DATE:	December 6th, 2019
TO:	Dr. Emanuel Tutuc, Ms. Rebecca Kurlak
FROM:	Canyon Evenson, Michael Flanders, Jennings Inge, Saarila Kenkare, John Koelling, Janvi Pandya
SUBJECT:	Triaging, Diagnosing, and Predicting Software Errors System Design Report

1.0 INTRODUCTION

This report establishes the system design of our senior project, “Triaging, Diagnosing, and Predicting Software Errors.” This project is partnered with Dell Technologies’ EMC division which provides enterprise and midmarket solutions for IT infrastructure and workforce technology [1]. The goal of our project is to build tools that improve upon Dell’s existing testing infrastructure by: 1) decreasing the time to run tests after software changes, and 2) increasing the amount of bugs found before software is shipped. By meeting these goals, we will decrease the human capital requirements of Dell’s testing process.

From Dell’s requirements, our research of prior art, and our views of an ideal testing tool, we have come up with a modular design of a regression test selection tool for the R programming language. To help implement our design, we have created project management strategies and performed small experiments to reduce potential risks. This report outlines these project management strategies such as project activities, assignments, scheduling, and budget that extend through next semester. We have also investigated potential problem areas—e.g. interacting with the database, using a code coverage tool, parsing and splitting up tests, parsing changes between code revisions, and building call graphs of the system under test—and created proof-of-concept tools to show the feasibility of our design and reduce potential risks.

2.0 DESIGN PROJECT BACKGROUND

Our project aims to create an efficient system to handle bugs in a codebase and shorten the time it takes developers to run tests. We aim to accomplish this by selecting specific tests for a developer to run when they make a code change, so that all tests don't have to be run, resulting in saved time. We select tests based on what functions they cover with the help of a code coverage tool. Our final deliverable for this project will be a testing harness made up of three distinct modules, each relating to a mapping of source code to tests. Each of these deliverables has been designed in conjunction with the industry contact, along with a set of specifications.

2.1 Design Problem

Software is constantly changed to include new features or satisfy new requirements. To ensure that these changes did not break previous functionality, software developers will run tests to check their correctness. This is called regression testing [4]. Because regression testing is a time consuming process, developers will often choose to run a subset of the total tests for a software application. However, poor test selection can lead to certain areas of the application not being properly verified, and can cause certain bugs to go unnoticed until after the product ships [4]. This then leads to vast increases in testing and maintenance costs, as bug detection, reporting, and patching become much more expensive the further down the line bugs are identified [3]. Automated regression test selection can mitigate these problems by ensuring that all relevant tests are run each time a software application is changed.

2.2 Deliverables

The team's industry contact at Dell has specified four deliverables as our main goal:

1. The first deliverable is a module that maps a software application's tests to its methods. To create this mapping, the module will run the test suite of an application and track which functions are called by each test. An example application and a model of its interactions are shown in Fig. 1.

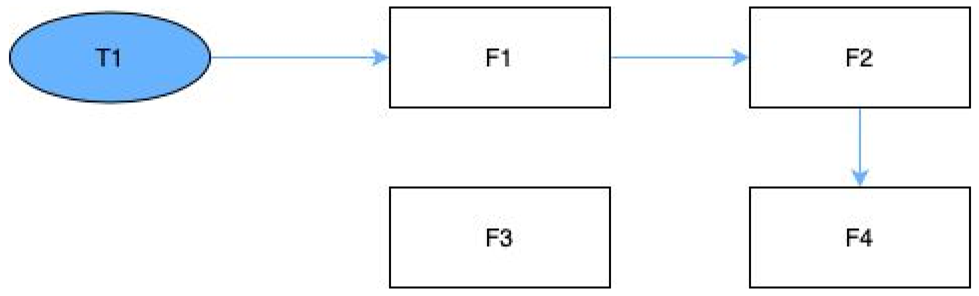


Figure 1. Test to Source Mapping

This application contains one test T1 and four functions F1, F2, F3, and F4. T1 calls functions F1, F2, and F4 but does not call F3, therefore, this module would create a mapping of T1 to F1, F2, and F4.

2. The second deliverable is a module that specifies which tests should be selected for regression testing after an application is modified. This module uses the mappings created by the previous module to connect modified functions with the tests that execute them. Then, the module outputs all tests found in the previous step. An extension of the previous application, with a new test T2 that calls functions F3 and F4 is shown in Fig. 2.

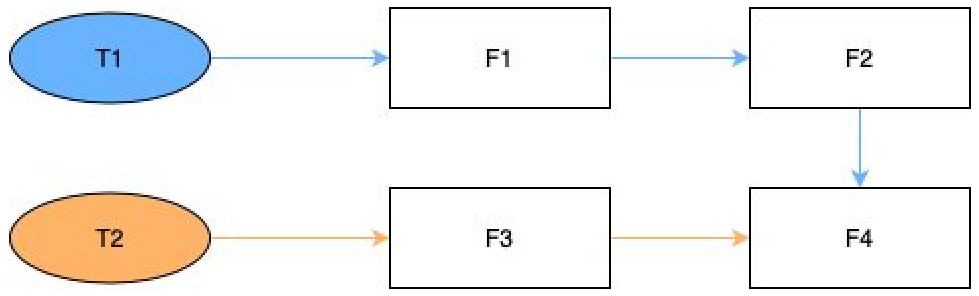


Figure 2. Test to Source Mapping (Multiple Tests)

In this example, if function F4 is modified, then tests T1 and T2 are selected for regression testing. Alternatively, if function F2 is modified, then only test T1 is selected.

3. The third deliverable is a module that takes a source code file as input and returns the tests that exercise the functions in the file. This module also uses the mappings created by the first module. For every function in a source file, this module looks up the tests associated with it and outputs each one. An associated test is one that calls any of the functions in the given file. Fig. 3 extends the example application again so that it has two sources files S1 and S2. S1 contains functions F1, F2, and F4 while S2 contains function F3.

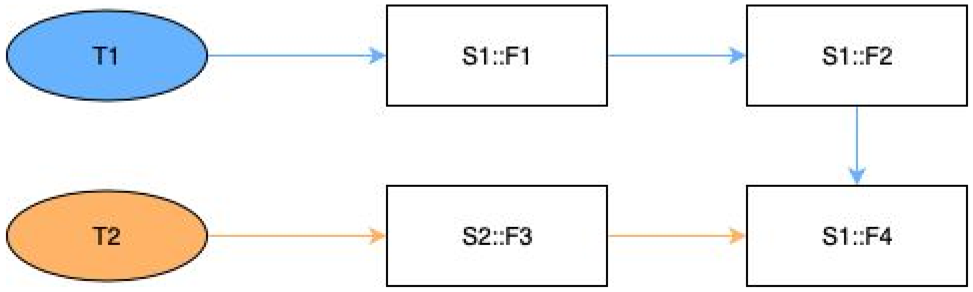


Figure 3. Test to Source Mapping (Multiple Source Files)

Given source code file S1, this module would return tests T1 and T2. If S2 was the input, then only T2 would be returned.

