Team TBD

CSCI 362, Fall 2016

Final Report

Walter Blair

Matthew Claudon

Nicholas Johnson

Table of Contents

Preface

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.

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.

In this report, our initial exploration of the open-source instant-messaging platform Pidgin is described in Chapter 1. Our attempts to work through the obstacles that arose as well as our decision to move to a different project is detailed in Chapter 2. The development of our testing framework is explained in Chapters 3 & 4, and the application of our framework to identify faults that we injected is demonstrated in Chapter 5. Our final thoughts and reflections are expressed in Chapter 6.

Quick Start Guide

Testing Instructions:

1. Clone the project from the Team TBD repository.
2. Navigate to the *TestAutomation* folder.
3. Run *./scripts/runAllTests.sh*

Open Source Project:

1. Clone the project from ph7-Simple-Java-Calculator
2. To run, navigate to project directory and type <java -jar SimpleJavaCalculator.jar>

Chapter 1

Overview

Pidgin, similar to our original H/FOSS candidate Empathy, is an open-source platform for messaging across a number of different protocols used by Facebook, AIM, and other popular instant messaging networks. Pidgin is based off of another IM messaging service called Jabber. A number of 3rd party plugins add networks like Twitter, Skype, and many more. There is currently no mobile app for Pidgin, though there is a portable app. Apple's development restrictions and the giant leap of porting to Android are significant obstacles to mobile development.

https://pidgin.im/

https://developer.pidgin.im/

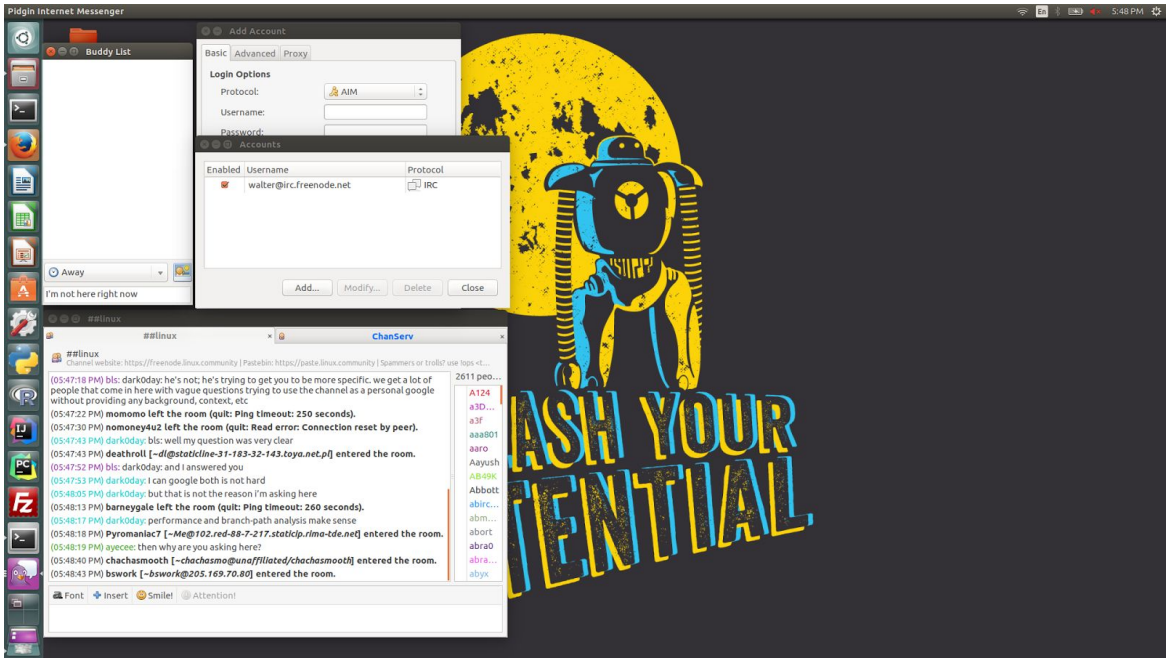


Figure 1. A typical Pidgin session in the ##linux IRC channel, though other networks could be engaged simultaneously as desired.

Application Tests

We successfully compiled Pidgin in Ubuntu and have tested built-in IM services for Facebook and IRC as well as 3rd party plugins for classic Battle.net and Skype.

Early updates:

- IRC is working very well. It's easy to login and search/browse channels.
- Connecting to Facebook Chat is smooth, and the IM feature works beautifully. It's initially annoying that contact names pop up all over the Desktop, but the functionality is good.
- We've tested a few 3rd party plugins at this point. So far we have installed classic Battle.net and Skype but haven't been able to log in to Battle.net without a CD key for one of the older games, and we're troubleshooting Skype's failed SSL handshake. We plan to try the Twitter plugin next, but Twitter's previous ban on clients and the lack of updates for this plugin in the last two years doesn't make this look promising. Ran it with google talk, seems to be working.
- Tried MySpaceIM and Yahoo chat, both have issues connecting. Yahoo refuses any username and password regardless if the credentials are correct or not. MySpace seems to connect, but just says "Connecting…" until a timeout occurs. Theorizing that Pidgin may not be the cause as MySpace is so obsolete the MySpaceIM servers may no longer be up. Yahoo is just broken.

Update: As we were getting started with the project, we recognized that Pidgin clearly isn't the most popular platform in the world for accessing various instant messaging networks. Nevertheless, it certainly seemed functional and relatively up-to-date for an open-source project. It was a bit concerning that networks like Facebook were clearly trying to disable 3rd party access by applications like Pidgin, but this seemed more like a long-term issue rather than something that would be problematic during the semester.

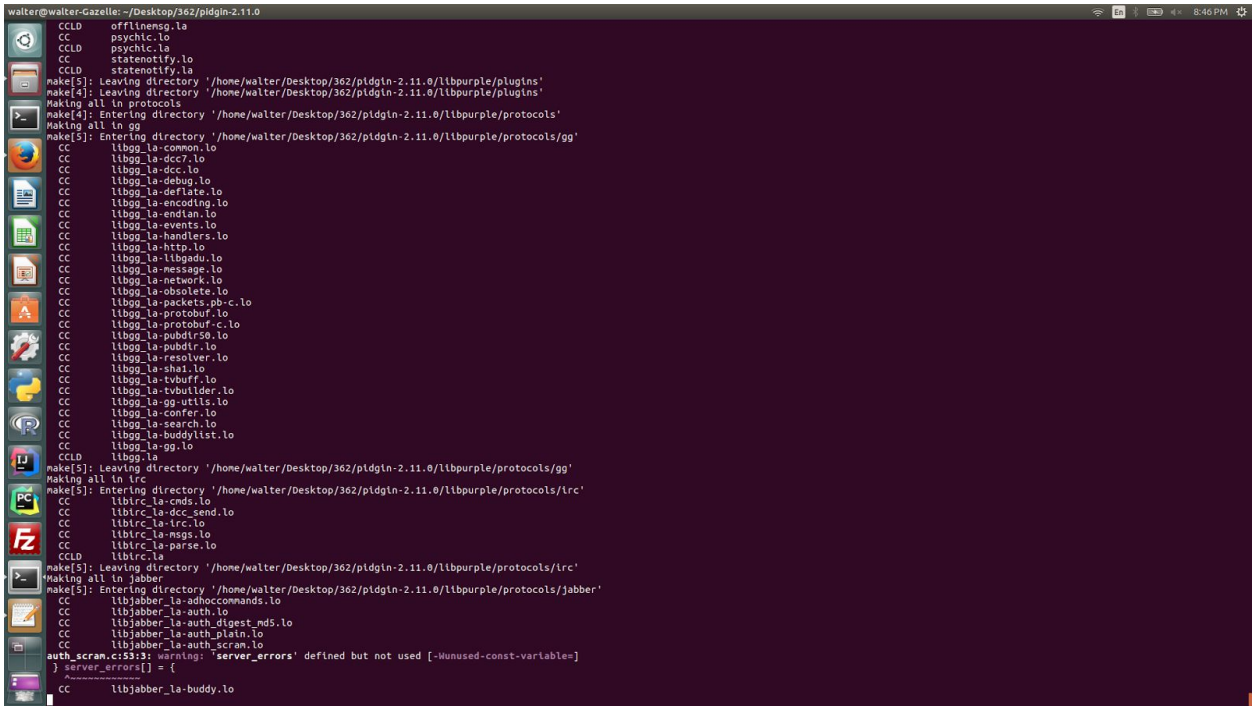


Figure 2. Snapshot of console while executing ‘make install’ in the pidgin source directory. To build Pidgin, a configure script points the user to various C language library dependencies like glib. When the configure script runs successfully, the user is ready to execute the makefile as seen above.

