Data Verity Inc. Automated Testing Suite Final Report

June 18, 2019

Data Verity Team #1

Cameron Kuan, Grant Marsh, David Fresco, Eric Klatzco
Client: Brian Flannery

Table of Contents

Introduction 2
- Client Information 2
- Product Vision 2

Requirements 2
- Functional 2
- Non-functional 3

System Architecture 3

Technical Design 4
- Record 4
- Playback 5
- Comparison 5
- Database Tables 6

Quality Assurance 6
- Version Control 6
- Integration Testing 6
- Architecture 6
- Pair Programming 7
- Code Review 7
- User Acceptance Testing 7

Results 8
- Features Not Implemented 8
- Summary of Testing 8
- Results of Usability Testing 9
- Future Work 9
- Lessons Learned 9
  - PHP 9
  - JSON 9
  - Agile 10

Appendix 11

INTRODUCTION

Client Information

Data Verity, a Colorado-based company founded in 2006 by brothers Gordon and David Flammer, develops enterprise-level customer relationship management and business intelligence tools. They mainly work with consulting firms and financial institutions to make state of the art cloud-based solutions. The company has worked on projects such as full CRM enhancements, predictive analytics, and many more. The project "Automated Testing Suite" is focused on the idea of creating a way to automatically test existing web database applications. It is necessary to make it easy for a developer to validate their own changes before suggesting merges. This will allow developers to start making changes without fear of accidentally causing bugs.

Product Vision

The goal of this project is to create a system with which function executions can be recorded, then played back under the exact same conditions to compare the outputs. The results of the playback are compared to the recorded output to ensure the functionality of the code has been preserved. Due to the nature of the large web application base, there are many types of inputs and outputs relevant to each function. Function executions may rely on global variable states, user session information, or even database information stored in a local cache. This is, of course, in addition to the widely varied types and numbers of arguments that each function can potentially supply. The main challenges of this project are efficiently and effectively storing inputs and outputs as well as relevant memory states at the beginning and ends of the function execution. Addressing this problem means storing all the relevant data in a database in a way that maximizes both space efficiency and access speed.

REQUIREMENTS

Functional Requirements

Given individual credentials to the company's main server, the team will write PHP testing classes that will record and replay the arguments passed to tested methods. One method will be called at the beginning of a function that records arguments, a start time, and other state variables. Another method will be called when a function returns to record the return value, execution time, and other state variables. In addition to this functionality, the testing suite will also be able to record changes in pass-by-reference variables, and any errors thrown as part of the logic flow of the website. The team will also need to test the suite in order to ensure that appropriate exits occur for every function as well as other integrity tests to ensure that tests are viable. Finally, another script is needed that can replay tests that have been recorded. The purpose of this script is to distinguish which function outputs differ from the expected outputs stored in a database and to keep track of performance improvements/degradations. A stretch goal is to port the code to JavaScript in order to be able to use it on the client side.

Non-Functional Requirements

While the core of the usefulness for the system lies in the back-end software, in order to interpret returned data, there must be a cohesive front-end. The system needs to be easily interpretable by the user in order to quickly identify and rectify errors. The data also needs to be consistent and both store and return the necessary information. Lastly, the system needs to be built within Data Verity's codebase and be easily implementable into their pre-existing code.

In addition to having an easily usable and implementable system, the team was also tasked with minimizing memory and data abundance. In order to accomplish this, the team must have their functions run as quickly as possible and using as little resources as necessary, as to not interfere with preexisting workflow. The database that stores all necessary testing data also needs to be designed as efficiently as possible as to minimize the amount of total data storage it requires.

SYSTEM ARCHITECTURE

In Figure 1 below, the team outlined its initial design for the project. At first, there were going to be seven separate database tables consisting of inputs and outputs of normal functions, the database, and the cache along with an error table, all holding their own state variables. In the terminals, the team could also access the database information by directly using MySQL. In order to test the code, the team utilized the company's admin website to make sure tests were passing or failing.