4. The final deliverable is a testing harness that combines all of the previous modules. The harness should execute the test suite of an application, collect code coverage information, and construct the test-to-source mapping shown in Fig. 3.

This tool will be used by engineering teams at Dell so code documentation, usage guidelines, UML flowchart diagrams, and UML class diagrams will also be provided to ease usage and further development. The remaining tools will have particular deliverables identified upon completion of the first tool, and will be determined in coordination with our industry contact.

2.3 Specifications

Our project has four deliverables with unique specifications for each. For the operating environments, specifications will stay consistent across all modules. We developed these based on requirements from Dell as the tools need to fit within the development environment, both on Windows and Linux. Performance specifications were developed by researching prior art and determining different metrics based on the module. As each module has a different purpose, inputs and outputs will differ. This section details specifications created by the team in conjunction with the Dell industry contact (Appendix A).

2.3.1 *Input and Output Specifications*

There are four input-output specification pairs provided, one for each deliverable, as outlined in Table 1. For deliverables (1) and (3), the inputs will the name(s) of source file(s) and the output will be the names of tests. The regression test selection module, deliverable (2), will take the differences between two code revisions and output the names of all tests to be rerun. The last deliverable, the harness that ties everything together, will take the entire code base and test suite as input, and will output a mapping of test cases to their respective source code files as well as code coverage information. A more general input-output specification is listed in Table 1. Each input into any one of these modules will be specified via a command-line interface, and the developer will be able to view the reports generated by the tool.

Table 1. Input and Output Specifications for Items From "Deliverables" Section

	Deliverable 1 (Test-to-source-code mapping)	**Deliverable 2** (Diff-to-Test mapping)	**Deliverable 3** (Source-code-to-test mapping)	**Deliverable 4** (Harness)
Inputs	Test cases or test suite files	Version history code diff	Source code file(s)	Full test suite and product source code
Outputs	Source file(s) covered by tests	Tests that should be ran as regression tests	Test cases or test suite files that exercise the file(s)	Mappings of test cases to source code files and code coverage %

2.3.2 ***Operating Environment Specifications***

The tools are meant to be used during the software development life cycle by Dell's testing team and developers. Dell develops their products in Windows and Linux, therefore our tools will be compatible with both operating systems. All work will be done using Dell resources. The product source, build and code coverage tools, and needed execution environments will all be provided in existing virtual desktop environments.

2.3.3 ***Performance Specifications***

The performance of the final product will be based on accuracy and speed. Since the main objective of deliverable (2) is to decrease testing time by only running relevant tests, the module will meet performance expectations if the time required to select and run these regression tests is less than the time required to run all the tests [4]. Additionally, the output of the other tools used to map test cases to source code files (and vice versa) will be inspected to see how accurately it identifies relevant sections of code. Additional performance metrics will be determined once the project is functional and work begins on any of the stretch goals.

3.0 SYSTEM DESIGN

Once the team fully researched the project background and determined specifications for each deliverable, we began to investigate different ways we could implement each module. After researching prior art and evaluating various software tools, we have decided to use Python as the implementation language, a MySQL database for storing the outputs between runs for storing the outputs between runs, and covr as the R code coverage tool which will assist with creating mappings of source code and tests. The three modules that form the testing harness will be implemented in Python and read/write from a MySQL database. Once we got our design ideas approved from our industry mentor, we completed our risk reduction experiments: small, modular tools which implemented some function relevant to our project. Each of these experiments deals with processing R source code and test suites, a fundamental part of the project, and manipulating it to get meaningful results. Based on the results of these experiments, for each deliverable, we have some proof-of-concept done and have finalized which tools we will be using.

3.1 Design Concept

Our final design—shown in Fig. 4—of a regression test selection (RTS) tool for the R programming language is composed of four distinct modules: (1) a test-to-source mapping tool, (2) a diff-to-test mapping tool, (3) a source-to-test mapping tool, and (4) a test harness that combines the previous three tools. While numerous other sources [4-8] helped us make informed design decisions, we drafted our own modular design based on Dell's requirements and our perceptions of developers needs for a testing tool. We chose a modular design because it clearly encapsulates related functionality, is easier to maintain and test, and allows us to build separate tools that each fulfill one of Dell's requirements. Furthermore, the design allows our end users to use any of the first three modules directly or through a testing harness that provides a simpler interface.

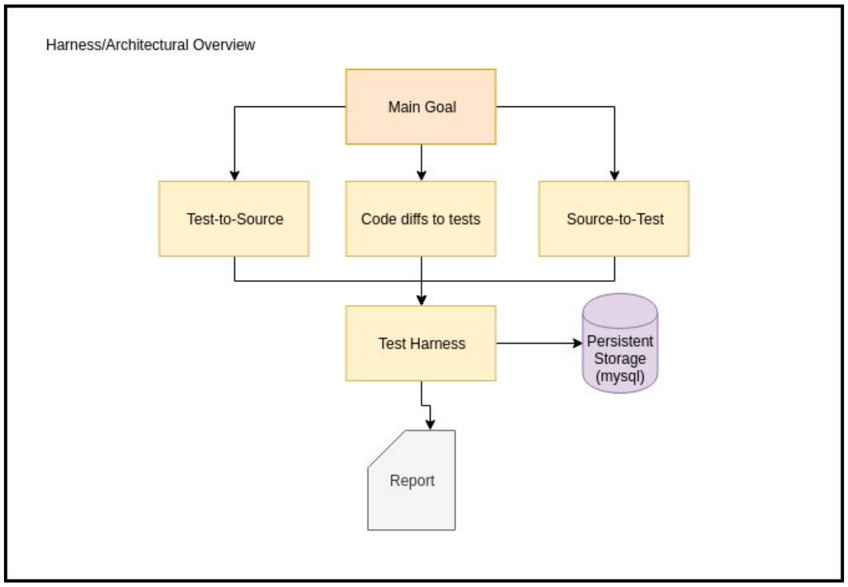


Figure 4. High level overview of our design for a regression test selection tool

3.1.1 *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. 5. The testing system will then be able to efficiently detect the origin of a test failure based on the function calls within that test. The input to the module is a specific test case, or test suite, and the output of the module is a report detailing which areas of 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. The use of persistent storage will stop the testing harness from re-executing mappings that have already been identified, and thus help create a faster, more effective testing tool.

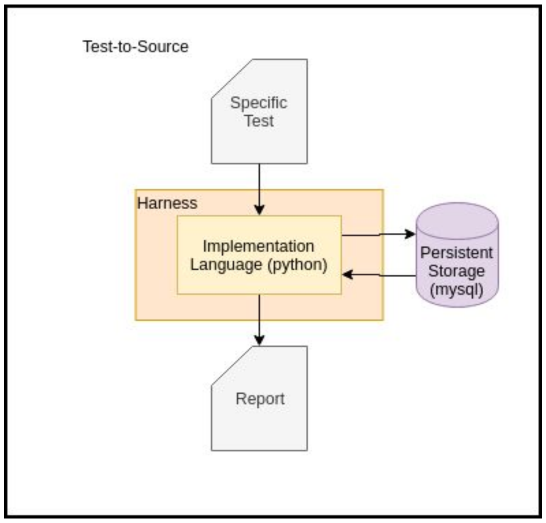


Figure 5. Test to Source Mapping

3.1.2 *Module 2: Differences in Code*

As shown in Fig. 6, this module will take a revision of code and select associated tests. Git is the version control system used by Q, and so this module will integrate with Git’s built in revision information tools. This also will use a persistent storage module, which records which areas of code are associated with each test. Based on the areas of code changed, this tool will recommend tests to run those fragments of the program.