Source Code Exploration

We started to dive into the source code by looking at pidgin/test-driver.sh and the pidgin/libpurple/tests folder that we thought might contain validation test suites for us to run. Not understanding what these files were initially, we emailed the pidgin development team on 9/9 for assistance and started working through the test-related files described below. We never received a response from the developers, and it now seems to us that these are routine tests that the program runs during operation to check basic utilities, IM protocols, and encryption methods.

Update - Below is a record of our attempts to understand what each of the test-related files do. During this period we were trying improve our understanding of bash scripting and the C language. When we started out, we were not at all surprised that we didn’t understand how these scripts and C files worked. As we gained familiarity bash and C, we started to realize that the files were not intended for testing during development but rather are used routinely by the program during start-up and various activities like encrypting messages and establishing network connections over various protocols.

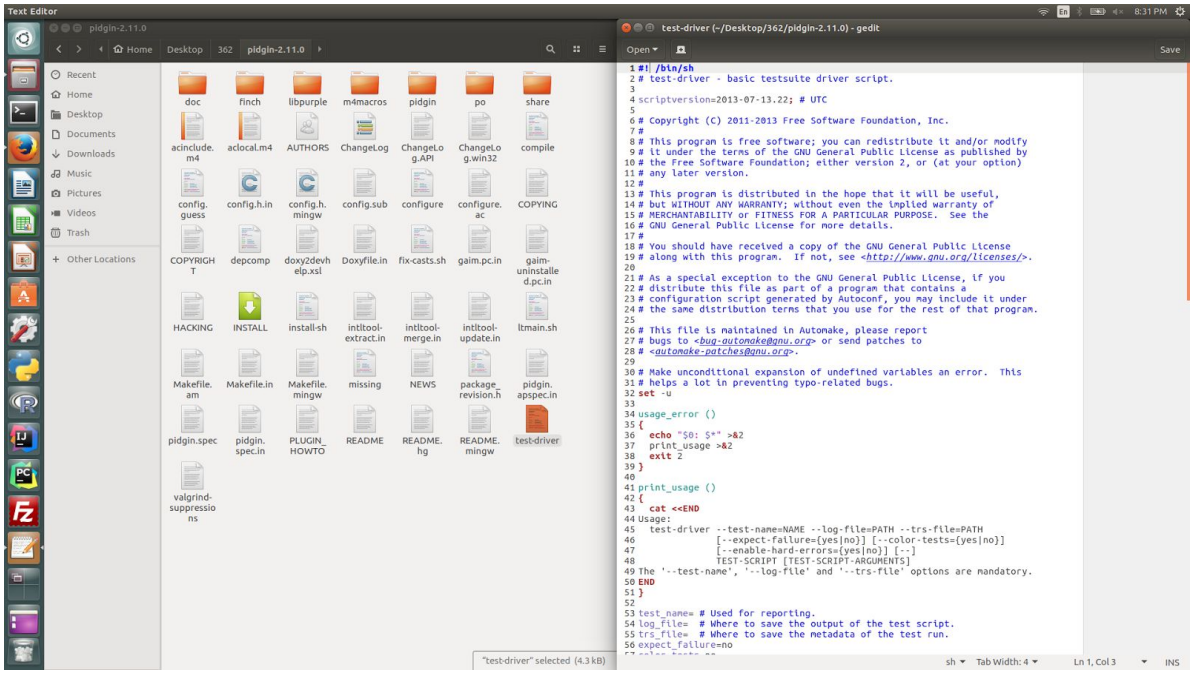


Figure 3. The top-level directory of the pidgin source folder showing the top-level test-driver.sh script file location and introductory comments and code.

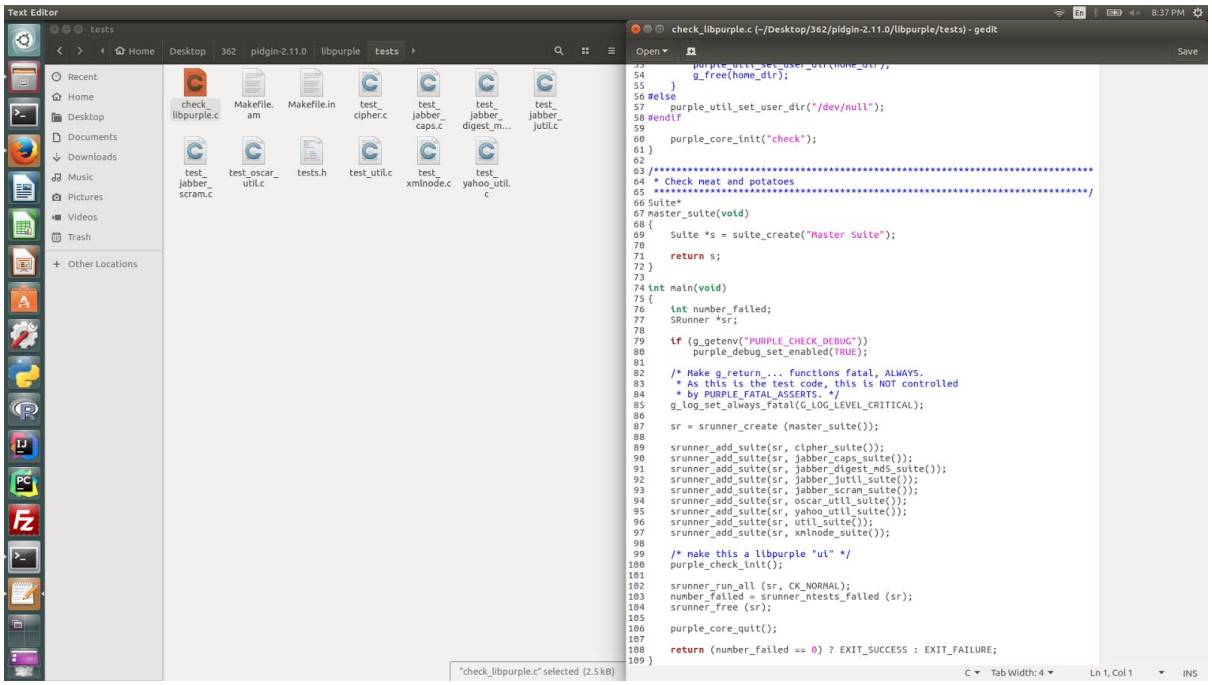


Figure 4. The libpurple/tests directory containing a number of test.c drivers for testing various components.

Summaries of test-related files:

pidgin/

- test-driver.sh
    - This is the driver included to make running tests easier. The command takes three parameters that are the test name, log file path, and trs file path. The test-name is the name of the test which is mostly meant to be used in console reports about the test suite.The log and trs paths are used for outputting the results and data acquired by the tests. There are other optional parameters as well that colorize the terminal output, determine if a failure is expected, or decide how hard errors are treated versus normal errors.

pidgin/libpurple/tests/

- Makefile.am
    - Makefile.am = automaker of other make files.
    - http://stackoverflow.com/questions/2531827/what-are-makefile-am-and-makefile-in
- test_jabber_*
    - It would appear that this project is using something called the Jabber test Suite which is an old open source testing framework that has been very hard to find information about, but it would appear that the Jabber testing framework was built to test an old IM messaging system that Pidgin is built off of. The main purpose of this particular testing suite is to stress test the server capacities.
    - http://www.drdobbs.com/stress-testing-jabber-with-the-jabber-te/199101609
    - http://jabbertest.sourceforge.net/
- test.h
    - Header file that relates all the test suites.
- check_libpurple.c
    - Appears to be a test suite of the other test suites. Could not run as it has a missing dependency that is “glib.h”.
- test_cipher.c
    - Tests following encryption methods as part of Cipher Suite:
        - MD4 - Message Digest algorithm
        - MD5 - replacement for MD4 (both now obsolete)
        - SHA1 - Secure Hash Algorithm 1

        - SHA256 - 256bit hash
        - DES - Data Encryption Standard
        - DES3 ECB - Triple DES Electronic CookBook
        - DES3 CBC - Triple DES Cipher-Block Chaining
        - HMAC - hash message authentication code
- test_util.c
    - Utility Functions suite:
        - Base 16 and 64 data encoding
        - Invalid file names
        - Invalid email addresses
        - Invalid IP addresses
        - html to xml markup
        - text strip mnemonic?
        - utf8 strip unprintables?
        - strdup_withhtml?
- test_xmlnode.c
    - Testing for billion laughs DOS attack that targets xml parsers with haha's.
- test_oscar_util.c
    - OSCAR (Open System for CommunicAtion in Realtime) is AOL's proprietary protocol for AIM (and ICQ before they sold it), but lots of third parties have reverse engineered it.
    - Oscar Utility Functions suite:
        - Testing the name input for using protocol?
- test_yahoo.util.c
    - Yahoo IM's proprietary protocol.
    - Yahoo Utility Functions suite:
        - Testing the name input for using protocol?

