DATE: April 10, 2020

TO: Dr. Emanuel Tutuc, Oliver Davidson

FROM: Canyon Evenson, Michael Flanders, Jennings Inge, Saarila Kenkare, John Koelling, Janvi Pandya

SUBJECT: Triaging, Diagnosing, and Predicting Software Errors Test and Evaluation Report

INTRODUCTION

Our project aims to create a regression test selection tool for the R programming language. This tool will recommend specific tests for a developer to run when they make a change to a software application, so that they can test their changes without running the entire test suite of the application. The success of the project will be measured by the accuracy of the tool's test recommendations, and the speed at which it performs its tasks.

SUMMARY OF DESIGN

This project consists of three distinct modules, each relating to a mapping of source code to tests, and a testing harness combining the functionality of these modules. The developer will interact directly only with the fourth module, the testing harness. The central functionality of the project is implemented in the first three modules.

Module 1: Test-to-Source Mapping

In order to reduce time costs, this module maps specific tests to the areas of the source code that the test exercises, as detailed in Fig. 1. The input to the module is a specific test case, or test suite, and the output of the module is a list of which functions from the source code were covered by the tests. Using a parsing method implemented in Python, the tool uses a MySQL database to keep track of what tests have been run. By interfacing with module 2, this module will be able to find out if the code or tests have been changed, and if so, it will regenerate mappings. Otherwise, it will use the already stored mappings in the database to output the exercised functions.

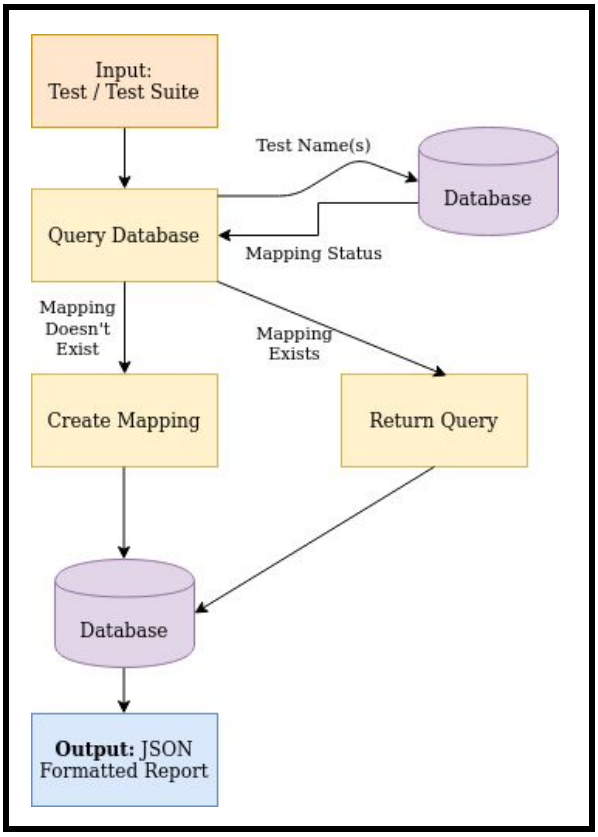

Figure 1. Test to Source Mapping

An implementation of this module has now been completed. Although Python is our implementation language, R is the source code language, so we rely on both R scripts and Python processing for implementation. We tried out two different methods of accomplishing this goal, and selected the one that was best for accuracy, as that is our most important metric.

The implementation method we decided to use relies on an R code coverage library called covr. As we decided to use function-level granularity, we ask covr to calculate the coverage percentage between each test case and function. If it's greater than 0, we map the test case to the function and store the mapping in the database. If it was decided later by the users of our tool that they want a higher coverage (for example, if they only wanted tests which test at least half of the given function), we could easily change our script to have a higher threshold for linkage.

The user can input a specific test name and all of the functions that test calls will be returned. There is also a command line parameter which allows the user to say if the program should go through the mappings process or not. If so, it redoes all mappings and stores any new ones in the database. If not, it queries the database for the given test.

Module 2: Differences in Code

As shown in Fig. 2, this module has two execution modes. The first execution path will take a revision of code and select associated tests. The second execution path will inform the harness of files that have been changed. If any changes have been made, the module will request that a new mapping be generated.