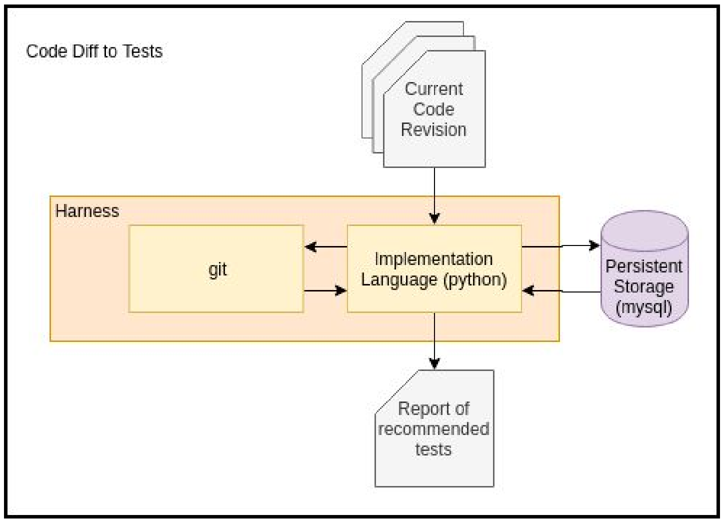

Code Diff to Tests
Current Code Revision
Harness
git
Implementation Language (python)
Persistent Storage (mysql)
Report of recommended tests

Figure 6. Code Diff to Tests

3.1.3 *Module 3: Source-to-Test Mapping*

Fig. 7 shows the setup of the source-to-test module. The input is a specific R source code file. This file is parsed using a Python script to identify all of its functions. For each of these functions, the mapping already stored in the database is queried to find all test cases exercising any of them. This mapping is generated using covr, which can deliver a report of which functions were covered by a particular test case. A MySQL database will be used to ensure that we will not have to rebuild the mappings that already exist in the database. Then, a report is delivered to the user of which test cases map to the given source file input.

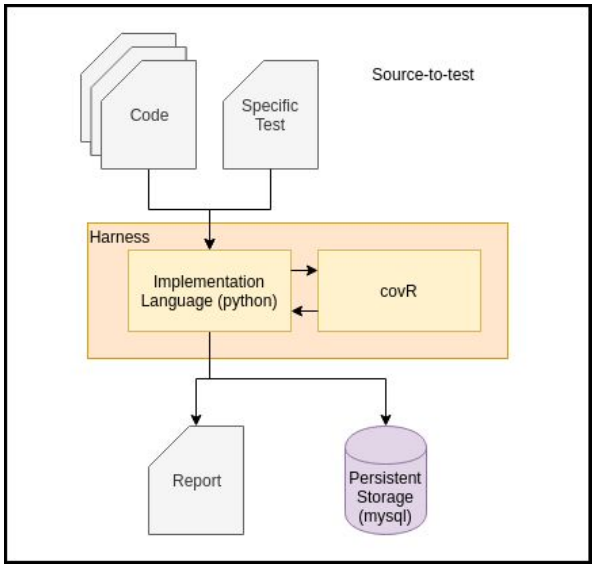


Figure 7. Source to Test Mapping

3.1.4 *Module 4: Testing Harness*

The testing harness combines the three previous modules to create a simplified command line interface for Dell's engineers to interact with. The harness will interact with the codebase, test files, and a database to: execute tests for the product, collect code coverage information, and construct test-to-source mappings as previously specified.

The harness will use some of Python's command line parsing libraries to figure out which tool or task the user wants to run. For example, if a developer wants to reconstruct the test-to-source mappings after making changes to the codebase, they might type, "python TestingHarness.py --reconstruct-mappings". Since Dell has specified that a MySQL database will be available for our tools, the harness will also interact with this database for storing code coverage data across multiple revisions.

We have decided to design the harness to perform regression test selection at a function level granularity. This means that if there are multiple functions in one file, any changes to one particular function will only retrigger the tests that deal with the changed function. This is in contrast to the other designs we looked at [4, 5]. In [4], the author surveyed several other RTS tools and showed that a file level granularity is preferable as it can achieve the same level of correctness in a shorter amount of time. However, the design presented in [4] deals mostly with class oriented Java code bases which naturally lend themselves to a file level granularity whereas R code bases are very function oriented. On the other hand, the design presented by [4] performs an eleven step process to select tests which we believe would be too slow for R code bases and cause us not to meet our speed performance metric.

3.1.5 *User Interface*

All tools will be interacted with using a command line interface. Users will have the option to tweak a tool's behaviour by specifying different command line arguments. Since all tools will require files, tests, or codebase revision history to analyze, users will also specify these target files as command line arguments.

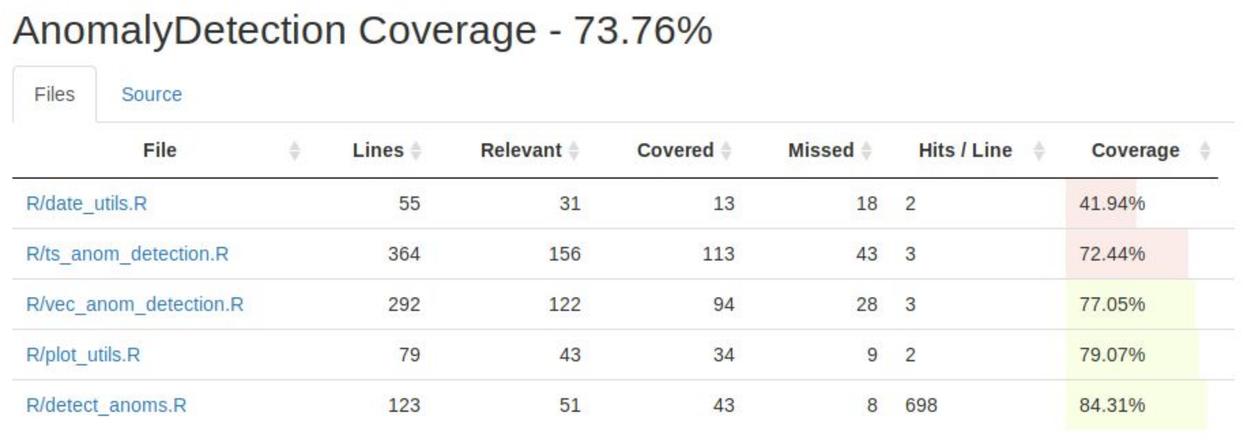
3.2 Risk Reduction

For the project as a whole, there are multiple steps that need to be taken for the goal to be completed, some of which are understanding R code and testing, using code coverage tools to map code to tests, creating persistent storage through a database, and parsing tests. We completed nine risk reduction experiments, three as a team and six individually to work on small problems related to our project. The goal of each of these was to reduce risk so that when we begin work on the real codebase, we have already selected tools and have concrete plans for implementation for each module. Our five most relevant and successful experiments, all relating to processing R source, are detailed in this section; each one implements a tool which will serve useful on this project.