Chapter 2 - Rise & Fall of Pidgin

The Testing Process

The ultimate goal of our team term project is to develop an automated testing framework for Pidgin and implement the framework with twenty-five test cases, five injection faults, and plenty of documentation. The project is incremental, as we work toward our completed test suite through several chapters described below. While Pidgin is an active and well supported open-source project, our challenge as a team will be to identify tests that are manageable for us to implement given our limited experience with the C programming language, communication protocols, and encryption algorithms.

1. Deliverable #2: Overall test plan and identification of our first five test cases.
2. Deliverable #3: Automated testing framework with all necessary documentation on how it works and how to use it.
3. Deliverable #4: Implementation of our automated testing framework using our 25 test cases.
4. Deliverable #5: Fault injection - 5 faults that cause some but not all of our 25 test cases to fail.
5. Deliverable #6: Final Report

Requirements traceability

In the broadest sense, the user requirements for Pidgin involve secure access to a wide range of instant messaging protocols with a high level of availability. The major tasks of Pidgin then are to securely handle user authentication, protect against intrusions, and quickly connect users to the networks and channels within networks that they wish to access. Secondary requirements include user profile management, buddy management, and GUI experience.

Update - when we wrote the above paragraph, we were not yet to a point where we could specifically address user requirements that were to be tested. We were still trying to dig into the source code to identify methods that we could get a handle on, but all of them dealt in some way with a complicated data structure or authentication/encryption method. We actually never got to the point where we were able to name specific method requirements and describe the associated test for each requirement.

Tested items

Methods that will be tested (identify by directory/filename/linenumber-methodname):

Nicholas:

- libpurple/core.c/289-purple_core_get_version(); Simple method, when called is supposed to return the version, we can run it and make sure that it is returning the correct version. *update: I used our test driver to try and run this method and i got this message... pidgin-2.11.0/libpurple/internal.h:102:21: fatal error: gmodule.h: No such file or directory. This looks like a bug in their code. I'll have to look into it.
- libpurple/version.c/ *purple_version_check(guint required_major, guint required_minor, guint required_micro)
- libpurple/version.h/ shows us the correct version numbers.
- libpurple/idle.c/291-purple_idle_get_ui_ops(void); Looks to be a simple get method. Ran tests and got the same issue with idle.c, no Gmodule.h which is supposed to be included in internal.h.
- libpurple/idle.c/291-purple_idle_get_handle(void); Another get method.

Walter:

- libpurple/buddyicon.c/248-purple_buddy_icon_data_new(guchar *icon_data, size_t icon_len, const char *filename); glib dependencies noted above are solved by including a flag that tells the compiler to look for glib library. Additional dependencies throw errors, I believe they are other pidgin files that need to be compiled.
- libpurple/imgstore.c/70-purple_imgstore_new_from_file(const char *path)

Matthew:

- libpurple/proxy.c
    - After solving glib.h and glibconfig.h dependencies, trying to include the proxy.c leads to a lot of undefined references.
- libpurple/tests/test_cipher.c
    - Including this in the test driver does not give undefined references, but has a dependency on an include interal to the test_cipher.c file.

Update - the methods above were as close as we got to naming specific requirements. We were never able to compile the associated C files to run the methods at the command line as described below.

Testing schedule

Deadline	Deliverable	Resources
Sept 27th	Test Plan & five test-cases	Pick first test cases and begin work on test driver.
Oct 18th	Automated Testing Framework	Resolve compiler dependencies, finish test driver, get used to IDE C unit testing features, organize team github repo directory
Nov 10th	Testing Implementation with twenty-five test-cases	TBD
Nov 22nd	Fault Injection	TBD
Nov 29th	Final Report	TBD

Test recording procedures

All test results will be pushed to /TestAutomation/temp with a structured name format by COB on the day the test was run. We started with reports that would be generated with a timestamp of the day on which it was generated, but we later realized that we should only store only the most recent report so scrapped the timestamp.

Hardware and software requirements

We assumed at first that we would employ the Eclipse C/C++ IDE with CDT unit testing feature to test our Pidgin code, but we actually never used Eclipse for anything other than exploring the code and trying unsuccessfully to run components like finch.

Obstacles

We have been working on the automated testing framework. We have worked out how to write a script that calls a C test driver whose main invokes an external function. We have uploaded a

sample script that runs the test driver for a separate hello world function to document our progress. We believe that these three files represent the framework by which we will test specific pidgin functions with our test driver.

We do not yet understand how to invoke a pidgin function. We run into errors when trying to compile our test driver and pidgin file that contains the function we want to test. The compiler originally threw errors related to the glib library that it was not finding on our computers, but adding a flag to the compiler invocation solved these glib dependencies. Additional dependencies relate to other pidgin files and will presumably be solved by calling the test driver in a fully compiled program rather than the raw source code where all of the dependencies of our test function have not yet been compiled.

Our task in revising this test plan deliverable after our class presentation was to get much more specific in terms of writing user requirements and associated test cases, but we weren't able to get that far with Pidgin and decided to switch projects. This section is a record of our efforts to run Pidgin functions at the command line.

We were originally quite optimistic about creating a testing framework for Pidgin, because the source files came with a "test-driver" and associated test files. We weren't able to figure out on our own how to run the test-driver, because it required a number of command line arguments that we didn't understand and for which we couldn't find documentation. We emailed the development team on September 9th requesting some direction on using the test-driver but never heard back. We concluded that the test-driver is probably used within the application, by the application, for the purposes of testing encryption protocols while the application is running.

The next step was to identify methods that we would be able to run from the command line, but we were unable to do so. Our strategy was to search in the simplest possible c files (those with the fewest #include dependencies) for methods that required simple inputs that we could replicate. Most of the methods we found usually took some type of complex data structure as an argument, usually related to the user's buddy list and other social and communication connections, but we did find some simple functions with straightforward arguments. The major problem was that we couldn't find any c files that we were actually able to compile due to complex dependencies. We tried to manually compile dependencies and we also tried to use the built-in makefile for our own purposes, but in both cases the composition of files was too complex for us to break down into smaller independent components that we could call from a command line.

Our final step was to try working just with the finch package which is the underlying communication platform without the extra pidgin bells and whistles. Finch has its own makefile, and we tried the same approach of compiling only finch and isolating finch methods, but we were stymied by the same complexity issues of the whole pidgin program.