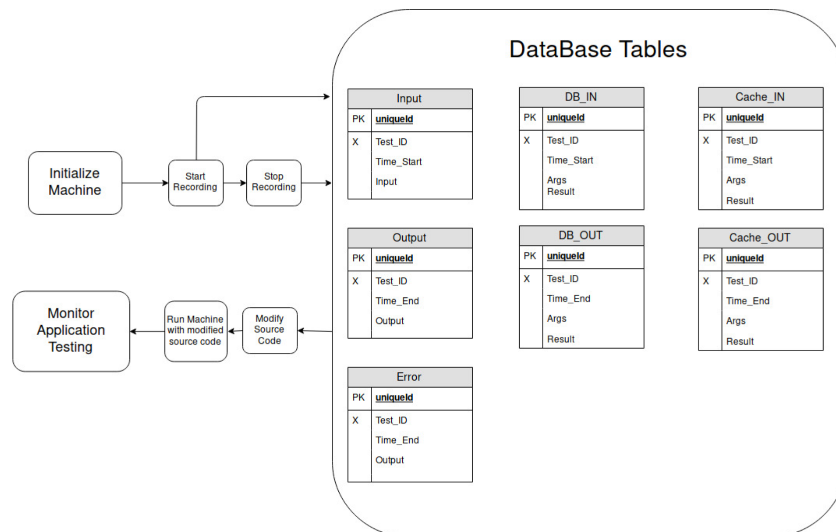

Figure 1: Initial Project Design

As the team began to understand the project more, the initial project design was deemed too complex. Instead of creating seven different tables that would all hold similar components of function execution, two database tables were created. These two tables, the recording and comparison tables, were the only necessary tables needed for the suite. The recording table holds most of the necessary elements needed when running the testing suite. The parts in the comparison table are necessary in order to specifically identify modifications made to the code. Figure 2 below is the final table design.

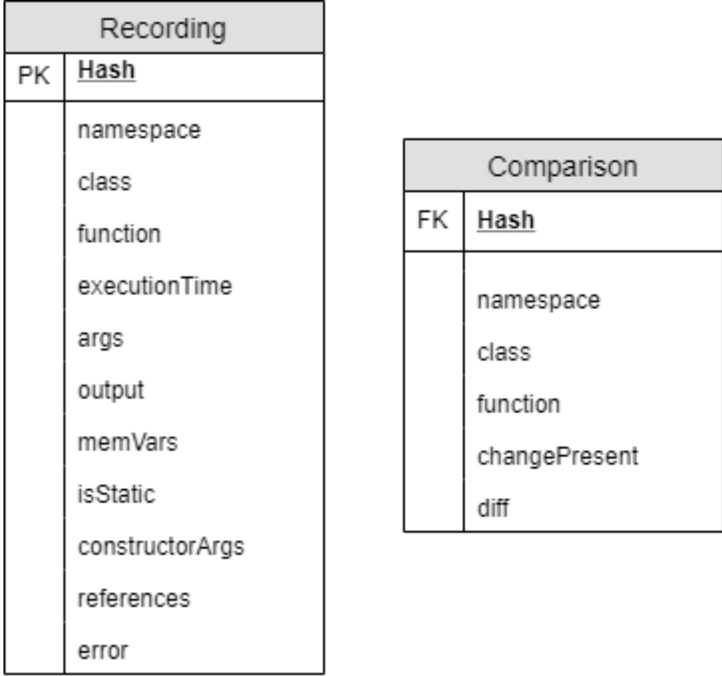

Figure 2: Final Table Design

TECHNICAL DESIGN

Recording

The recording aspect of the testing suits had to record arguments and state variables relevant to a function's execution and store them in a database. The initial solution involved tracking function executions from a static context. However, this soon posed a few big problems for the multithreaded workload the team would be testing. Because multiple functions can be executed at once, the suite would also require a set of active function executions. Upon termination of a function, the suite finds and remove the function from the set. This solution added a great deal of complexity and scaled very poorly as the number of functions in the air increased. Furthermore, a severe problem would be encountered when two instances of the same function executed at once. More than likely, the instances would collide and corrupt each other unless a timestamp is stored as a scope variable in the function and supplied at the return in order to distinguish one

execution from another. However, this could be unreliable in a system that potentially receives thousands of requests every minute.