3.2.1 *R Code Coverage*

One of the keys to reducing risk for this project was understanding how to extract meaningful information from our R source code. Since none of our team has had prior experience with the R coding language, we applied prior research over static analysis in object oriented languages such as Java to ultimately decide on covr. This R library allows us to extract code coverage percentages based on package, file, function, or even line granularity and will greatly assist us in our deliverable of mapping source code to tests.

In our first group risk reduction experiment with covr, we demonstrated a working understanding of the functionalities of this library within the R studio development environment. By using a simple example involving test cases of arithmetic operations, we were able to extract code coverage information in both raw text and formatted HTML reports that provided us with percentages based on the total code exercised by a singular test, as shown in Fig 8. These metrics of code coverage impact our final project design in terms of creating our mappings between source code and test cases. In practice, a non-zero output from these code coverage runs would cause us to create a mapping between the given test and source code function aligned with the goal of Module 1.


AnomalyDetection Coverage - 73.76%

File	Lines	Relevant	Covered	Missed	Hits / Line	Coverage
R/date_utils.R	55	31	13	18	2	41.94%
R/ts_anom_detection.R	364	156	113	43	3	72.44%
R/vec_anom_detection.R	292	122	94	28	3	77.05%
R/plot_utils.R	79	43	34	9	2	79.07%
R/detect_anoms.R	123	51	43	8	698	84.31%

Figure 8. covr HTML Report

Although our prior work with covr was integral to our understanding of the package, our group risk reduction experiment sought to automate this process by removing it from the R studio development environment and to automate the process and minimize human interaction to obtain persistent code coverage outputs.

The first objective of this experiment was to detach the functionality of covr from the R Studio development environment. Although it seems as if R and RStudio are synonymous, R is comparable to the engine that drives a vehicle while RStudio is the user interface. After stripping down to the essentials of R, we were able to execute the setup commands for the prior risk reduction experiments through using the source command in R. Since these commands are exactly the same across a given project, these scripting capabilities can be run in conjunction with other processes to produce the desired code coverage metrics when placed in the working directory of the test cases and source code.

The final objective of this risk reduction experiment was to ensure the persistence of covr outputs after an execution of our scripts. After separating covr from the R Studio development environment, the outputs from a run of covr would ultimately produce coverage information via output on the command line, or in a formatted report viewed via the default system web browser. Although this information is exactly what we desired, our quest for automation led us to seek better means of persistence rather than copying outputs from the command line and manually saving HTML files. Utilizing the sink and report commands with modifiers to uniquely identify covr runs, we were able to achieve uniform outputs that persisted after closing the parent terminal that can be interpreted by humans and other modules of our projects to produce the percentage coverage on a method and file level granularity.

3.2.2 *R Code Parsing and Database Storage*

As mentioned previously, we made the decision to store our mappings and information about the given codebase in a database. The basis for one of our risk reduction experiments was to create a setup for this database based on a codebase. Then, the objective was to write a Python script reading information about an R codebase and store it into this database. We used the Python library PySQL to interface with a MySQL database.

We decided on a schema that stored information about functions and test cases in the codebase, as shown in Fig. 9. Each box represents a table, and each line between two tables represents a field that links between the two tables. One table, Repositories, keeps a list of each repository and its path. A repository can have multiple files, of which the relevant ones are either source code files or test case files. In this case, the "repositoryID" field serves as a link between the Repositories and RFiles table. All files are stored in RFiles with a field indicating if the file contains code or tests, and each file is linked to a repository by a unique number. If a file is a source code file, it can contain multiple functions, and for each of these, there will be an entry in the RFunctions table with the fileID field linking back to the file the function is in. Similarly, for each test case in a test file, there will be an entry in the RTestCases table.

The table at the bottom of the figure links to both the RTestCases table and the RFunctions table. An entry in the RCodeToTestCases table means that the test case represented by the ID exercises the function represented by the function ID. Since a test case can exercise multiple functions and a function can be exercised by multiple test cases, there may be multiple entries for each ID.
We used the previously mentioned Anomaly Detection R codebase and used it as an input to a Python script which went through all files, found functions and test cases, and stored them in the database according to the schema explained above.

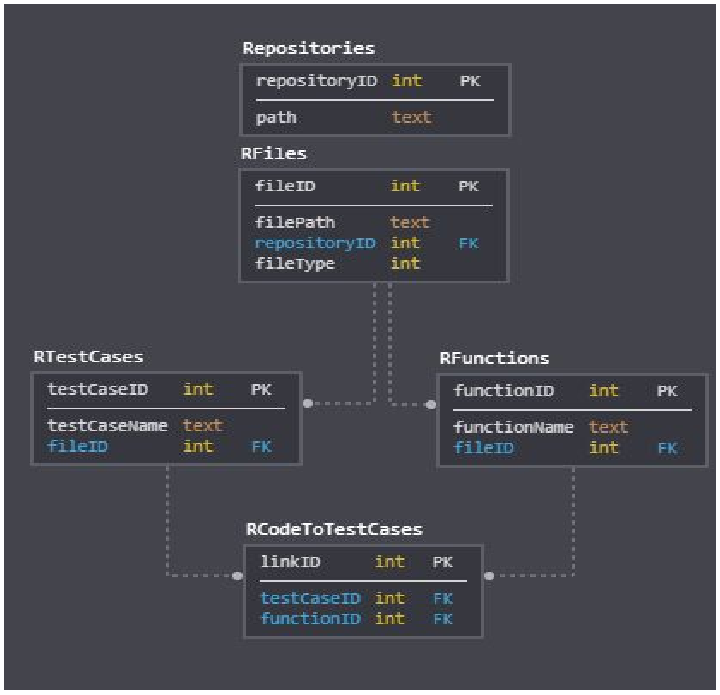


Figure 9. Schema for R Code Database Risk Reduction

Then, we modified the codebase so that the R functions and test cases printed out their own names when run. When running the unit tests, a log file was generated containing all of this information. Then, a Python script was written to parse this log file. From the order of the print statements, we were able to determine which test cases called which functions. The determined mapping was then stored in the database.

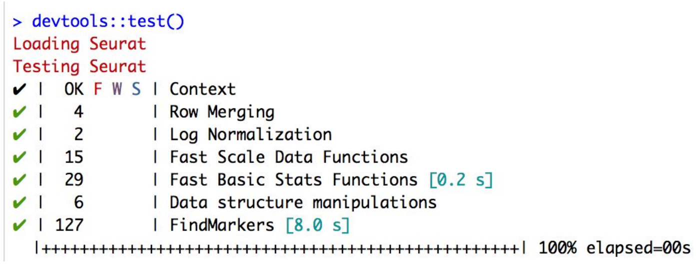
Finally, the script allowed for the user to input a source code file. Based on the mapping, it found and outputted all test cases exercising any of the functions in the file, all without parsing the actual code, since the information was already stored.