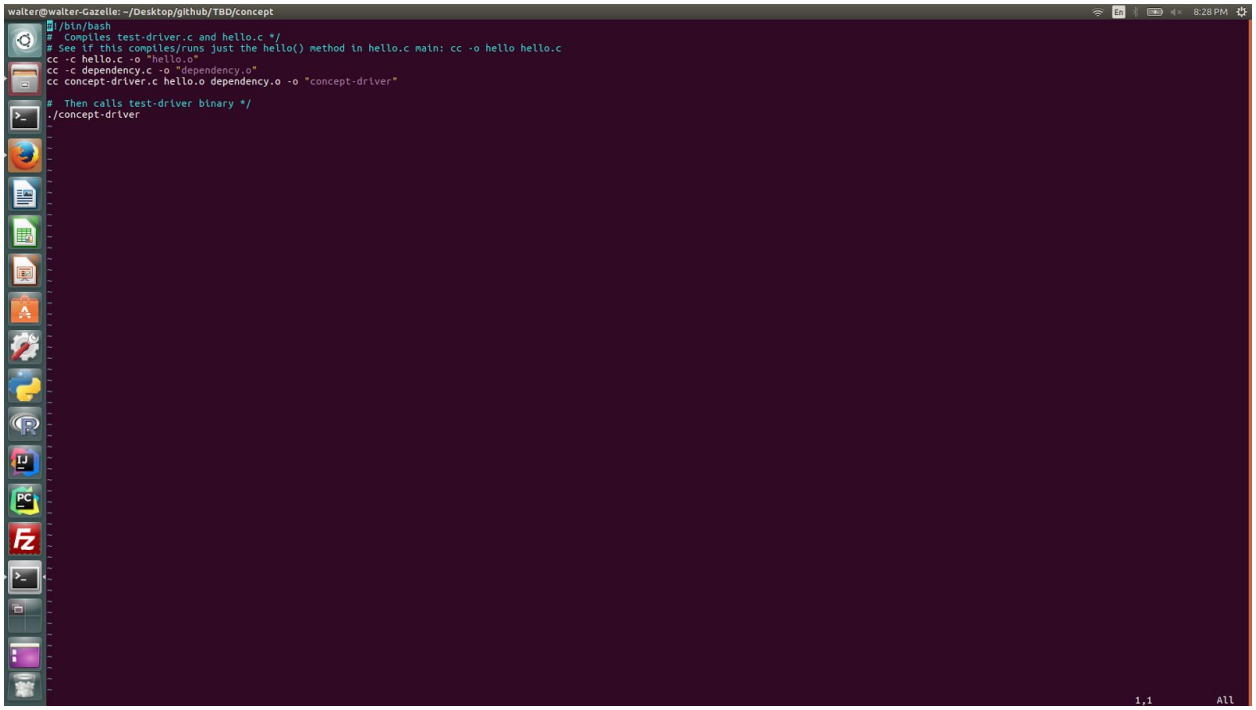


Figure 5. We wrote the above script to demonstrate the feasibility of manually compiling two C files and linking them together to create an executable. This hello, world C method worked beautifully.

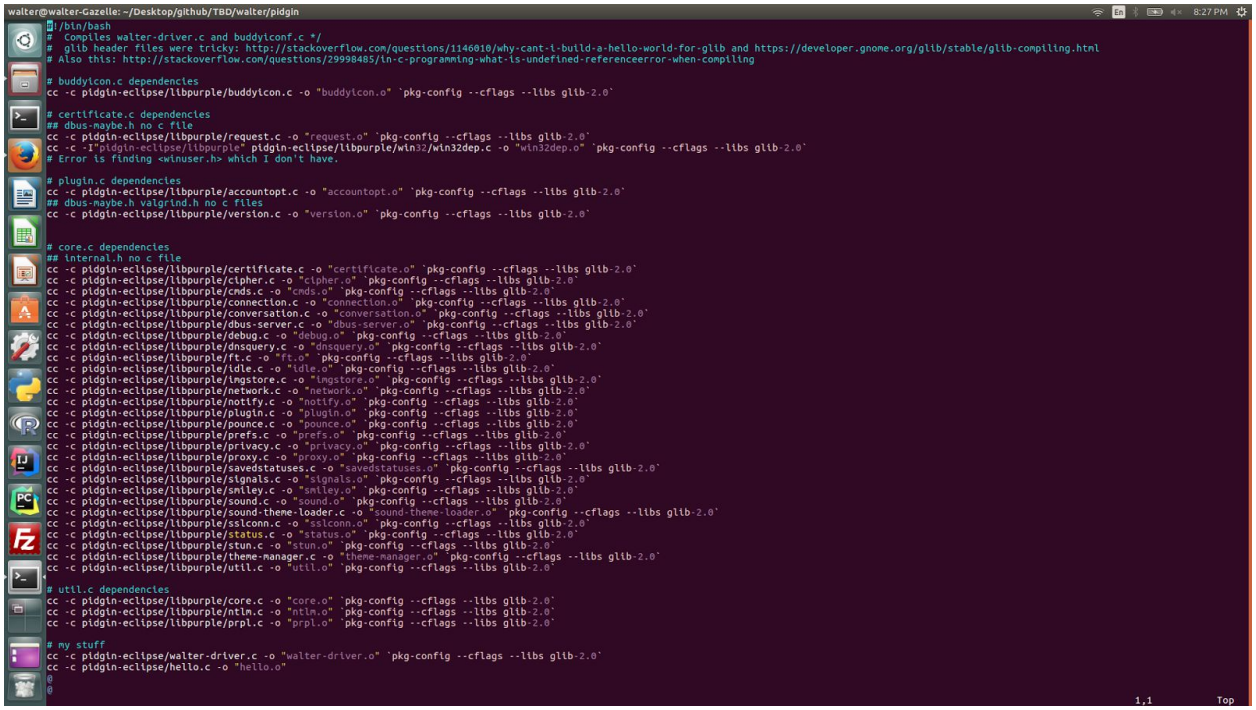


Figure 6. This is a portion of the convoluted script we were trying to build that would manually compile a Pidgin C file and its dependencies and link them together to create an executable. We abandoned this effort after it became apparent that we needed the makefile instead.

Pidgin Post-mortem

In retrospect, it's not too surprising that Pidgin proved too difficult a problem to reduce into smaller pieces. Probably any instant messaging platform is going to have integrated encryption that is always running and is going to always be connected to a dynamic contact list data structure as well as other profile information stored online.

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.

- Big projects require a more complex framework to pull it all together
- Reinforced the idea that documentation is very important (pidgin had basically none)
- Computer languages have a lot of useful built in tools that are sometimes passed by that should be more thoroughly studied

We decided to move to a new project and started afresh on a new testing plan described below. We browsed github for open-source calculator until we found one that had a good balance of functionality and accessibility.

Chapter 3 - Automated Testing Framework

The Testing Process

The ultimate goal of our team term project is to develop an automated testing framework for the ph7-Simple-Java-Calculator and implement the framework with twenty-five test cases, five injection faults, and plenty of documentation. We switched to this simple Java calculator from Pidgin, and recorded our experience with Pidgin in Chapter 2.1.

The 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. See Figures 7 & 8 below.

Because we recently switched projects, we are planning on the modified schedule below:

1. Deliverable #3:
    a. Chapter 2 Revised - Deliverable #2 regarding our experience with Pidgin.
    b. Chapter 3 - New project test plan and testing framework with 12 test cases .
2. Deliverable #4:
    a. Completed and revised automated testing framework with 25 test cases and all necessary documentation on how it works and how to use it.
3. Deliverable #5:
    a. Fault injection - 5 faults that cause some but not all of our 25 test cases to fail.
4. Deliverable #6:
    a. Final Report

Update: We actually didn't use the modified schedule above but rather caught up immediately to where we needed to be for Deliverable #3. It's nice to arrive on schedule despite a major setback!

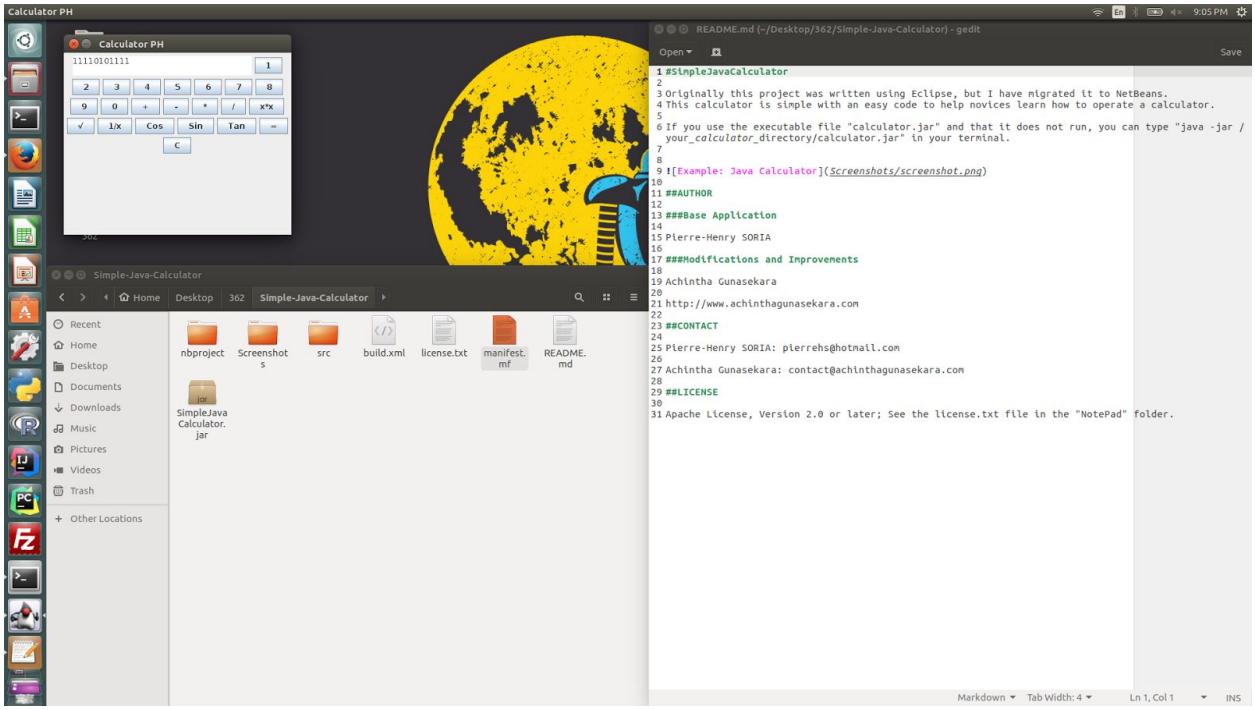


Figure 7. A sample session of the calculator's GUI and a look at the project's top-level directory and github Readme.md. What caught our attention was the thorough documentation, simple interface, and use of Java.