While considering these obstacles, the team realized that scope variables that are local to a function would be required regardless. In order to mitigate the collision problem and do away with the confusion of which scope variables would be required at once, it was decided to instantiate a Record object in the scope of a function in the very first line and call its “finish” function the line before the return statement. The finish function records outputs and writes the finished test to the table. This means each function execution handles its own recording, and each instance of its execution is unique from the next. This solution eliminates the need of a big set to track active function executions because once the function returns, the relevant data that was already recorded, and its associated memory, is freed automatically by PHP garbage collection.

Playback

The goal of the playback function is to replay functions that utilize the testing functionality and run the results against those stored in the database. This was accomplished by accessing the database and determining if the table of recorded tests exists. Then, the function checks if the table is populated. Once it is determined that the table is both in existence and populated, the suite is then able to use the saved class name, function name, namespace, and JSON encoded arguments to re-instantiate the method to be tested. Re-creating the exact program state that the function was recorded under, however, was more difficult than simply running the method with a given set of arguments. For example, if a member function, *object* contains unique constructor arguments or references other member variables, the suite would be forced to record those separately in the record class in order to playback an identical test.

The playback function is able to access the information needed to find a function from a specific class and under a specific namespace using recorded information within the table. In addition to this information, playback is also able to access both member variables and constructor arguments. These are a few things that would fall outside of the scope of arguments and returned outputs, and re-create an almost identical instance of testing a function. Once the return value is recorded, or an exception is thrown, the suite is able to compare that value to the pre-recorded value, and determine if the functionality of a tested method has changed based on what is returned.

Comparison

The goal for this project was not only to compare return values for functions, but also note execution time, database outputs, pass-by-reference variables, and errors. Not only is this comparison aspect useful for checking to see if anything broke, but confirming that modifications to functions are working properly by providing expected and received outputs. Another feature is the comparison of execution times. Should a function still produce the right outputs after modification but take twice as long to execute, the modification was not as well written, as it impacts the efficiency of the program. In this way, the comparison features serve as a tool to assist in the refactoring and improvement of the code, not just its validation.

Database Tables

There are two tables that the testing suite uses to properly function. The recording data table stores all data needed to replay a test. This includes an identifying hash ID, plaintext class and function information, arguments, outputs, instance variable states, and errors. The recording data table is used as the base point for comparison. The comparison data table holds all information relating tests once they have been played back along with identifying information, such as the hash. It permits for easy reading and recording of passing and failing tests, along with the difference between the recorded and actual tests.

An important part of the table and suite design is the hash ID, which is composed of the namespace, function and class name, along with the function arguments. The hash is made using the 'md5' algorithm. It allows for the suite and its tables to be both efficient, accurate, and precise. As any given function has the three naming conventions, queries do not have to be based around a certain name or random key, and thanks to the use of the function arguments, multiple tests can be made on just one function. Thus, pulling a row from the database only requires a call to a hash-making method and a company-made querying method. Finally, as the ID is the primary key in the recording table and foreign key in the comparison table, there are no storage-wasting duplicates.

QUALITY ASSURANCE

Version Control

For version control, the team was given individual checkouts to access the company's database in order to separately work on the components needed for the project. Data Verity uses Subversion, a version control program nearly identical to Git. With separate checkouts, the team was able to transfer the latest and best working files amongst each other while also being able to develop smaller sections of the project.

Integration Testing

The created code had to integrate well with the company's existing codebase and database. Thus, the team required a way to be sure that all bits properly functioned when placed together. Due to the nature of this project, there was little need for unit testing, but a heavy emphasis on integration testing. Since the basic structure or architecture of the project consisted of four main components: record, playback, comparison, and database, unit testing simply consisted of echo statements or test queries. Integration testing was needed so that the team was able to find any bugs within the interactions between the functioning classes in developed code.