This risk reduction experiment served as a test of the source-to-test mapping module of the testing harness. In addition, it served as a proof-of-concept for using a database to store mappings. Although the schema and behavior may not stay the exact same, the general idea will serve useful when implementing this deliverable.

3.2.3 *R Test Parsing and Splitting*

The ability to parse a test file containing dozens of tests and split each test into its own file is critical to meeting our speed requirements. A common practice in software testing is grouping all tests with similar functionality into one file or test suite. While this makes the tests easier to read and understand for developers and testers, this is a problem for regression test selection as it makes collecting code coverage for individual tests much more difficult.

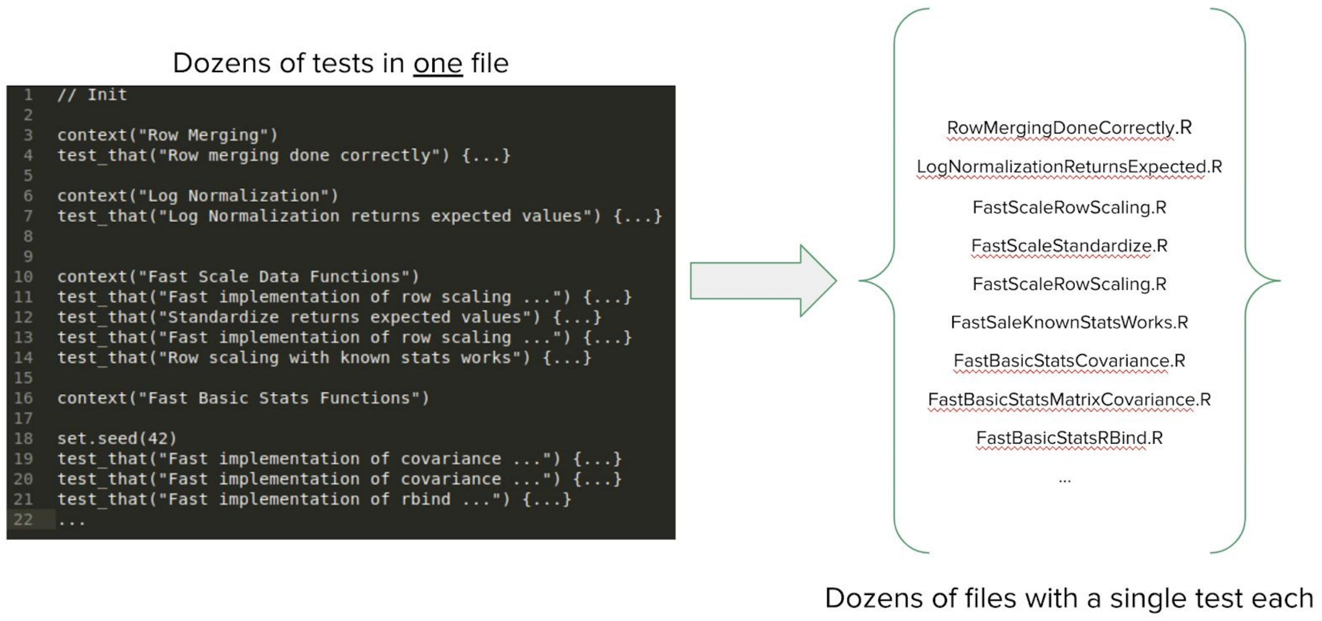
Fig. 10 shows the number of tests per file for Seurat, an R toolkit for single cell genomics [9]. The file for testing markers contains 127 tests and the file containing tests for fast basic statistical functions has 29 tests. Running our test-to-source mapping tool on these files will report an incorrectly large amount of code coverage as the reported coverage will actually be for these 127 tests rather than each individual test.



```
> devtools::test()
Loading Seurat
Testing Seurat
✔ |  OK F W S | Context
✔ |   4       | Row Merging
✔ |   2       | Log Normalization
✔ |  15       | Fast Scale Data Functions
✔ |  29       | Fast Basic Stats Functions [0.2 s]
✔ |   6       | Data structure manipulations
✔ | 127       | FindMarkers [8.0 s]
  |+++++++++++++++++++++++++++++++++++++++++++++++++++| 100% elapsed=00s
```

Figure 10. Number of test cases per file for R library, Seurat [9].

To overcome this problem, we built a tool to parse each test file and move each of these 127 test cases into 127 new, separate files each containing one test case as shown in Fig. 11. We can then run each of these test files through our tools and collect much finer grained code coverage data. Finally, we can use this new data to select only the tests covering new, changed code rather than having to rerun all 127 tests.

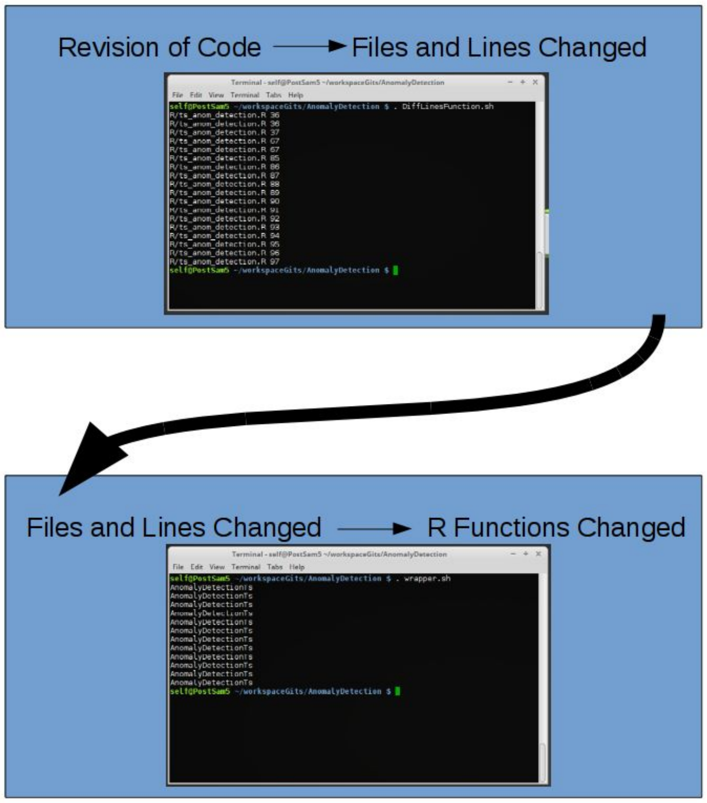

Dozens of tests in one file
RowMergingDoneCorrectly.R
LogNormalizationReturnsExpected.R
FastScaleRowScaling.R
FastScaleStandardize.R
FastScaleRowScaling.R
FastSaleKnownStatsWorks.R
FastBasicStatsCovariance.R
FastBasicStatsMatrixCovariance.R
FastBasicStatsRBind.R
...
Dozens of files with a single test each

```
// Init

context("Row Merging")
test_that("Row merging done correctly") {...}

context("Log Normalization")
test_that("Log Normalization returns expected values") {...}


context("Fast Scale Data Functions")
test_that("Fast implementation of row scaling ...") {...}
test_that("Standardize returns expected values") {...}
test_that("Fast implementation of row scaling ...") {...}
test_that("Row scaling with known stats works") {...}

context("Fast Basic Stats Functions")

set.seed(42)
test_that("Fast implementation of covariance ...") {...}
test_that("Fast implementation of covariance ...") {...}
test_that("Fast implementation of rbind ...") {...}
...
```

Figure 11. Example test file for R library, Seurat [9], and some files created after splitting up tests.