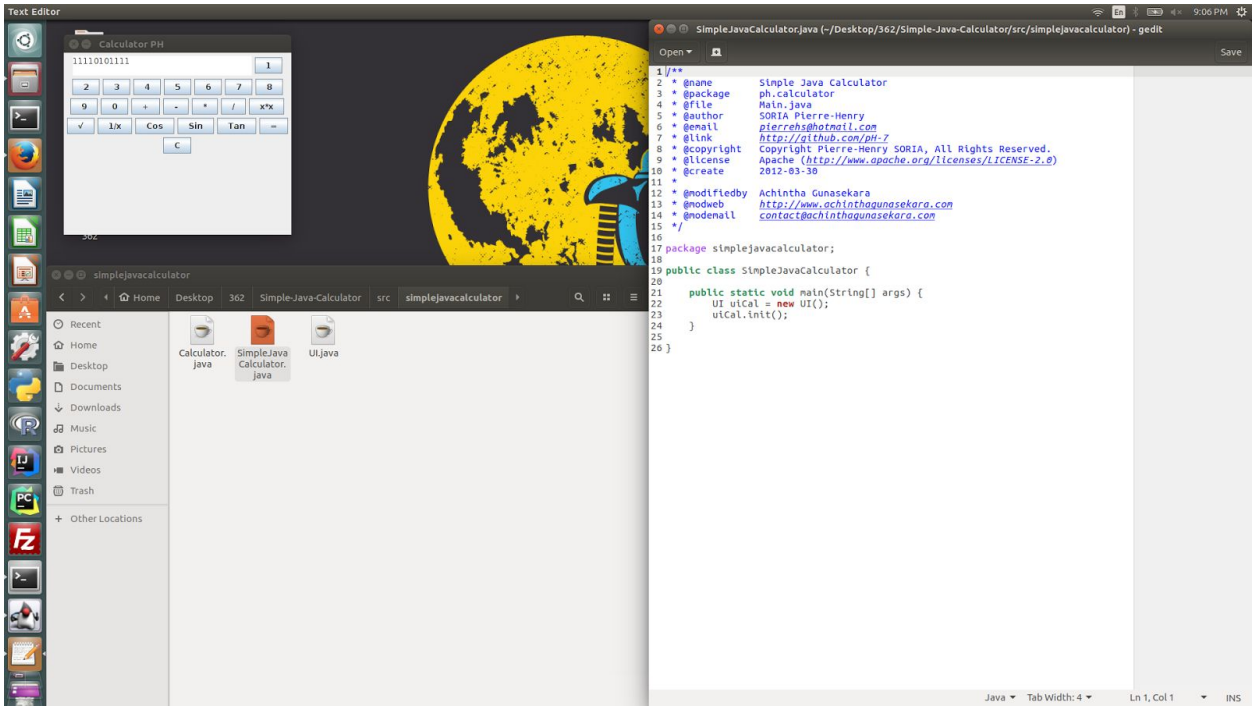


Figure 8. A look at the three source files. The simplest of the three files is shown to the right and is simply a main used to instantiate a new calculator object.

Testing Framework

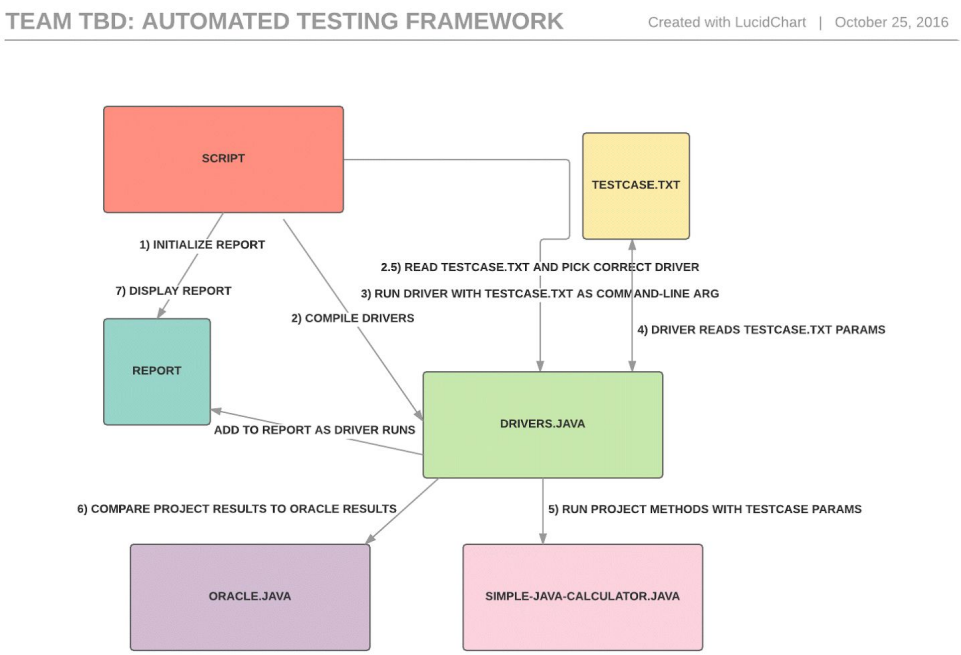


Figure 9. The original testing framework required the Java driver to do too much work and was revised after presentation feedback.

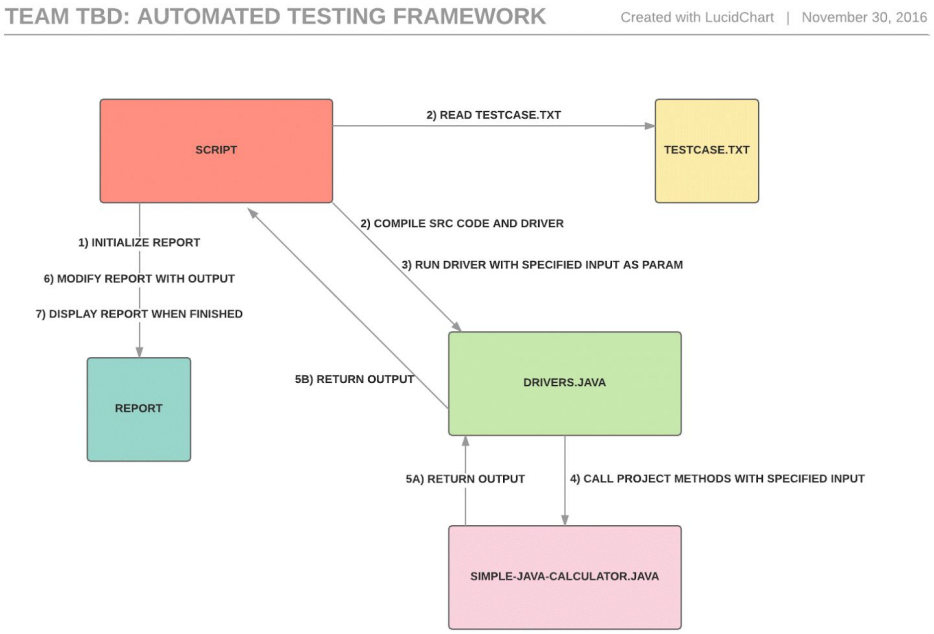


Figure 10. The revised testing framework moved all of the work to the script and also removed the separate Java oracle file since our calculator output was so simple to validate.

Original (See Figure 9 above)

- scripts/runAllTests.sh is a shell script that compiles and executes each testCase.java driver, passing the appropriate testCase.txt file as a command line argument.

- Each testCase.java driver takes a testCase.txt file name as a command line arg and reads the inputs specified in the file. The driver then executes the specified Calculator method, passing the specified inputs as parameters. Each driver can run multiple testCase.txt test cases, because the method that it chooses to run is specified in each test case file. The driver also runs an oracle.java file that calculates and returns expected results that are compared to the actual results produced by the Simple-Java-Calculator.

- Each testCase.txt file contains the details of each test case.

Revised (See Figure 10 above)

- 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.