Architecture

Architecture played an important role in the team's plan as the software needed to run efficiently. Almost everything the team designed was run through project manager Brian Flannery and his

colleague, David Flammer. Seeing as how the team accessed the company codebase to only add in a testing application, it was of utmost importance for the project manager to approve the team's ideas for how the suite should be built. Everything was designed with the Open-Closed Principle in mind, with the exception of a table-making class. The table-making class should remain the same and the team did not see the need to have a great deal of parent/child classes that would only build a part of a table. Instead, the focus was more on have the functionality and making sure that Data Verity could easily expand upon the code if need be.

Pair Programming

With the practices learned in Software Engineering (CSCI306), pair programming helped the team minimize the amount of duplicate work done and decrease the number of possible bugs in the code during product development. Furthermore, the team was able to work on different aspects of the project and tailor the work towards those who have more knowledge in the area while allowing for those with less experience to learn along the way. Additionally, by pair programming, the base stage of the application was developed quickly so that any new features would be established on a sound foundation.

Code Review

Code review goes hand-in-hand with pair programming because it allowed the team to refine their already working code and put all members in an equal understanding of the project's state. Since the team was working in separate checkouts, once the goals of a weekly sprint were complete, Subversion was used to make sure everyone had the same updated, working version of the code in their own directories so that copied and pasted code that did not work was avoided.

User Acceptance Testing

User Acceptance Testing was the most important part of the team's QA plan, as passing it essentially meant that they passed their finish line. Since the project was based on the server-side and solely ran by their client, it was integral that their solution met the following requirements:

1) Was usable by Data Verity.
2) Met required specifications.
3) Caused no trouble to use.
4) Behaved properly and worked how the team expected it to, i.e. bug-free.

The application should meet the given requirements before the team could even consider user acceptance testing, as this should just be fixing anything minor. Thanks to the Agile method, user acceptance testing was done on almost a daily basis by members of the Data Verity team through conference calls or office visits. Unfortunately, full user acceptance testing was never completed, which will be elaborated on in the 'Future Work' section.

RESULTS

Features Not Implemented

From the team's initial plan, the features not implemented would be the testing of database and cache inputs. When discussing with the project manager, the team was informed that the two would require a great deal of work and learning and that there was not enough time to complete this feature. Besides this, flawless PHPUnit use was not established. The testing suite did not permit for the session information to be altered after echoing a statement, which would break session management.

While the initial plan was to do comparison through PHPUnit, this idea was eventually scrapped for a few reasons. Firstly, the existing infrastructure at the time the project was being worked on was not able to give PHPUnit a database connection. Since PHPUnit is usually initiated through the command line, its runtime contains none of the session or user information that is required when functions are called through the website's user interface. This means its runtime lacks the authentication needed to connect to the database. Another issue plaguing the PHPUnit interface is the fact that its strict, no-warning policy prohibits testing pass-by-reference variables. Instead, invoking the functions through PHP Reflection would mean that pass-by-reference variables are supplied as constants. This method would not crash the program, but would still throw a warning, which PHPUnit treats as an error.

Summary of Testing

When testing the automated testing suite, the team had two options; they could have used either Data Verity's admin website or the command line. Since the project was in PHP, a terminal based language, there were no actual debugging tools making it difficult to search for bugs if the tests failed in any fashion. So, by default, the team would write echo statements inside the code in order to make sure certain lines before and after said echo statement executed the way that was intended. By using the echo statements, using the admin website was the usual way of checking the results of executing the code. The other main way of testing the code was to directly use the command line within the terminal. This would be done by either directly printing the output or using MySQL in order to check if the database was properly updating. Regardless of which method of testing was used, if an unexpected error was detected, the team had access to the error log and the admin log to specifically find what went wrong within the code. This was the lone method of searching for a specific error within the code.

Results of Usability Testing

After completing the project, it was shown that the testing suite worked with both sample code that the team created, along with 'apps' already in use by Data Verity Inc. One such example is Parser.cls.php, which is able to break up any given message, including comments and newlines. The suite properly reads all namespace, class, and function information while also taking in arguments and the appropriate outputs. Minor changes to the Parser code also proved that the

suite was able to detect when playback outputs failed to align with the previously recorded outputs.