3.2.4 *Git Diff and R Function Identification*

Since our tool would provide a way to target specific tests appropriate for a revision of code, we would need a way to extract meaning from the Git version control system about the newest revision of code. And since our tool will measure code coverage at a function level of granularity, we need to produce information in regard to R functions. This risk reduction experiment was a proof of concept for that area of our project. Two tools and a wrapper harness were made. The overall arrangement of this system is shown in Fig. 12.


Revision of Code → Files and Lines Changed
Terminal - self@PostSam5 ~/workspaceGits/AnomalyDetection
File Edit View Terminal Tabs Help
Files and Lines Changed → R Functions Changed
Terminal - self@PostSam5 ~/workspaceGits/AnomalyDetection
File Edit View Terminal Tabs Help

```
self@PostSam5 ~/workspaceGits/AnomalyDetection $ . DiffLinesFunction.sh
R/ts_anom_detection.R 36
R/ts_anom_detection.R 36
R/ts_anom_detection.R 37
R/ts_anom_detection.R 67
R/ts_anom_detection.R 67
R/ts_anom_detection.R 85
R/ts_anom_detection.R 86
R/ts_anom_detection.R 87
R/ts_anom_detection.R 88
R/ts_anom_detection.R 89
R/ts_anom_detection.R 90
R/ts_anom_detection.R 91
R/ts_anom_detection.R 92
R/ts_anom_detection.R 93
R/ts_anom_detection.R 94
R/ts_anom_detection.R 95
R/ts_anom_detection.R 96
R/ts_anom_detection.R 97
self@PostSam5 ~/workspaceGits/AnomalyDetection $
```

```
self@PostSam5 ~/workspaceGits/AnomalyDetection $ . wrapper.sh
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
AnomalyDetectionTs
self@PostSam5 ~/workspaceGits/AnomalyDetection $
```

Figure 12. Tools for Git Diff and R Function Identification

The first tool was a parser that extracted a list of filenames and lines changed for a given Git revision. Git's builtin output produces a variety of data in a human-readable format. Although command line options allow for machine-readable output, such variations do not include revision detail with line level granularity. This tool filtered the full, human-readable output and reformatted it into a machine readable format.

The second tool was a lookup script that converted a list of filenames and lines changed into R function names. The tool reads through R source files (specified earlier by filename) and automatically parses their structure. For each line looked up, the tool will return the R function that the line of code belongs to. This is useful because a line changed can be associated with the R function whose behavior would be modified by that change.

The wrapper harness executes both tools in series. It looks up the most recent Git revision of code, and by running the second tool with input from the first, it produces a list of R function names modified by the most recent revision. This makes the change sensible for the rest of our program, and allows association with our test coverage which is specified at a function level. This was a highly valuable risk reduction experiment because it proved that this juncture was actually feasible -- a conversion between Git's language agnostic output and R function names as required by the rest of our system.

3.2.5 *Call Graphs in R*

Mapping is a consistent design topic used throughout our modules, and is a vital resource for developing an efficient and accurate testing system, such as our test harness tool. Moving forward with our design, we decided to explore different ways to produce a high level overview of function calls within a collection of R code files. We soon discovered that generating a call graph of the R functions and test suite was the most effective way to get an idea of what code the test functions are calling. Furthermore, call graphs gave us a first step in understanding the mappings among the source code and tests.

A call graph is essentially a plot that outputs information about method calls, such as what functions call what other functions from within. By applying a call graph at the end of a test suite, we were able to see what functions each test had called within that test suite. In Fig. 13, the call graph plot visualizes the mainTest() suite calling test1() and test2(), as well as the fact that test1() calls the main() function, but test2() does not. The lines on the call graph represent a higher level function calling a lower level function, and the colors of the lines indicate at what level is the function being called from.

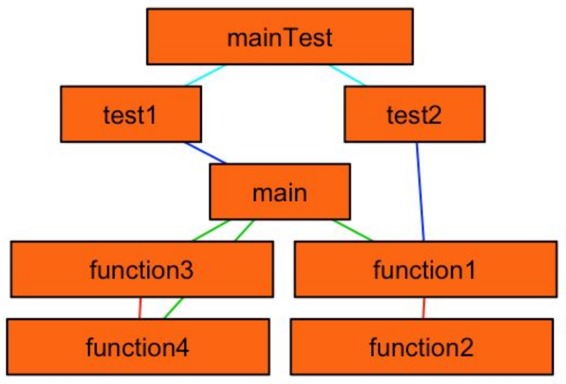


Figure 13. Example Call Graph in R

While call graphs will help our project begin the steps into a high level approach of method mapping, call graphs can also be used as a way to primarily understand the source code that we have been given. By generating a call graph that easily shows which functions were called by what, what functions were called the most or the least, we are able to greatly reduce the cost of each member of the team attempting to understand the make-up of the source code.

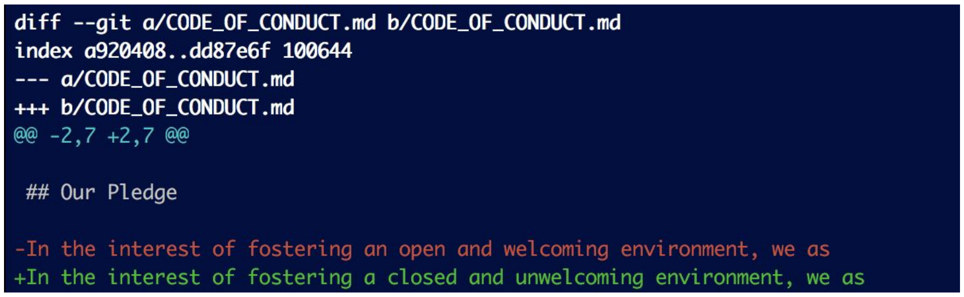
3.3 Test Plan

The focus of our testing will be on the: (1) test-to-source, (2) diff-to-test, and (3) source-to-test modules. . Modules (1) and (2) are difficult to test as they require test oracles. Test oracles are input-output pairs that are encoded as test cases and previously known to be correct. The following list represents the steps necessary to obtain test oracles for these first two modules.

1. Write a test case that exercises a handful of functions.
2. Run the test case and collect the code coverage information.
3. Manually inspect the collected coverage data.

If the collected coverage data is incorrect, then we will have to debug and fix the code until the output from step 2 is correct. Once the output is correct, we will encode the test case and the observed, correct output as a new test case.