Instructions

- To actually run the calculator GUI, clone the whole repo and run in the terminal:
    - java -jar SimpleJavaCalculator.jar .

- To run our testing framework, navigate to TestAutomation and run scripts/runAllScripts.sh at the command line.

Requirements traceability

Requirement	**Associated Testcase ID**
Return square of a positive double	1, 2, 23
Return square of a negative double	21, 22, 24
Return square root of a positive double	3, 17, 19, 20
Return square root of a negative double (nan)	18
Return square root of 0.0	4
Return the number 1 divided by a positive double	5, 6, 16
Return the number 1 divided by a negative double	25
Return cosine of a positive double	7
Return cosine of a negative double	15
Return cosine of 0.0	8
Return sine of a positive double	9
Return sine of a negative double	14
Return sine of 0.0	10
Return tangent of a positive double	11
Return tangent of a negative double	13
Return tangent of 0.0	12

Tested items

ID: 01
Requirement: Calculate squares
Component: Calculator.java
Method: calculateMono(MonoOperatorModes square, Double num)
Inputs: 2.0
Expected Outcome: 4.00

ID: 02
Requirement: Calculate squares
Component: Calculator.java
Method: calculateMono(MonoOperatorModes square, Double num)
Inputs: 9.0
Expected Outcome: 81.00

ID: 03
Requirement: Calculate square roots
Component: Calculator.java
Method: calculateMono(MonoOperatorModes squareRoot, Double num)
Inputs: 9.0
Expected Outcome: 3.00

ID: 04
Requirement: Calculate square roots
Component: Calculator.java
Method: calculateMono(MonoOperatorModes squareRoot, Double num)
Inputs: 0.0
Expected Outcome: 0.00

ID: 05
Requirement: Calculate division
Component: Calculator.java
Method: calculateMono(MonoOperatorModes onedividedby, Double num)
Inputs: 1.0
Expected Outcome: 1.00

ID: 06
Requirement: Calculate division
Component: Calculator.java
Method: calculateMono(MonoOperatorModes onedividedby, Double num)
Inputs: 10.0
Expected Outcome: 0.10

ID: 07
Requirement: Calculate trig functions(cos)

Component: Calculator.java
Method: calculateMono(MonoOperatorModes cos, Double num)
Inputs: 1.0
Expected Outcome: 0.54

ID: 08
Requirement: Calculate trig functions(cos)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes cos, Double num)
Inputs: 0.0
Expected Outcome: 1.00

ID: 09
Requirement: Calculate trig functions(sin)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes sin, Double num)
Inputs: 1.0
Expected Outcome: 0.84

ID: 10
Requirement: Calculate trig functions(sin)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes sin, Double num)
Inputs: 0.0
Expected Outcome: 0.00

ID: 11
Requirement: Calculate trig functions(tan)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes tan, Double num)
Inputs: 1.0
Expected Outcome: 1.56

ID: 12
Requirement: Calculate trig functions(tan)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes tan, Double num)
Inputs: 0.0
Expected Outcome: 0.00

ID: 13
Requirement: Calculate trig functions(tan)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes tan, Double num)
Inputs: -1.0
Expected Outcome: -1.56

ID: 14

Requirement: Calculate trig functions(sin)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes sin, Double num)
Inputs: -1.0
Expected Outcome: -0.84

ID: 15
Requirement: Calculate trig functions(cos)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes cos, Double num)
Inputs: -1.0
Expected Outcome: 0.54

ID: 16
Requirement: Calculate division
Component: Calculator.java
Method: calculateMono(MonoOperatorModes onedividedby, Double num)
Inputs: 42.0
Expected Outcome: 0.02

ID: 17
Requirement: Calculate square roots
Component: Calculator.java
Method: calculateMono(MonoOperatorModes squareRoot, Double num)
Inputs: 5.0
Expected Outcome: 2.24

ID: 18
Requirement: Calculate square roots
Component: Calculator.java
Method: calculateMono(MonoOperatorModes squareRoot, Double num)
Inputs: -5.0
Expected Outcome: NaN

ID: 19
Requirement: Calculate square roots
Component: Calculator.java
Method: calculateMono(MonoOperatorModes squareRoot, Double num)
Inputs: 100.0
Expected Outcome: 10.00

ID: 20
Requirement: Calculate square roots
Component: Calculator.java
Method: calculateMono(MonoOperatorModes squareRoot, Double num)
Inputs: 10.5
Expected Outcome: 3.24

ID: 21
Requirement: Calculate squares
Component: Calculator.java
Method: calculateMono(MonoOperatorModes square, Double num)
Inputs: -2.0
Expected Outcome: 4.00

ID: 22
Requirement: Calculate squares
Component: Calculator.java
Method: calculateMono(MonoOperatorModes square, Double num)
Inputs: -9.0
Expected Outcome: 81.00

ID: 23
Requirement: Calculate squares
Component: Calculator.java
Method: calculateMono(MonoOperatorModes square, Double num)
Inputs: 73.5
Expected Outcome: 5402.25

ID: 24
Requirement: Calculate squares
Component: Calculator.java
Method: calculateMono(MonoOperatorModes square, Double num)
Inputs: -73.5
Expected Outcome: 5402.25

ID: 25
Requirement: Calculate division
Component: Calculator.java
Method: calculateMono(MonoOperatorModes onedividedby, Double num)
Inputs: -15.5
Expected Outcome: -0.06

Chapter 4 - Refactoring Testing Script

Summary

Team TBD has revamped our testing script in order to work per specifications. We have also cleaned it up to produce easier to read results. The script now calls the test driver specified in the test cases. We have also made more test cases bringing our total up to the required 25. Our test cases are more thorough and fully test each method. We developed our drivers as well.

Testing Framework

/scripts/ holds our runAllTests bash file.
/testCases/ holds the collection of text cases in format explained below.
/project/* holds the **bin/** and **src/** folders containing the calculator files themselves.
/testCaseExecutables/ holds the drivers used for each type of test.
/reports is where the report of the tests run is placed, and contains an external stylesheet for its formatting.

Running Instructions

4. Clone the project from the Team TBD repository.
5. Navigate to the *TestAutomation* folder.
6. Run *./scripts/runAllTests.sh*
7. The tests will run and print out each result in the terminal.
8. After completion of all the tests, an HTML report of the test results will be created and opened.

Note on test cases:

We have written our test cases in the following format:

<Parameter> + [TAB] + <Value>

An example test case file would appear as the following:

ID	1
Req	Calculating the square of the input
Comp	Calculator.java
Method	public Double calculateMono(MonoOperatorModes square, Double num)
Inputs	2.0
Expect	4.0
Driver	testDriverSquare

Results

Test #	Method	Input	Expected	Result
Test #1	public Double calculateMono(MonoOperatorModes square, Double num)	2.0	4.0	4.0
Test #2	public Double calculateMono(MonoOperatorModes square, Double num)	9.0	81.0	81.0
Test #3	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	9.0	3.0	3.0
Test #4	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	0.0	0.0	0.0
Test #5	public Double calculateMono(MonoOperatorModes oneDividedBy, Double num)	1.0	1.0	1.0
Test #6	public Double calculateMono(MonoOperatorModes oneDividedBy, Double num)	10.0	0.1	0.1
Test #7	public Double calculateMono(MonoOperatorModes cos, Double num)	1.0	0.5403023058681398	0.5403023058681398
Test #8	public Double calculateMono(MonoOperatorModes cos, Double num)	0.0	1.0	1.0
Test #9	public Double calculateMono(MonoOperatorModes sin, Double num)	1.0	0.8414709848078965	0.8414709848078965
Test #10	public Double calculateMono(MonoOperatorModes sin, Double num)	0.0	0.0	0.0
Test #11	public Double calculateMono(MonoOperatorModes tan, Double num)	1.0	1.5574077246549023	1.5574077246549023
Test #12	public Double calculateMono(MonoOperatorModes tan, Double num)	0.0	0.0	0.0
Test #13	public Double calculateMono(MonoOperatorModes tan, Double num)	-1.0	-1.5574077246549023	-1.5574077246549023
Test #14	public Double calculateMono(MonoOperatorModes sin, Double num)	-1.0	-0.8414709848078965	-0.8414709848078965
Test #15	public Double calculateMono(MonoOperatorModes cos, Double num)	-1.0	0.5403023058681398	0.5403023058681398
Test #16	public Double calculateMono(MonoOperatorModes oneDividedBy, Double num)	42.0	0.023809523809523808	0.023809523809523808
Test #17	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	5.0	2.23606797749979	2.23606797749979
Test #18	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	-5.0	NaN	NaN
Test #19	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	100.0	10.0	10.0
Test #20	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	10.5	3.24037034920393	3.24037034920393
Test #21	public Double calculateMono(MonoOperatorModes square, Double num)	-2.0	4.0	4.0
Test #22	public Double calculateMono(MonoOperatorModes square, Double num)	-9.0	81.0	81.0
Test #23	public Double calculateMono(MonoOperatorModes square, Double num)	73.5	5402.25	5402.25
Test #24	public Double calculateMono(MonoOperatorModes square, Double num)	-73.5	5402.25	5402.25
Test #25	public Double calculateMono(MonoOperatorModes oneDividedBy, Double num)	-15.5	-0.06451612903225806	-0.06451612903225806

Figure 11. Our first report in which the format was not entirely correct. We have since updated it to explicitly state **pass** or **fail**.