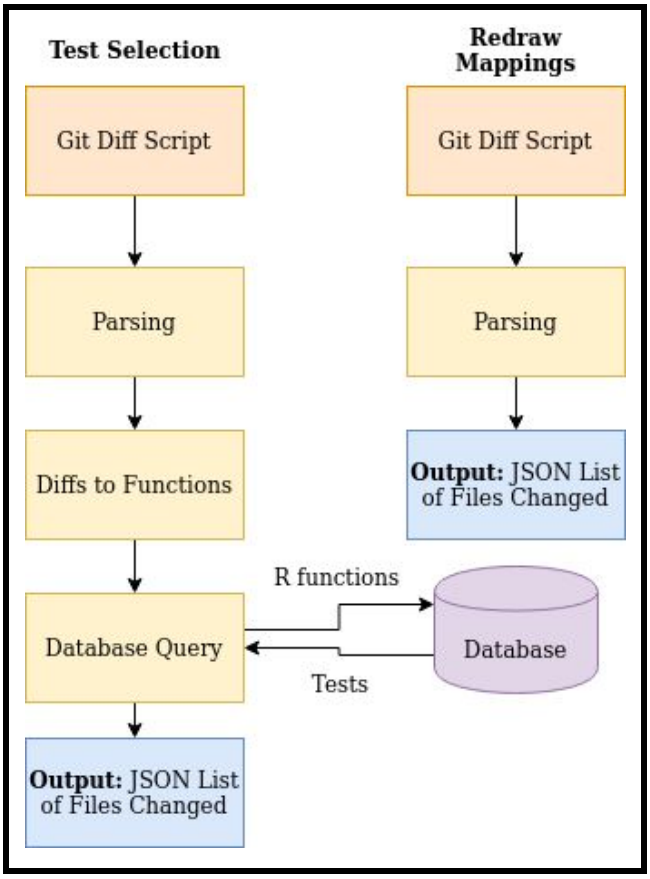

Figure 2. Code Diff to Tests

The first execution mode, test selection, will recommend the tests based on areas of changed code, so as to exercise only certain fragments of the program. Git will be the source code's version control system, and this module will integrate with Git's built in revision information tools. This module will reference the mappings in persistent storage created by other modules to produce a list of tests that exercise changed areas of code. The test selection execution will be conducted in three steps. Firstly, the module uses Git's built-in revision information tools to review the changes made in source code using the "git diff" command, and compiles a list of code changes. Secondly, the module parses through the output to yield a list of the specific functions that have changed. Third, the module queries the database for their source-to-test mappings, and outputs a list of associated tests to run. This module is implemented in a single python wrapper script so that it can be called by the harness.

The second execution mode, redrawing mappings, is invoked by the testing harness to update the mappings stored in the database. This will occur after any code change so that the database (and therefore test selection) will be up to date. The process begins by querying Git for code changes. The module will parse the output and return a list of files changed to the testing harness. The testing harness will then use that to conduct a targeted update of the mapping database, using the test-to-source module.

Module 3: Source-to-Test Mapping

Fig. 3 shows the setup of the source-to-test module. The input is the name of an R source code file, and the output is a list of the tests that exercise the source code file. The module will gather information about the source to test mappings from the MySQL database. These mappings will already have been generated by the Test-to-Source module. The output is a JSON report that shows all functions and test cases that map to the source code file.

