Calculating Pidgins

Team TBD

Walter Blair, Matthew Claudon, Nicholas Johnson

CSCI 362, Fall 2016

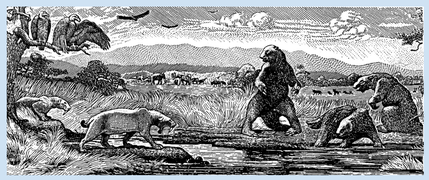
Introduction

Most of us got into this field because we enjoy coding, and we've all made it this far using the classic stop, drop, and code routine. This class changes everything! This semester of software engineering has given us the opportunity to see firsthand how planning, structure, documentation, and systematic testing are critical components of our work as computer scientists and software developers. The task this semester was to undertake a project that would put a real open source project to the test.

The objective of this project was to create an automated testing framework for any Open Source project of our choosing that can be run from a Linux machine and whose method can be called from the command line. The Testing Framework is applied to 25 test cases and 5 fault injections.

The overall purpose of the project was to give us experience in a real world application of testing and to introduce one way of accomplishing the task of testing a large program.

Our project began with an ambitious effort with an open-source instant messaging platform called Pidgin and wound its way to a much simpler, better documented, and more testable calculator written in Java. We have successfully created an automated testing framework that tests a number of basic calculator functions and detects injected faults as expected.



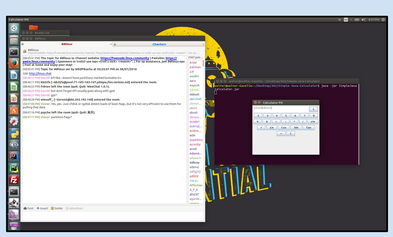
Frederick P. Brook's tarpit of system-less software development

Pidgin & Simple-Java-Calculator

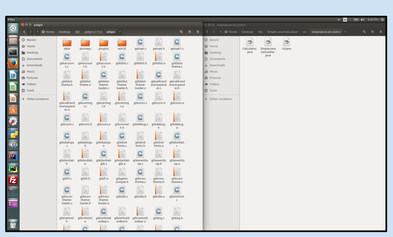
Pidgin is an open-source platform for messaging across a number of different protocols used by Facebook, AIM, and other popular instant messaging networks. A number of 3rd party plugins add networks like Twitter, Skype, and many more.

With Pidgin we took on a project that was far beyond our abilities and the amount of time we have to put into it, but we did learn some interesting things from our attempt to create a working test driver for it.

We decided to move to a new project with a good balance of functionality and accessibility. The ph7-Simple-Java-Calculator has just three Java files, one that defines the GUI, a second that defines the underlying Calculator, and a third that instantiates the GUI calculator. For this project we are ignoring the UI and simply testing the underlying Calculator class that contains computational methods.



Example sessions of Pidgin and the Java Simple Calculator



Comparison of Pidgin and Calculator source complexity

Automated Testing Framework

Our automated testing framework contains the following components:

- scripts/runAllTests.sh is a shell script that compiles and executes each testCase.java driver with the input specified in each testCase.txt file passed as a command-line arg. The script initializes the html report, reads the testCase.txt file for the appropriate driver and input, executes the driver, and then pipes the rounded output to the report before displaying it in a browser.
- Each testCase.java driver takes a Double as a command line arg and then executes its Calculator method, passing the specified inputs as parameters. The driver returns the output to the script.
- Each testCase.txt file contains the details of each test case.