Future Work

In terms of future work, it would start at the features not implemented. While the suite does work as expected, it is not functional for changes to the database and cache. In terms of User Acceptance Testing, while the team was constantly checking and confirming with the project manager and his partner, they did not have time to implement their hooks into a large amount of the code base, which would be the end goal of the project's implementation. Another goal for future work would be porting the whole project to JavaScript, as mentioned previously as a stretch goal, which would allow for client-side testing.

LESSONS LEARNED

PHP

As the team was very new to PHP, solving basic problems with the web programming language proved quite difficult. At times, the issues arose simply due to lack of knowledge on the proper syntax to accomplish simple tasks. This problem was sometimes exacerbated by the confusing notation used in the online PHP documentation. Once the team became accustomed to the strange and beautiful nature of the language, productivity picked up very quickly thanks to PHP's large number of reflection-related features, e.g. special constant keywords that will return the name of the function the keyword is used in. Another big stumbling block involved the use of the "$" character, which is used to denote a variable, and the replacement of the dot operator with the "->" arrow symbol. The dollar sign character is only used in front of scope variables, not member variables of an object. This caused a long bout of confusion when trying to access member variables, mainly in situations like when trying to access properties of a Record like "$record->$args" instead of the proper syntax, "$record->args". These syntax quirks found in the unholy amalgamation of bash, JavaScript, and C++ that is PHP combined with debugging methods limited to what is provided by Vim and an error log resulted in a very slow learning process.

JSON

JSON, or JavaScript Object Notation, was a very useful tool in the project. JSON allowed the team to encode any array of primitive data types into a format that could be perfectly decoded when needed. This format was used to take arguments, outputs, member variables, and global/session information and condense them into a heap of text that could easily be stored. It removed the worry of having a table with an extreme number of columns just to accommodate any amount of inputs (for example, creating fifty argument columns just in case a function had fifty arguments). Instead, all arguments were encoded through JSON as an array, becoming one blob of text which could be decoded later once pulled out of the database.

Agile

Even after learning the Agile principle in CSCI 306, the principle was not well understood by the team. However, after the duration of this project using the Agile method, it has proven to be quite useful. By participating in daily stand-ups and weekly sprint planning, the team was able to recognize what the week's goals were and accomplish these goals very efficiently with special help from a specific Agile principle: The War Room principle. This principle applies when a development team is in the same room working as a cohesive unit in order to facilitate communication, problem-solving, minimize errors, and give updates to the client. Working together at the same time allows for individual team members to continuously check and update each other's code to ensure few bugs are created as well as writing clean code that is not too, in the words of Prof. Mark Baldwin, "smelly." The War Room principle was used throughout the duration of the project as the team always worked together in the same room whether in the Alamode lab or at the Data Verity office.

APPENDIX

The application has two needed lines of code to be injected into any given function. The first is the general hook to be placed right below a function declaration:

```
$rec = new Record(func_get_args(), _METHOD_, isset($this) ? $this : NULL);
```

This hook requires no function-specific information to be added, as it just pulls any arguments, gets the function path, and determines if it is static or not.

Before any *return* or *throw* statements, the following hook is needed:

```
$rec -> finish(array(return_args), array(reference_vars), “error_msg”);
```

User modification is required for this hook. The application expects some sort of return argument, be it an array or a simple input (like an int or string), but will also accept a null. The reference variable argument is null by default and is only needed when the user wishes to also keep track of any variables that were passed by reference. Finally, the error argument should be used before a *throw* statement to take in whatever the error message will be. It too is set to null by default.

If the function being tested belongs to a class that requires constructor arguments for instantiation, setConstructor function must be called in the first line of the constructor:

```
Record::setConstructor(func_get_args(), $this);
```

This allows the test database to store a sample constructor that will be able to properly instantiate an object during playback. This line is required regardless of whether or not the function being tested is static.

To have the application record tests, navigate to the Test App folder, which is within the ‘App Management’ tab of the Admin Manager. There can be found the commands to enable and disable the testing mode, along with a command to run the test playback, which will output passes and failures.

All information can be found in the tables ‘test_recordingdata’ and ‘test_comparisondata.’ The typical table prefix must also be used. In case the tables are dropped, they may be recreated via the command for Test App under the ‘Update System Tables’ tab.