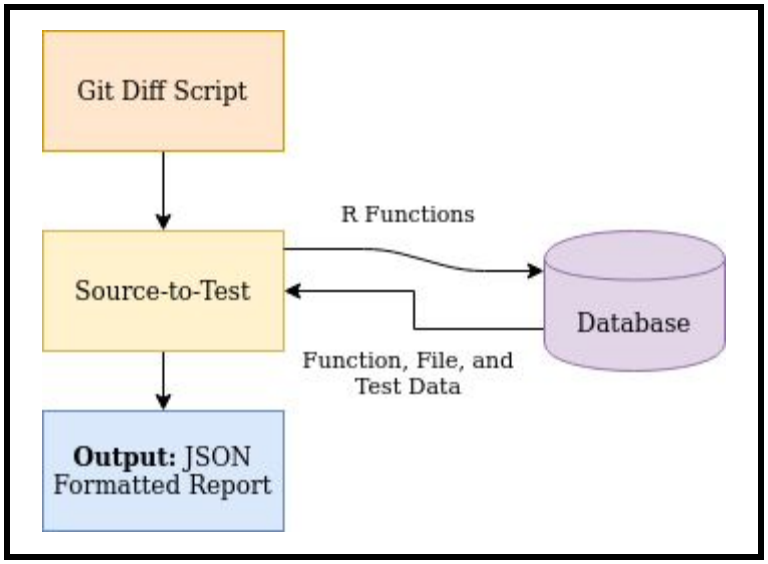

Figure 3. Source to Test Mapping

The implementation of this module has been completed and tested. For usage internal to the tool—i.e. usage by other modules—we have created a python class that can retrieve the source-to-test mappings by simply calling "SourceToTest.getJson(fileName)".

Module 4: Testing Harness

The testing harness, as detailed in Fig. 4, combines the three previous modules to create a simplified command line interface for Dell's engineers to interact with. The harness should be run after a user makes changes in code, then commits them. At this point, this module will: 1) invoke module 2 to compare the current revision to the previous one, 2) invoke module 3 to find which test cases map to the changed code, and then 3) run the selected subset of tests.

