Team TBD

CSCI 362, Fall 2016

Chapter 5 Draft

Matthew Claudon

Nicholas Johnson

Walter Blair

Table of Contents

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/

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.

- 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.

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. 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.

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 appears to be 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 - Pidgin

Pidgin Post-mortem

The Deliverable #2 work below was written for our September 27th Testing Plan. Our task in revising this test plan 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 have 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.

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.

Deliverable #2 (Original)

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.

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.

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

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.

Test recording procedures

All test results will be pushed to /TestAutomation/temp with a structured name format (TBD) by COB on the day the test was run.

Hardware and software requirements

Test suites will be small and therefore will not require a great deal of hardware resources. The Eclipse C/C++ IDE with CDT unit testing feature will be used for software test development.

Chapter 3 - Automated Testing Framework

Foreword - our experience with pidgin

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

The Testing Process

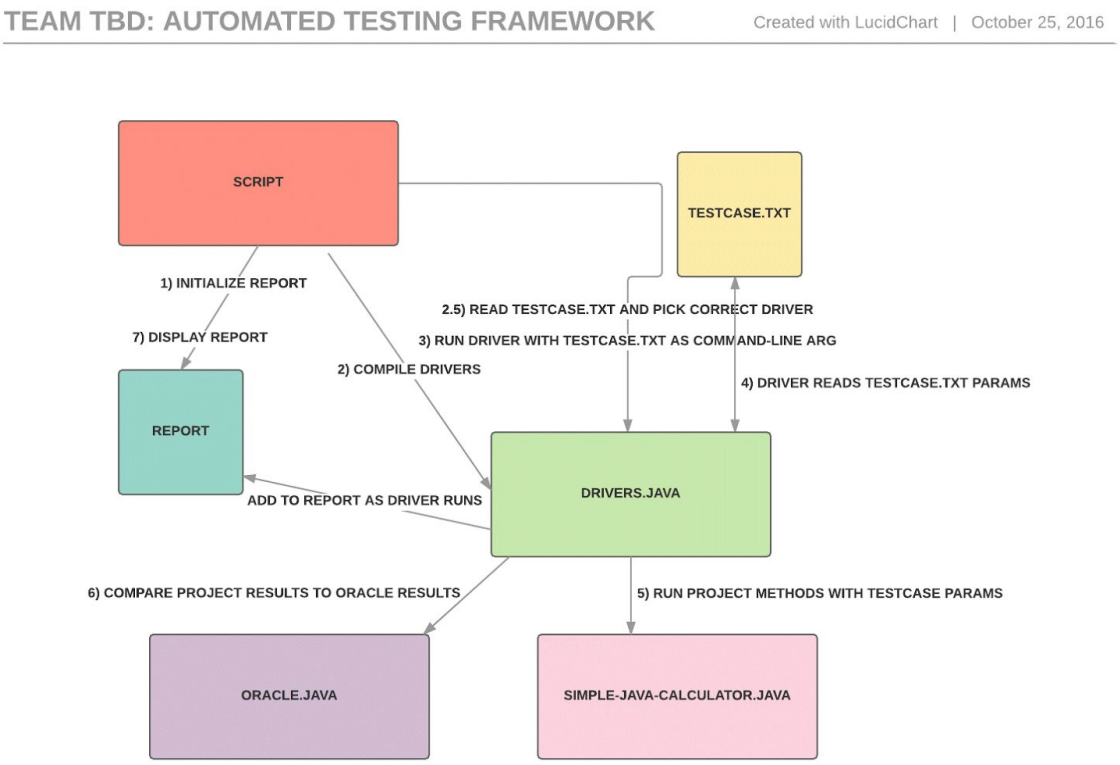
The ultimate goal of our team term project is to develop an automated testing framework for the ph7-Simple-Java-Calculator and implement the framework with twenty-five test cases, five injection faults, and plenty of documentation. We switched to this simple Java calculator from Pidgin, and recorded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

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

Testing Framework



- 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.

- 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.

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

Requirements traceability

List requirements here, see format

- *Take in numerical inputs*
- *Calculate squares*
- *Calculate square roots*
- *Calculate division*
- *Calculate addition*
- *Calculate subtraction*
- *Calculate multiplication*
- *Calculate trig functions*
- *Be able to clear the calculator inputs*
- *Display the human readable results*

Tested items

List of testcases

ID: 01
Requirement: Calculate squares
Component: Calculator.java
Method: calculateMono(MonoOperatorModes square, Double num)
Inputs: 2.0
Expected Outcome: 4.0

ID: 02
Requirement: Calculate squares
Component: Calculator.java
Method: calculateMono(MonoOperatorModes square, Double num)
Inputs: 9.0
Expected Outcome: 18.0

ID: 03
Requirement: Calculate square roots
Component: Calculator.java
Method: calculateMono(MonoOperatorModes squareRoot, Double num)
Inputs: 9.0
Expected Outcome: 3.0

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

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

ID: 06
Requirement: Calculate division
Component: Calculator.java
Method: calculateMono(MonoOperatorModes onedividedby, Double num)
Inputs: 10.0
Expected Outcome: 0.1

ID: 07
Requirement: Calculate trig functions(cos)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes cos, Double num)
Inputs: 1.0
Expected Outcome: 0.540302305868

ID: 08
Requirement: Calculate trig functions(cos)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes cos, Double num)
Inputs: 0.0
Expected Outcome: 1.0

ID: 09
Requirement: Calculate trig functions(sin)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes sin, Double num)
Inputs: 1.0
Expected Outcome: 0.841470984808

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

ID: 11
Requirement: Calculate trig functions(tan)
Component: Calculator.java
Method: calculateMono(MonoOperatorModes tan, Double num)
Inputs: 1.0
Expected Outcome: 1.557407724655

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

Testing schedule

Deadline	Deliverable	Resources
October 25th	Testing framework with 12 test-cases	ph7-Simple-Java-Calculator
November 17th	Revised framework with 25 test-cases	TBD
November 29th	Fault Injection	TBD
December 6th	Final Report	TBD

Test recording procedures

Tests are recorded as date stamped html files called record$DATE.html and include the $USER who ran the test within the file. Tests are stored in the records/ directory.

Chapter 4

Need this

Chapter 5 - Fault Injection

All faults have been 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.

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