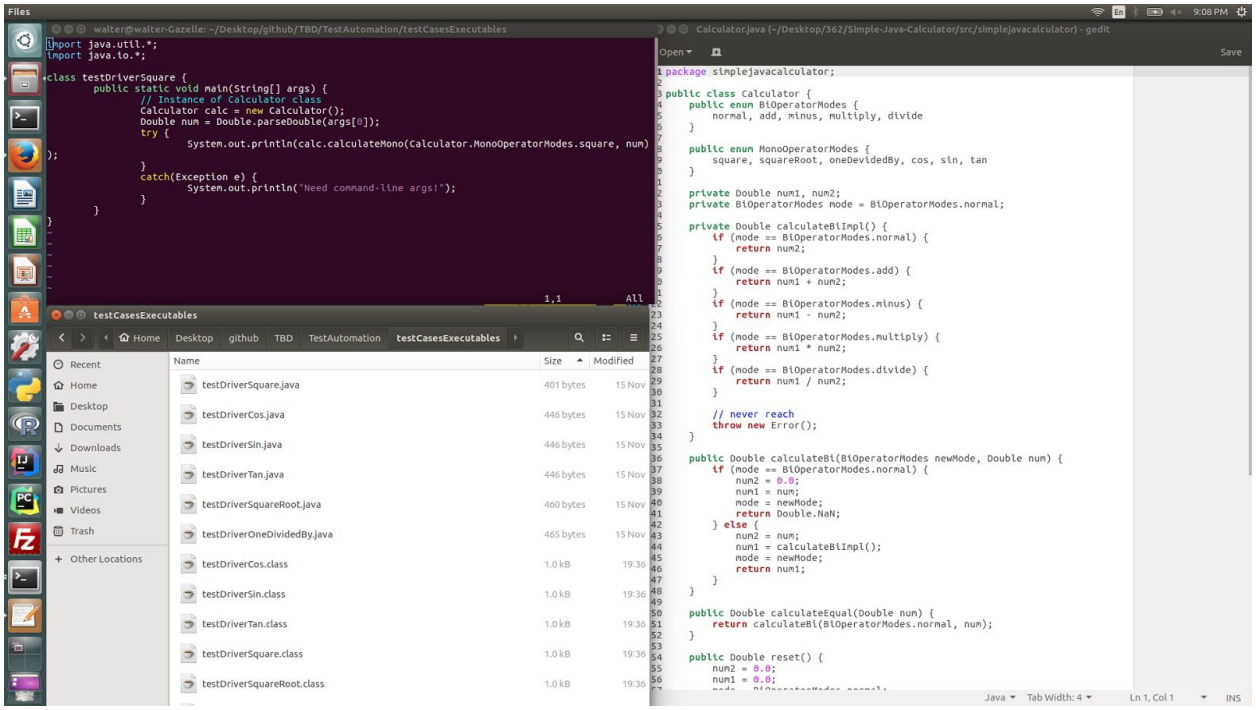
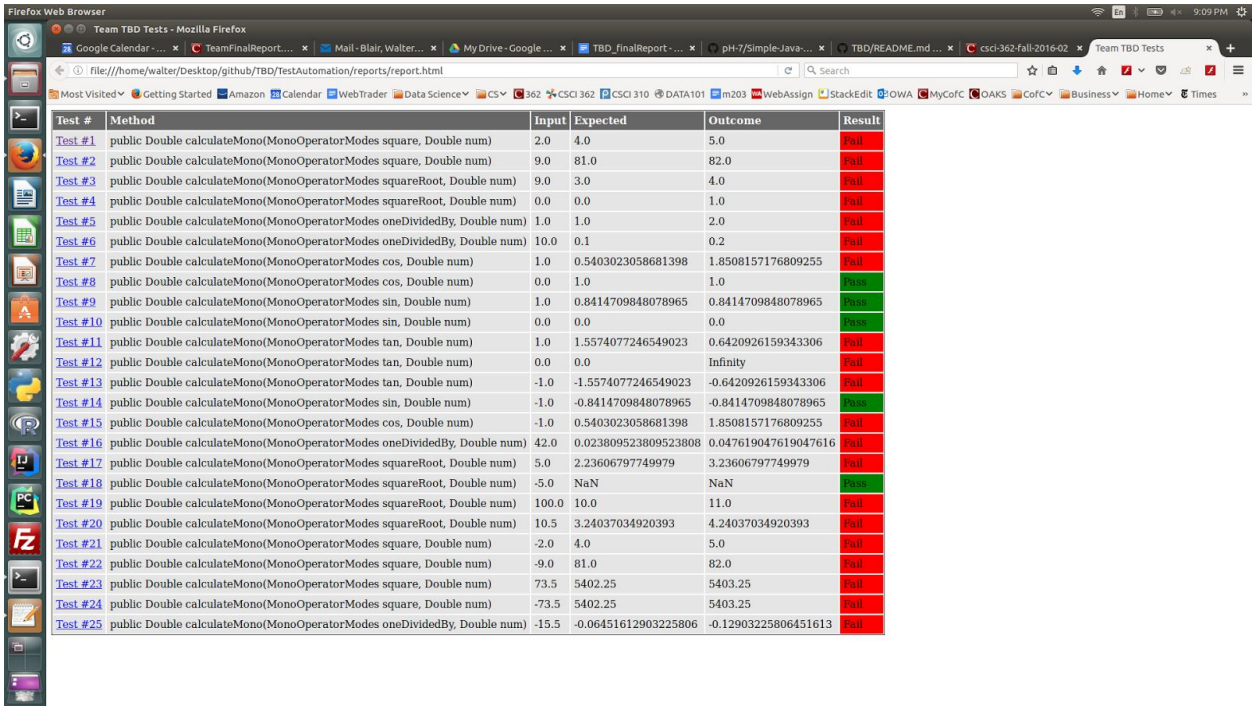


Figure 12. A look at an example testdriver.java file, the directory containing all of our test drivers, and the main project sourcefile Calculator.java that contains all of the methods we test.



Test #	Method	Input	Expected	Outcome	Result
Test #1	public Double calculateMono(MonoOperatorModes square, Double num)	2.0	4.0	5.0	Fail
Test #2	public Double calculateMono(MonoOperatorModes square, Double num)	9.0	81.0	82.0	Fail
Test #3	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	9.0	3.0	4.0	Fail
Test #4	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	0.0	0.0	1.0	Fail
Test #5	public Double calculateMono(MonoOperatorModes oneDividedBy, Double num)	1.0	1.0	2.0	Fail
Test #6	public Double calculateMono(MonoOperatorModes oneDividedBy, Double num)	10.0	0.1	0.2	Fail
Test #7	public Double calculateMono(MonoOperatorModes cos, Double num)	1.0	0.5403023058681398	1.8508157176809255	Fail
Test #8	public Double calculateMono(MonoOperatorModes cos, Double num)	0.0	1.0	1.0	Pass
Test #9	public Double calculateMono(MonoOperatorModes sin, Double num)	1.0	0.8414709848078965	0.8414709848078965	Pass
Test #10	public Double calculateMono(MonoOperatorModes sin, Double num)	0.0	0.0	0.0	Pass
Test #11	public Double calculateMono(MonoOperatorModes tan, Double num)	1.0	1.5574077246549023	0.6420926159343306	Fail
Test #12	public Double calculateMono(MonoOperatorModes tan, Double num)	0.0	0.0	Infinity	Fail
Test #13	public Double calculateMono(MonoOperatorModes tan, Double num)	-1.0	-1.5574077246549023	-0.6420926159343306	Fail
Test #14	public Double calculateMono(MonoOperatorModes sin, Double num)	-1.0	-0.8414709848078965	-0.8414709848078965	Pass
Test #15	public Double calculateMono(MonoOperatorModes cos, Double num)	-1.0	0.5403023058681398	1.8508157176809255	Fail
Test #16	public Double calculateMono(MonoOperatorModes oneDividedBy, Double num)	42.0	0.023809523809523808	0.047619047619047616	Fail
Test #17	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	5.0	2.23606797749979	3.23606797749979	Fail
Test #18	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	-5.0	NaN	NaN	Pass
Test #19	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	100.0	10.0	11.0	Fail
Test #20	public Double calculateMono(MonoOperatorModes squareRoot, Double num)	10.5	3.24037034920393	4.24037034920393	Fail
Test #21	public Double calculateMono(MonoOperatorModes square, Double num)	-2.0	4.0	5.0	Fail
Test #22	public Double calculateMono(MonoOperatorModes square, Double num)	-9.0	81.0	82.0	Fail
Test #23	public Double calculateMono(MonoOperatorModes square, Double num)	73.5	5402.25	5403.25	Fail
Test #24	public Double calculateMono(MonoOperatorModes square, Double num)	-73.5	5402.25	5403.25	Fail
Test #25	public Double calculateMono(MonoOperatorModes oneDividedBy, Double num)	-15.5	-0.06451612903225806	-0.12903225806451613	Fail