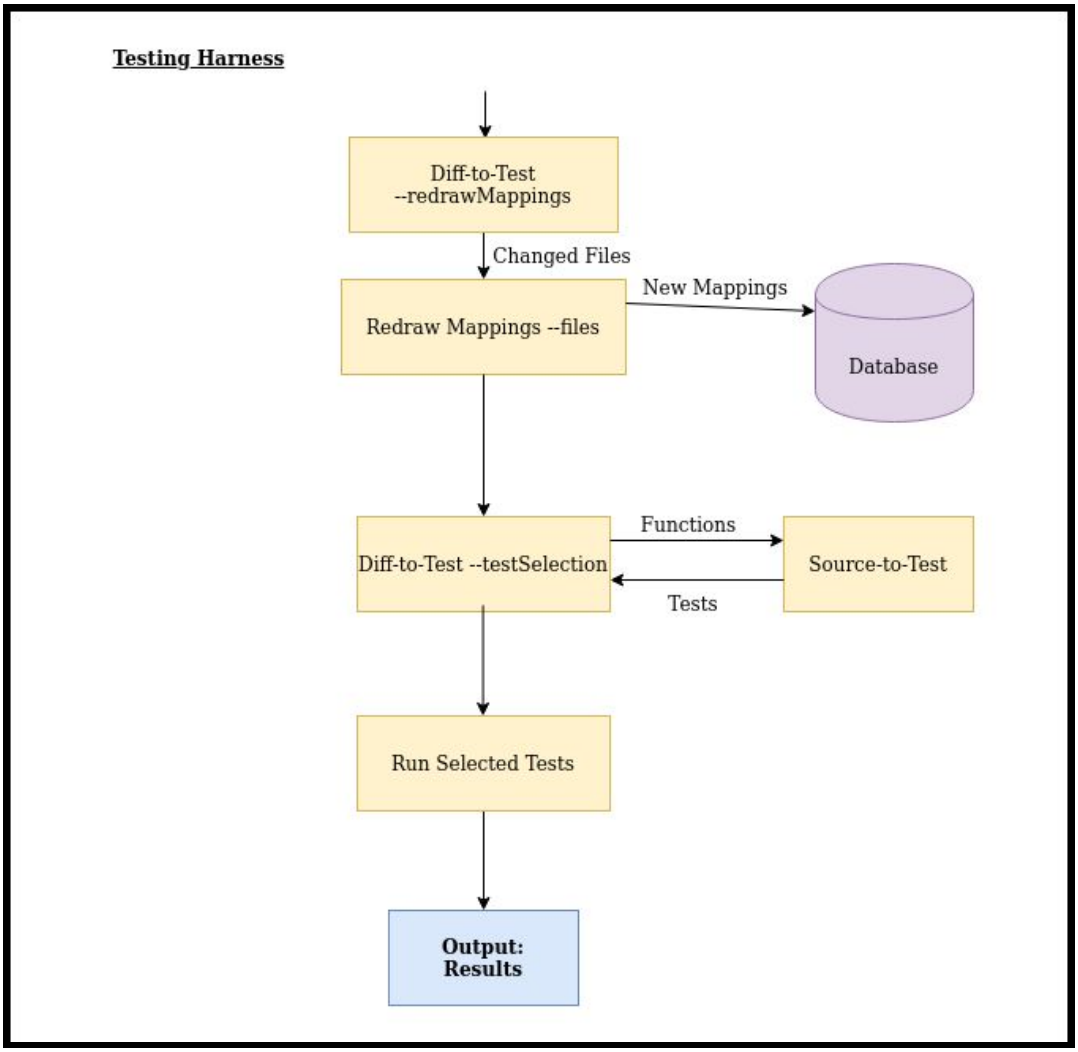

Figure 4. Testing Harness

When Module 2 is run, it will check if the test-to-source mappings have already been prepared. If not, then this means that either: 1) none of the mappings have been created or 2) the source code file is new--i.e. it was introduced in the latest code revision. In the first case, the mappings will need to be created before using this module--by running Module 1--and the module will warn the user about this. In the second case, we will need to recreate mappings for any new or modified test files since mappings will not already be stored for any newly created or changed files. If the

source still does not have a mapping stored in the database, then we will again warn the user. Finally, we will output the source-to-test mappings to users as a JSON report as shown in Fig. 4.

PROJECT TESTING

The overall goal of the project is to select tests accurately in a short amount of time. Although correctness of tests has been determined as a more important metric than time, time is a quantitative metric to use for optimization of our project's performance. The overall execution time is as follows.

$$I = Harness\ initialization\ and\ test\ selection\ time$$

$$T = Time\ to\ run\ all\ selected\ tests$$

$$M = Time\ to\ create\ and\ update\ stored\ mappings$$

$$(Eq.\ 1)\ Total\ run\ time = I + T + M$$

While time T is dependent on the tests themselves, it can be reduced by selecting the right tests to run. Times I and M are dependent on the modules themselves and can be optimized. The total run time is ultimately most significant.

Testing Method

Since the modules and testing harness are designed to be scalable to many R code repositories, we have decided to test functionality of all aspects of the project with a simulated repository of simplistic code to exercise module performance independent of parsing. In addition to unit testing, system testing is performed with the Anomaly Detection code base to verify linkage between modules and that the project works with an established codebase.

Module 1: Test-to-Source

Although the overall success metric for our module is correctness of outputs, optimization of speed also lends itself well to a more complete and interactive developer experience when interfacing with this module.

Testing Method

The test to source module has two parameters that can be defined on runtime. The --doMappings flag takes either "true" or "false" as the output, indicating whether the system should attempt to query the database for existing mappings or redraw them using internal R scripts driven by COVR. In conjunction with the --doMappings flag, the --testCaseName flag takes a filename as input and outputs the functions mapped to that test case.

Functionality testing has been done through the means of unit testing. For each of the flags associated with this module, we exercise functionality for intended use cases, corner cases, and incorrect inputs to verify that the module is able to execute successfully or exit safely and inform the user of the problem. Using dummy data, we are able to populate the database with simplistic functions and tests to ensure that our internal mapping structure is correct. Using the assert command in python, we are able to run multiple tests across a single instance to automate the process of checking that the module produces expected mapping outputs.

Performance testing has been conducted as a measure of efficiency for our module. Since the --doMappings flag directs the system to either retrieve existing or produce new mappings, we utilized the built in timing library in python to measure runtime between the two flags. To simulate a real workload, we used our AnomalyDetection library during performance testing which consists of 12 test files. By combining the --testCaseName flag for each of these modules with the --doMappings flag assigned to true and then false, we systematically collected the runtimes for these two situations to simulate how long it would take to query the database for existing mappings and redraw mappings.

Results

Testing has been split for this module into functional unit testing and performance runtime analysis. As mentioned previously, our metrics are correctness first, and then performance, so both types of tests are necessary to verify the success of our tool.

The unit tests for this module are as follows: 6 tests verifying correctness of mappings using simplistic arithmetic and math functions with associated tests, and 4 tests exercising the interface of the module script on the command line. When running the combined test to source testing script, all 10 of these unit tests pass assertions associated with their expected outputs, thus verifying the functionality of our module.

Fig. 5 highlights the performance testing results for the AnomalyDetection test cases. For each of these test cases, we ran the test to source module with the --doMappings flag defined in the headers 10 times to standardize the execution time across runs. After collecting the mean runtimes for these tests, we once again averaged the times across tests to get the final values for the AnomalyDetection test suite. Redrawing all mappings in the database took on average 0.075184s while retrieving an existing mapping from the database took on average 51.57077s.