For module (2) we will need to test that we are correctly parsing the differences between two code revisions and correctly identifying the changed file and the tests that exercise this changed file. An example of a test for module (2) would require code differences to be saved in a test case—e.g. the one saved in Fig. 14—and then checking that our code correctly identifies that the changed file is “CODE_OF_CONDUCT.md”.



```
diff --git a/CODE_OF_CONDUCT.md b/CODE_OF_CONDUCT.md
index a920408..dd87e6f 100644
--- a/CODE_OF_CONDUCT.md
+++ b/CODE_OF_CONDUCT.md
@@ -2,7 +2,7 @@

 ## Our Pledge

-In the interest of fostering an open and welcoming environment, we as
+In the interest of fostering a closed and unwelcoming environment, we as
```

Figure 14. Differences Between Two Code Revisions

4.0 PROJECT MANAGEMENT

The team has several technical goals for next semester along with multiple writing assignments and presentations for EE 464K. To effectively manage each of these and ensure that we are on track, we have created a Work Breakdown Structure, Linear Responsibility Chart, and Gantt Chart based on the tasks for our project and the EE 464K syllabus. This guarantees that for any task, we know exactly when it’s due and what work each of us needs to do on it. This may change if we gain new information about our project or the syllabus changes, but we will update each of these tools accordingly.

4.1 Project Activities

The Work Breakdown Structure (Appendix B) contains all tasks necessary for this project from the time of submission of this document until the end of the spring semester. The only activity remaining this semester is the Project Video. Activities next semester include both technical activities for the actual completion of our project and reports, both oral and written, for assignments in EE 464K.

As stated previously, we have four software modules necessary for the completion of our project: a test-to-source mapping module, a regression testing module, a source-to-test mapping module, and a testing harness combining each of the previous three. For each of these, we plan on first designing the architecture together and in conjunction with our industry mentor. Then, we will spend time developing our system, and then write a testing system to ensure each system works. The three modules that are part of the testing harness will be worked on in parallel, each by two people on the team. Once these are complete, we will again work with our industry mentor to plan how the overall harness will use the three modules, then work on implementing it together.
We have three major writing assignments due next semester: the Design Description Report, the Test and Evaluation Report, and the Written Final Report. For each writing assignment, we will write an outline together, then prepare a draft. We will review the entire paper together, bring the draft to our writing TA to get their feedback, then revise it according to each others' and the TA's feedback, and submit it. We will also follow this same workflow when creating our Design Poster, and make sure to get as much feedback as possible after creating a draft.
We also have multiple presentations next semester: the Oral Design Review, Mid-Project Demo, and the System Demo. For each presentation that we will do next semester, we plan on writing an outline of the slide deck and other materials, preparing a practice script, then rehearsing it to ensure that we are all in agreement.

4.2 Task Assignments

All members of this senior design team have technical backgrounds in software development and have nearly identical qualifications due to taking similar courses, therefore assignment of tasks were based on individual interest. The Linear Responsibility Chart (Appendix C) provides a clear indication as to those primarily responsible, contributing, and receiving information for a given task.

Since this project is operating within a Dell development ecosystem, we have followed suit with an agile development model based on an iterative and incremental approach to developing our project. Leadership and ownership of tasks are paramount for success in an agile development environment, so those who are primarily responsible for a task are expected to stand up in front of the team on a weekly basis and provide a transparent status report. Work on the overall testing harness has been split between subteams of two that will focus on development of each module of the assignment. Module 1 (test to source mapping) will be handled by John and Canyon, Module 2 by Saarila and Janvi, Module 3 by Jennings and Michael, and Module 4 will be a full team collaboration involving merging the prior three modules into a final testing harness. Within the development of each module, we have indicated the subtasks of architecture design, development, testing, and documentation. To ensure equal distribution of work, subteams have been divided to where each member will be primarily responsible for exactly two of the subtasks and contributing to the others. During the implementation of Module 4, task assignments were ultimately given on a voluntary basis since all members are expected to significantly contribute to the overall harness.

In terms of writing assignments and oral presentations, our general methodology for completing these assignments is to outline, draft, and finalize the work. During the planning and writing stages of these assignments, individuals assigned to a module will take lead. Since all members must be proficiently informed about the holistic development of each module, this system ensures the remaining group members are meaningful contributors.

4.3 Project Schedule

Remaining work for this project is in two areas: EE 364 work until the end of this semester, and EE 464 work throughout the spring semester. Only one task remains for EE 364—namely, the project video—which will be finished by December 9, 2019. Various course related assignments will be completed according to the course schedule in EE 464. The entire project schedule is itemized and scheduled in the Gantt Chart, Appendix D.

During EE 464, the project itself will be created. This broader task is subdivided into four deliverables, each of which has four phases. The phases are Architecture Design, Development, Testing, and Documentation. Three of the deliverables can be developed in parallel, and we will divide our team into three sub-teams as described above in Task Assignments. The fourth deliverable—the testing harness—will depend on the previous three, and therefore must wait until those are finished, which should occur at the beginning of spring break. After development of the testing harness is finished, testing will begin on both the testing harness itself and the project as a whole. To test the entire project, functional tests will be used (simulating actual usage by a developer) as well as any other tests that exercise the whole program as a contiguous system.

During the Architecture Design phase, which is the first phase of each deliverable, general specifications for the deliverable will be determined and documented. Such specifications include coding language, database solutions, class hierarchies, and UML diagrams. We will write the code during the Development phase. During the Testing phase, the deliverable will be tested and revised as needed until ready for integration with other modules. The project will be made accessible for future developers during the Documentation phase, when writing work will take place to record the project's implementation and usage.

4.4 Project Budget

No funding is needed for this project, as detailed in Appendix E. Dell has provided the codebase and virtual machines to work on the project with. We have no need to integrate proprietary paid software. Therefore, all expenses associated with this project have already been covered by Dell.

5.0 CONCLUSION

Software testing is a complicated process and requires a large amount of human oversight, therefore effective automation is highly valuable for integration. Our set of tools will form an automated system to reduce the cost and effort involved in Dell's testing and development. Furthermore, this project will reduce the expense of fixing bugs late in the product life cycle.

Our software-specific tool will be written in Python, and will utilize tools such as covr, MySQL, and R language scripts to run, analyze, and maintain a set of tests written in the R programming language. The four modules associated with our design have ultimately been developed by applying relevant prior research and identifying techniques for reducing technical risk. In addition, our team has administered individual task assignments that adhere to the strengths of each member. Moving forward, our team plans to continue working on the testing tool as it relates to changes in the ongoing Dell EMC Q project. Based on the work we have completed with our preliminary design, prior art research, and risk reduction experiments, we feel prepared for the upcoming semester as we focus on executing the technical components of this project.

With successful implementation of a bug identifying software program, developers will be able to spend less time running tests irrelevant to their code changes, and more time solving bugs as opposed to diagnosing them. Ultimately, this project will not only increase the quality of software at Dell, but also reduce the product workload and better the Dell developer experience.

REFERENCES

[1] Dell EMC’s Products, https://marketplace.vmware.com/vsx/company/emc-corporation.

[2] K. Aoyama, Z. Campbell, N. Gaur, and H. Khatri, “Software Bug Triaging,” UT ECE Capstone Project, Spring 2019. [Online]. Available: https://www.ece.utexas.edu/capstone/projects/software-bug-triaging.