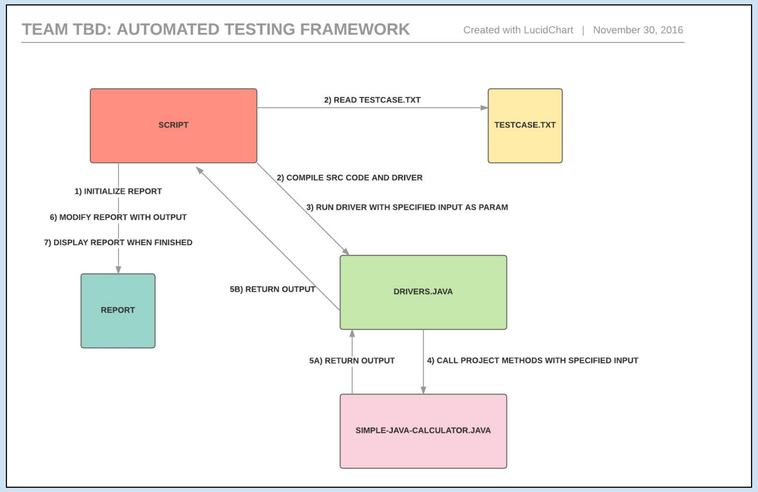
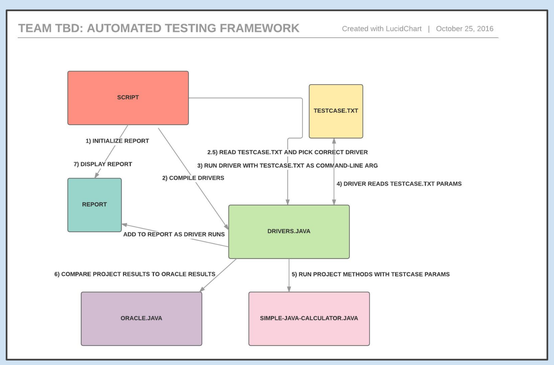


Diagram of our automated testing framework. The bash script initializes the report and begins by reading the first test case. The script compiles the specified driver and passes the specified input to the driver as a command line arg. The driver calls the source method and returns the output to the script. The script adds the results to the report and continues the process with all of the remaining test cases. After all cases are run, the report is finished and displayed in a browser.

We learned a great deal while working through the structure of our testing framework. We initially created the design shown below, where the Java driver did all of the hard work. This project helped us get out of our Java comfort zones and run with Bash scripting.

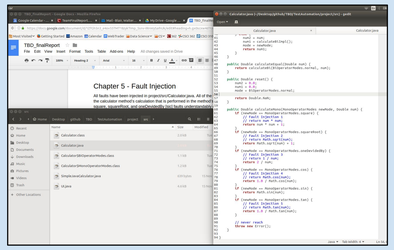


Our original testing framework required the Java driver to do too much work and was revised after presentation feedback.

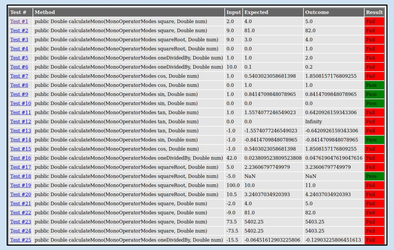
Fault Injection & Reporting

All faults were injected in project/src/Calculator.java. All of the faults involve changing the calculator method's calculation that is performed in the method's return statement. The square, squareRoot, and oneDevidedBy [sic] faults understandably cause all related test cases to fail, but the replacement of cosine with secant and tangent with cotangent leave some room for test cases to potentially pass.

We didn't get too adventurous in our fault injections, so we finished them quickly without much trial and error. The only fault we thought of injecting that didn't succeed was a correction of the misspelling of the 'oneDevidedBy' method. Correcting the spelling resulted in a failure to compile, because this variable was not declared. If we corrected the spelling in the declaration, then of course the spelling would be fixed and there would be no fault introduced. After this run-in with a compile-time error, it seemed clear that our calculator would only have a few possible runtime errors to choose from for our fault injection.



Faults injected into the Calculator.java source file.