AnomalyDetection Test	doMappings=true timing (s)	doMappings=false timing (s)
test-NAs-0.R	0.070233s	50.77727s
test-NAs-1.R	0.068709s	51.46744s
test-NAs-2.R	0.069646s	51.65987s
test-NAs-3.R	0.072884s	51.40461s
test-edge-0.R	0.069529s	51.64313s
test-edge-1.R	0.078420s	52.05467s
test-ts-0.R	0.071355s	51.03300s
test-ts-1.R	0.763204s	51.49347s
test-ts-2.R	0.077705s	51.54903s
test-vec-0.R	0.079135s	52.10049s
test-vec-1.R	0.077434s	51.20021s
test-vec-2.R	0.072091s	52.03627s
Average	**0.075184s**	**51.57077s**

Figure 5. Test-To-Source Performance Results

Analysis

The functional unit testing success yields a deterministic true or false for each of the assertions, so a 10/10 passing record is consistent across all runs. Nonetheless, having these tests during the development of the module was how we attained this 10/10 scoring. Using these tests along with manual inspection allowed us to pinpoint a glaring mistake in our previous mapping methodology which prompted us to switch to the well documented COVR scripts in place of our own R scripts.

Redrawing mappings takes considerably more time than retrieving from the database. Running the --doMappings true flag would cause significant time overhead on the first run of a testing cycle, thus increasing our *M* defined in eq 1. On runs thereafter, we are then able to simply query the database for existing mappings with 685 times the efficiency. When pairing this increase in efficiency with the testing harness, we are able to compound this time saved to produce only a subset of tests that need to be run between code changes to reduce overall testing time.

Recommendations

Although our testing has revealed that the module operates within our expectations and has a reasonable time curve to overtake the brute force method of test execution, we will continue to develop more functional tests to mature the testing library of this module. By using more conditional calls, loops, and potentially interfacing with APIs, we hope to continue development to strengthen confidence in this module's correctness of mappings.

Module 2: Regression Test Selection

The regression selection module interfaces our project with Git, the version control system used by the underlying code base. The module also invokes other modules within the project. The testing plan is to ensure that it performs correctly in all operating conditions and is efficient.

The present implementation uses a built-in R utility, source(), to convert raw information from Git (line numbers of code changes) into usable information for test selection (R functions that were affected by the code changes.) The current implementation involves partial executions of the R functions. This is not ideal, primarily because it could take a long time for the execution to finish if the R code is work intensive.

We are still in the process of finding a better way to implement this module without executing anything from the R codebase. Once this is finished we will have more testing to perform.

Additionally, that will create an opportunity to compare the current testing results with results after the implementation has been changed.

Testing Method

The regression test selection module depends on the definitive changes made, and then committed within a codebase repository on GtHub. As a result, the testing process for this module uses both unit testing and manual verification to verify that specific changes in code correctly output a set of tests that exercise those changes.

Because this module pulls data from multiple external locations, such as the GitHub repository storing the codebase and the MySQL database that is used to store the mappings, we heavily tested the scripts that access each external location. Parsing the codebase to find the changed lines of code is dependent on the speed and accuracy of the "git diff" command that is built into GitHub. Accessing the database to find the tests mapped to source code is dependent on the number of current mappings in the database.

To test the files that call "git diff" to find changes in code, we used the AnomalyDetection project codebase to verify that the files are properly outputting the names of functions that have been changed in the most recent commit. To test the files that query into the database and return mappings, we created dummy mappings and wrote unit tests to verify that the files were properly returning test and function names. Then, using mappings created by the test-to-source module, we integrated the two processes within a wrapper script and manually verified that the changes in code led to a correct query of the tests to be run.

Results

From these testing methods, we found that the accuracy of the module depends on the "git diff" command returning the correct file names, as well as the correctness of the existing mappings in the database. On the other hand, we found that the speed of the module is highly dependent on a single file, contextLineNumber.R, that parses the source code to find function names based on the file names returned by "git diff". In order to parse the source code, contextLineNumber.R uses a function "source()" that essentially runs the entire codebase. As a result, the time it takes to run the module increases with the time it takes to run through the codebase. Because the source() function is called for every changed line in the project, the amount of changed lines also determines how long it takes the module to run.

