

ACKNOWLEDGEMENTS

We would like to thank Dr. James Bowring for his valuable criticism and advice given over the course of this project.