Figure 13. Our current report that states Pass/Fail. Faults are injected here to demonstrate Fails.

Chapter 5 - Fault Injection

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.

To fix each fault injection, just uncomment the return statement that is commented and comment out the return statement below it.

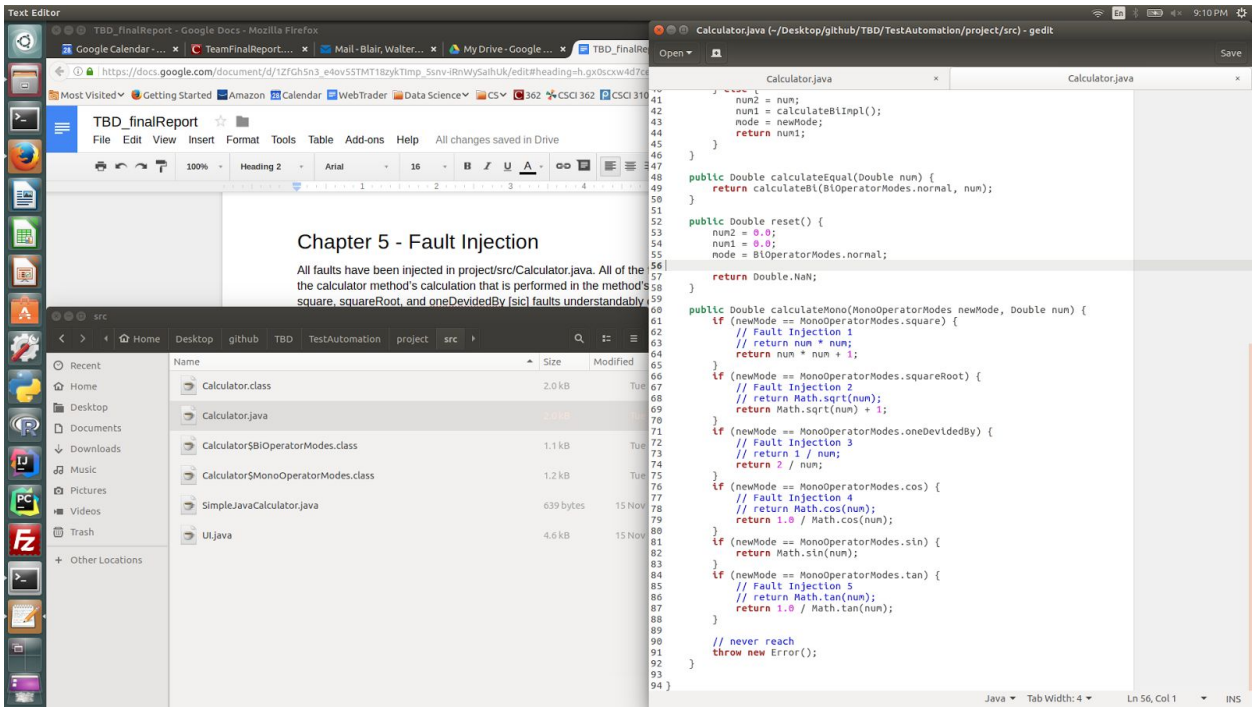


Figure 14. A look at the faults injected into Calculator.java

Faults:

Fault 1: method: calculateMono(square, num), line 63-64

Change: replaced <return num * num;> with <return num * num + 1;>

Adds 1 to the square

Test(s) that should fail:

- 1
- 2
- 21
- 22
- 23
- 24

Fault 2: method: calculateMono(squareRoot, num), line 68-69

Change: replaced <return Math.sqrt(num);> with <return Math.sqrt(num) + 1;>

Adds 1 to the square root

Test(s) that should fail

- 3
- 4
- 17
- 19
- 20

Fault 3: method: calculateMono(oneDevidedBy, num), line 73-74

Change: replaced <return 1 / num;> with <return 2 / num;>

Replaces numerator 1 with 2

Test(s) that should fail:

- 5
- 6
- 16
- 25

Fault 4: method: calculateMono(cos, num), line 78-79

Change: replaced <return Math.cos(num);> with <return 1.0 / Math.cos(num);> (replaced cosine with secant formula)

Test(s) that should fail:

- 7
- 15

Fault 5: method: calculateMono(tan, num), line 86-87

Change: replaced <return Math.tan(num);> with <return 1.0 / Math.tan(num);>

(replaced tangent with cotangent formula)

Test(s) that should fail:

- 11
- 12
- 13

Chapter 6 - Final Thoughts

Overall Experiences

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. We enjoyed learning about how testing should work in a real world situation.

What We Learned

We have learned a great deal overall from this project so I am going to list the things we have learned in bullet point form.

- Documentation is super important.
- Team communication can be very difficult and schedules are hard to match up, but communication is key.
- 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.
- Organization of files and data helps creating a project run faster and smoother.
- Working remotely on a project does not work without good and intensive documentation.

Self-Evaluation

Our overall team evaluation was that we overcame some big, stressful challenges midway through the semester by finding a way around the Pidgin problem and building a successful automated testing framework despite our time crunch in the last few weeks. If we had to do it over again, we would be much more cautious in selecting an open source project. We probably would have introduced ourselves to the development teams of our candidate projects at the very

beginning and only selected a project with a responsive development team. It would have saved us a month or so of frustration to understand what the Pidgin ‘test suite’ was about and how to compile individual components, if possible. We struggled to find our roles in the team early on but settled in somewhat to be quite productive toward the end of the semester.

We have some individual notes, beyond our overall team perspective, below.

Nicholas Victor Johnson: I found that I had a very big weakness in the area of Linux and open source software. I fell behind early on because I thought it was going to be easy going and ran into a rude awakening in trying to install Linux and getting all of the proper dependencies to work, in fact my Linux install still fails to connect to the CofC private wifi server for whatever reason. I could have done much better time management for this project, which is honestly unlike me for the most part. I usually start things earlier than other people due to the fact that I know I will run into problems, but this time I failed to exert my usual wisdom.

Walter Blair: I was definitely overconfident in the beginning while we were selecting projects to work on and then later on in the semester when we were trying to find a good project to work on. I underestimated the difficulty of orienting to and understanding project source code and the many obstacles that arise when trying to access individual components of the system. Next time I would slow down and think more carefully about realistic goals. I also could have done much better in communicating with my team, but I always struggle with group work.

Assignment Evaluation

This assignment was a great experience and taught us all a lot about what it would be like to work on a real team on a real automated testing framework. We now know how hard and time consuming it can be to make a good product, not only is it a lot of organization and work to get all of the structure and coding itself done, but there is a whole side of testing that requires enormous efforts. Planning is super important to pull off a large project like the ones we talked about and worked on in class.

It would be tempting after the numerous frustrations in this assignment to wish to work on more polished software projects, like a software product produced by a large private company, but there is something really refreshing and exciting about seeing a work in progress and getting our hands dirty in the project of improving something that everyone has access to. There’s the feeling that open source work is sometimes less of a corporate steam engine rolling forward with intense focus on developing a single product line and more of a sailboat floating along with the prevailing winds (or occasionally against them). It seems like open source projects probably ebb and flow depending on who is interested in putting time and creative energy into the project at any given time and what the other alternatives are to customers interested in the software. I think the open source perspective offered by this assignment was well worth the hurdles.