Verification of which parts of the module took the longest time is detailed in Fig. 6. The most recent commit in the AnomalyDetection project shows that 18 lines were changed. As a result, we averaged the execution time of five distinct parts of the module over 10 runs each, for the scenario of up to 18 lines changed in the AnomalyDetection codebase.

	Time it takes to execute (seconds)				
Number of Changed Lines	**source() to find function names (in test selection)**	**"git diff" to find changed lines (in redraw mappings)**	**Querying into MySQL database**	**Running Module2 --mode testselection**	**Running Module2 --mode redrawmapings**
1	0.192 s	0.0199 s	0.00128 s	0.1887 s	0.0177 s
3	0.539 s	0.0206 s	0.00128 s	0.5287 s	0.0183 s
6	1.0599 s	0.0207 s	0.00235 s	1.0281 s	0.0191 s
9	1.518 s	0.0203 s	0.00516 s	1.536 s	0.0177 s
12	2.101 s	0.0207 s	0.00489 s	2.051 s	0.0184 s
15	2.637 s	0.0205 s	0.00963 s	2.575 s	0.0171 s
18	3.304 s	0.0211 s	0.00899 s	3.062 s	0.0194 s

Figure 6. Regression Test Selection Performance Results

Analysis

Fig. 6 details the timing tests done on the different aspects of module 2. Calling the source() method has a major effect on the timing of the test selection, as the table shows how the timing of calling source() is almost exactly equivalent to the timing of executing the overall module 2 test selection mode. Additionally, Fig. 6 shows how querying into the database has a marginal timing effect on the overall execution of the test selection execution.

While correctness remains the top priority, the performance of this module would still be improved by making it faster. The tests we performed for the correctness passed and validated that the basic functionality of this module was working in the current configuration. Since the bulk of the module's execution time was related to the source() command in contextLineNumber.R, as mentioned above, our time testing underscored the fact that we should

research alternative ways to implement the conversion from Git code change output to R function names.

There are two modes of execution in this module and one of them, redraw mappings, is very lightweight. Fig. 6 shows that the difference between running redraw mappings and test selection on the AnomalyDetection project is about 3.04 seconds. Since the only output needed for redrawmappings is a list of files changed, this execution mode needs only to execute git diff and parse the result. That explains why this execution mode did not take long, and means that we do not expect it to run much faster after the source() implementation is changed.

Recommendations

Regardless of the test results, it would greatly improve this module to replace the source() call with another means of accurately identifying R functions corresponding to code changes. This would avoid issues with incomplete and poorly written code that does not execute properly. It is quite clear that the module would run faster by not having to execute the R code in any case.

In the future, more unit tests should be deployed to regularly check on the project's correctness and performance. A suite that exercises it along both execution modes with a variety of inputs would be helpful. After the source() call is replaced, tests suitable to that replacement should be implemented.

Module 3: Source-to-Test

The Source-to-Test module's main job is to retrieve mappings from the database and to provide database helper methods for other modules. Exercising functionality of this module also entails the setup, teardown, and maintenance of the database members. The pytest library provided an easy way to construct and execute a variety of focused tests for portions of the module.

Testing Method

The Source-to-Test module and all of our code for interacting with the database uses the pytest library for unit testing and integration testing. For our usage, pytest makes it easy to set up the database and environment for each test and to assert that something has been stored or retrieved from the database correctly. For example, Fig 7 shows some of our pytest code that creates fake repositories, files, functions, and test cases; runs a test case; and then tears down the database.

For all of our code that interfaces with the database, we tested all of the functionality that another module or developer might need and all of the corner cases we could imagine. For example, for our *SourceFile* class that retrieves all tests for a given source file, we tested the normal use case where the file exists and is exercised by tests as shown in Fig 8. We also tested other corner cases where the file might not exist, there are no tests for a source file, or there are multiple source files with the same name but in different repositories.

```
@pytest.fixture
def with_database():
    Database.default_db = 'test'

    repo_id = create_repository()

    file1_id = create_file(repo_id, "one.r")
    file2_id = create_file(repo_id, "two.r")

    function1_id = create_function(file1_id, "doMath")
    function2_id = create_function(file2_id, "doOtherMath")
    function3_id = create_function(file1_id, "doSomeMoreMath")

    test1_id = create_test_case(file1_id, "first_test")
    test2_id = create_test_case(file1_id, "second_test")

    create_link(function1_id, test1_id)
    create_link(function1_id, test2_id)

    yield # Run the test case

    teardown()
```