[3] 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.htm.

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

[5] A. Pasala, Y. L. H. Lew Yaw Fung, F. Akladios, G. Appala Raju and R. P. Gorthi, "Selection of Regression Test Suite to Validate Software Applications upon Deployment of Upgrades," *19th Australian Conference on Software Engineering (aswec 2008)*, Perth, WA, 2008, pp. 130-138.

[6] Huang et al., “Method and apparatus for regression testing selection for a framework-based application,” U.S. Patent 8 793 656, B2. Jul. 29, 2014.

[7] E. Givoni, N. Ravitz, T. Q. Nguyen, T. Nguyen, “Automated software testing and validation system,” U.S. Patent 8 347 267, B2. Jan. 1, 2013.

[8] D. Episkopos, J. Li, H. Yee, D. Weiss, “Prioritize Code for Testing to Improve Code Coverage of Complex Software,” U.S. Patent 7 886 272, B1. Feb. 8, 2011.

[9] Seurat, https://github.com/satijalab/seurat.

APPENDIX A: RELEVANT STANDARDS

Our industry contact has not specified any specific standards, therefore we have adopted the standards laid out in Dr. Gligoric's Ph. D. dissertation [4]. This paper recommends four standards for regression test selection tools: efficiency, safety, precision, and performance. The first standard, efficiency, is met when selecting tests takes less time that the total running time of unselected tests. The next standard, safety, is met when an RTS tool can guarantee that all unselected tests are unaffected by the changes in the new code revision. The counterpart to this standard, precision, is met when an RTS tool can guarantee that all selected tests are affected by the changes in the new code revision. The final and most crucial standard is performance. To meet this standard, the time it takes an RTS tool to select tests, run those tests, and collect the results must be shorter than the runtime of the entire test suite.

APPENDIX B: WORK BREAKDOWN STRUCTURE

1.0 Project Video
    1.1 Creation of Slides for Project Video
    1.2 Filming/Editing for Project Video

2.0 Project Management
    2.1 Project Status Reports (Weekly)
    2.2 Project Review Meeting
    2.3 Oral Design Review
        2.3.1 Creation of Slides
        2.3.2 Rehearsals

3.0 Design Description Report
    3.1 Design Description Report Outline
    3.2 Design Description Report Draft
    3.3 Design Description Report Final

4.0 Overall Design Implementation
    4.1 Source-to-Test Module
        4.1.1 Architecture Design
        4.1.2 Development
        4.1.3 Testing
        4.1.4 Documentation
    4.2 Test-to-Source Module
        4.2.1 Architecture Design
        4.2.2 Development
        4.2.3 Testing
        4.2.4 Documentation
    4.3 Code Diffs-to-Test Module
        4.3.1 Architecture Design
        4.3.2 Development
        4.3.3 Testing
        4.3.4 Documentation
    4.4 Testing Harness
        4.4.1 Architecture Design
        4.4.2 Development
        4.4.3 Testing
        4.4.4 Documentation

5.0 Mid-Project Demo
- 5.1 Finalization of Code Presented
- 5.2 Creation of Presentation Materials
- 5.3 Rehearsal

6.0 Test and Evaluation Report
- 6.1 Test and Evaluation Report Outline
- 6.2 Test and Evaluation Report Draft
- 6.3 Test and Evaluation Report Final

7.0 System Demo
- 7.1 Finalization of Code Presented
- 7.2 Creation of Presentation Materials
- 7.3 Rehearsal

8.0 Design Showcase Poster
- 8.1 Poster Outline
- 8.2 Draft of Poster
- 8.3 Feedback from mentor/TA
- 8.4 Final Poster

9.0 Written Final Report
- 9.1 Written Final Report Outline
- 9.2 Written Final Report Draft
- 9.3 Written Final Report Final

APPENDIX C: LINEAR RESPONSIBILITY CHART

Linear Responsibility Chart Key : P : Primarily Responsible C : Contributing I : Information Only	**Canyon Evenson**	**Michael Flanders**	**Jennings Inge**	**Saarila Kenkare**	**John Koelling**	**Janvi Pandya**
1.0 Module 1 (Test to Source) (Canyon and John)						
1.1 Architecture Design	P	I	I	I	C	I
1.2 Implementation	C	I	I	I	P	I
1.3 Testing	P	I	I	I	C	I
1.4 Documentation	C	I	I	I	P	I
Module 2 (Regression Selection) (Saarila and Janvi)						
2.1 Architecture Design	I	I	I	P	I	C
2.2 Implementation	I	I	I	C	I	P
2.3 Testing	I	I	I	P	I	C
2.4 Documentation	I	I	I	C	I	P
Module 3 (Source to Test) (Michael and Jennings)						
3.1 Design	I	C	P	I	I	I
3.2 Implementation	I	P	C	I	I	I
3.3 Testing	I	C	P	I	I	I
3.4 Documentation	I	P	C	I	I	I
Module 4 (Harness) (All)						
4.1 Design	C	P	C	C	C	C
4.2 Implementation	C	C	C	P	C	C
4.3 Testing	P	C	C	C	C	C
4.4 Documentation	C	C	C	C	C	P
Written Reports						
5.1 Design Description Report	C	C	C	C	C	P
5.2 Mid-Project Demo	C	C	P	C	C	C
5.3 Test and Evaluation Report	C	C	C	C	P	C
5.4 System Demo	C	P	C	C	C	C
5.5 Design Showcase Poster	P	C	C	C	C	C
5.6 Written Final Report	C	C	C	P	C	C

APPENDIX D: GANTT CHART

	8-Dec		19-Jan	26-Jan	2-Feb	9-Feb	16-Feb	23-Feb	1-Mar	8-Mar		22-Mar	29-Mar	5-Apr	12-Apr	19-Apr	26-Apr	3-May	10-May
System Design Report	🟥																		
Project Video	🟥																		
Design Description Report							🟥	🟥											
Mid-Project Demo									🟥	🟥									
Test and Evaluation Report												🟥	🟥						
System Demo												🟥	🟥	🟥					
Design Showcase Poster																🟥	🟥		
Written Final Report																	🟥	🟥	
Test-to-source																			
Architecture Design			🟥																
Development				🟥	🟥	🟥													
Testing							🟥	🟥	🟥										
Documentation										🟥									
Diff-to-test																			
Architecture Design			🟥																
Development				🟥	🟥	🟥													
Testing							🟥	🟥	🟥										
Documentation										🟥									
Source-to-test																			
Architecture Design			🟥																
Development				🟥	🟥	🟥													
Testing							🟥	🟥	🟥										
Documentation										🟥									
Test Harness																			
Architecture Design												🟥							
Development													🟥						
Testing														🟥	🟥				
Documentation															🟥				
Comprehensive Tests																🟥	🟥	🟥	

APPENDIX E: BILL OF MATERIALS

Our project is purely software based, so our team has incurred no costs throughout the past semester. All work up to this point has been conducted on our personal machines as well as Dell's virtual machines that have been provided to us free of charge.