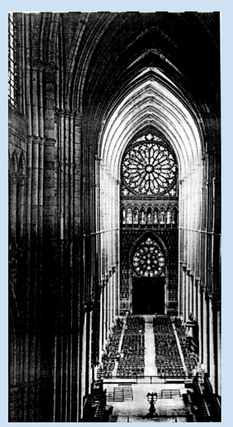
Test #	Method	Input	Expected	Outcome	Result
Test #1	public Double calculateMonoOperatorModes square, Double num)	2.0	4.0	5.0	Fail
Test #2	public Double calculateMonoOperatorModes square, Double num)	9.0	81.0	82.0	Fail
Test #3	public Double calculateMonoOperatorModes squareRoot, Double num)	9.0	3.0	4.0	Fail
Test #4	public Double calculateMonoOperatorModes squareRoot, Double num)	0.0	0.0	1.0	Fail
Test #5	public Double calculateMonoOperatorModes oneDividedBy, Double num)	1.0	1.0	2.0	Fail
Test #6	public Double calculateMonoOperatorModes oneDividedBy, Double num)	10.0	0.1	0.2	Fail
Test #7	public Double calculateMonoOperatorModes cos, Double num)	1.0	0.5403023058681398	1.8508157176809255	Fail
Test #8	public Double calculateMonoOperatorModes cos, Double num)	0.0	1.0	1.0	Pass
Test #9	public Double calculateMonoOperatorModes sin, Double num)	1.0	0.8414709848078965	0.8414709848078965	Pass
Test #10	public Double calculateMonoOperatorModes sin, Double num)	0.0	0.0	0.0	Pass
Test #11	public Double calculateMonoOperatorModes tan, Double num)	1.0	1.5574077246549023	0.6420926159343306	Fail
Test #12	public Double calculateMonoOperatorModes tan, Double num)	0.0	0.0	Infinity	Fail
Test #13	public Double calculateMonoOperatorModes tan, Double num)	-1.0	-1.5574077246549023	-0.6420926159343306	Fail
Test #14	public Double calculateMonoOperatorModes sin, Double num)	-1.0	-0.8414709848078965	-0.8414709848078965	Pass
Test #15	public Double calculateMonoOperatorModes cos, Double num)	-1.0	0.5403023058681398	1.8508157176809255	Fail
Test #16	public Double calculateMonoOperatorModes oneDividedBy, Double num)	42.0	0.023809523809523808	0.047619047619047616	Fail
Test #17	public Double calculateMonoOperatorModes squareRoot, Double num)	5.0	2.23606797749979	2.2360679774997 9	Fail
Test #18	public Double calculateMonoOperatorModes squareRoot, Double num)	-5.0	NaN	NaN	Pass
Test #19	public Double calculateMonoOperatorModes squareRoot, Double num)	100.0	10.0	11.0	Fail
Test #20	public Double calculateMonoOperatorModes squareRoot, Double num)	10.5	3.2403703492020393	4.2403703492020393	Fail
Test #21	public Double calculateMonoOperatorModes square, Double num)	-2.0	4.0	5.0	Fail
Test #22	public Double calculateMonoOperatorModes square, Double num)	-9.0	81.0	82.0	Fail
Test #23	public Double calculateMonoOperatorModes square, Double num)	73.5	5402.25	5403.25	Fail
Test #24	public Double calculateMonoOperatorModes square, Double num)	-73.5	5402.25	5403.25	Fail
Test #25	public Double calculateMonoOperatorModes oneDividedBy, Double num)	-15.5	-0.06451612903225806	-0.12903225806451613	Fail

The report displays results of tests after fault injection.

Lessons & Applications

We found that Open Source projects can be tricky to get involved in. They aren't all as open as one might think, so it is important to do a fair amount of research about a project before trying to jump into it. Our most frustrating experience with Pidgin was that they never responded to our email, and their code was very poorly documented. It was far too complicated for us to figure out how to build a testing framework for such a project. Once we switched over to the new project and started to build the framework for the simple calculator it was much easier to see what this class was all about. Below are some specific lessons that we have taken away from this project:

- Documentation is super important.
- Making an effective testing framework for software is not always a simple task and can take a large amount of time and resources.
- Testing is valuable beyond just knowing if a program works or not, it gives better data on the small parts of a program so that improvements can be made.
- The Linux operating system is complex and has a lot of options to work with, especially when trying to manually locate and install dependencies while building a project.
- Open source software isn't always completely open or easy to get involved with.
- A beautiful cathedral doesn't get built in a day, and neither do successful software projects. We learned how to apply our fun coding skills in a structured and productive engineering environment!



Frederick P. Brook's cathedral doesn't get built overnight!