Figure 7. Pytest code for setting up the database

```
def test_gets_the_test_names(with_database):
    sourceFile = SourceFile.get_by_file_path("one.r")
    expected = set(["first_test", "second_test"])

    assert expected == set(sourceFile.testCaseNames)
```

Figure 8. Example test case for retrieving test case names from source file name

Results & Analysis

The Source-to-Test module and the database classes have 30 tests between them, all of which pass, as shown in the Fig. 9 below. The Source-to-Test module has two requirements, if a source file exists, it will return the tests associated with the given file formatted as a JSON report. If a source file does not exist, it will return None. Our tests only verify the first requirement. The second requirement does not have any tests and thus cannot be verified as being met. The database manager classes were designed to replace all SQL queries used in the project, and simplify error checking when querying from the database. The results of our testing shows that these classes meet all design specifications. They return the correct values in the normal use cases and correctly handle corner cases.

```
❯ pytest-3
============================================= test session starts ==============================================
platform linux -- Python 3.6.9, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /home/jennings/Documents/seniordesign/ee464_diagnosing_errors, inifile:
collected 30 items

test/database_managers_test/function_test.py ........                                                 [ 26%]
test/database_managers_test/repository_test.py ...                                                    [ 36%]
test/database_managers_test/source_file_test.py .........                                             [ 66%]
test/database_managers_test/source_to_test_test.py .                                                  [ 70%]
test/database_managers_test/test_case_test.py .........                                               [100%]

============================================== warnings summary ===============================================
test/database_managers_test/test_case_test.py::TestCase
  cannot collect test class 'TestCase' because it has a __init__ constructor

-- Docs: http://doc.pytest.org/en/latest/warnings.html
================================== 30 passed, 1 warnings in 1.12 seconds ==================================
```

Figure 9. Pytest Test Results

Recommendations

Going forward, we will need to write tests to verify that the Source-to-Test module correctly handles corner cases, such as when a file does not exist. The database manager classes are completed, barring any future changes to specifications. Lastly, our test classes need to be refactored to make them more maintainable.

Module 4: Testing Harness

The testing harness ultimately serves as a wrapper for the three previous modules of this project to produce and execute a subset to exercise only necessary tests given a change in source code. Since the harness is mainly an interface to the other modules whose output is already standardized, our work testing the parts of the harness will also scale to the final product.

Testing Method

Although this module is still in development, we are employing the same test driven development mindset using dummy unit tests to verify functionality of the module in tandem with our expectations. We will also create integration tests to make sure that all four modules work well together. As the module nears completion, we will add in additional tests for error handling and corner cases.

Performance testing will also be heavily emphasized in this module as well. Using an adaptation of the scripts from module 1 (test to source) testing, we will compare the total runtime of the harness to the entirety of our AnomalyDetection test suite to verify that there are measurable efficiency gains after some N runs of the system.

Results, Analysis, Recommendations

Since this module is still in development, conclusive performance metrics are unavailable. As discussed previously, we are focusing on adapting functional tests from the modules as well as

developing new ones in tandem with the harness to ensure that internal and external outputs are within expectations.

CONCLUSION

The main deliverable of this project is a regression testing harness of three modules involving: 1) a tool for test-to-source mapping, 2) a regression test selection tool, and 3) a tool for source-to-test mapping. Rigorous functional and performance testing methodologies have been applied to all facets of these modules to ensure that they are operating within requirements and constraints set forth by the team. Looking forward, more unit tests would provide easier testing of particular areas of the project. As the project is improved and changed, tests should be made to adapt for that. Correctness of results remains the top priority, but efficiency can always be improved in the project's execution speed.

REFERENCES

[1] M. Gligoric, “Regression Test Selection,” Ph.D. dissertation, Dept. Comp. Sci., University of Illinois at Urbana-Champaign, Champaign, IL, 2015.

[2] The National Institute of Standards and Technology, (2002, June 28). “Software Errors Cost U.S. Economy $59.5 Billion Annually, ” [Online]. Available: http://www.abeacha.com/NIST_press_release_bugs_cost.html