CUT: automatic unit testing in the cloud

CUT (Cloud Unit Testing) is a tool that allows you to automatically execute unit tests in distributed execution environments. CUT will handle everything by allocating computation resources and granting access to virtual machines. Developers won’t be required to change any unit test code that has already been written and will be able to control all aspects of test execution. In addition to all this CUT will also provide resources that will generate analytics and give you insight into the unit tests that have been executed.

CUT shares many similarities with JWalk. Both attempts to automate the process of unit testing through software that makes runtime decisions to effectively test code. JWalk generates the tests and runs them itself alleviating the programmer of the effort involved in designing test cases. This does take control away from the programmer but will help speed the testing process. It also generates recommendations on future tests that may return informative results. In a very similar way CUT uses cloud computing to automatically allocate computation resources and automatically generate analytics and suggestions. Both CUT and JWalk use software to automate the testing process and provide feedback and recommendations to the programmer.

Advances in Unit Testing: Theory and Practice

Parameterized Unit Testing (PUT) expands on previous industry practises which were based on traditional unit testing without parameters. A PUT takes parameters, calls the code to be tested then states the results of the test. PUT also allows for the separation of Black Box testing and White Box testing. Allowing the use of parameters with traditional closed unit tests will assist in generating specific inputs to optimize and efficiently create results.

JUnit is very similar to the PUT testing method as it also takes parameters, calls the code being tested and returns the results of the test. Both PUT and JUnit do not try to automate the process in any way and give full control to the programmer. JUnit can be seen as an implementation of the PUT method.

Unit testing performance with stochastic performance logic

The article specifies a formal language to be used (known as stochastic performance logic) to break the constraints of unit testing when used in performance testing. Unit testing is the testing of small modules and may overlook bugs on a larger scale like integration testing. When testing our calculator program, we noticed the shortfalls as well, and we noticed where a formal specification like SPL could be useful. One clear issue with unit testing is the test implementation and design, an new tester would have difficulties setting up these module tests in a way to encompass the design of the tests.

Fortunately, in our case we were able to work around this issue in unit testing by having a strong understanding of test methodologies and in what order to implement them. Though we are not seasoned, we have a strong enough understanding of test methodologies and implementations where this wouldn’t be a large issue. Yet this article points out a very relevant issue ran into in many unit tests, and a strong work around to the issue.

Pythia reloaded: an intelligent unit testing-based code grader for education

This article outlines the usage of unit testing alongside automated competition software graders to create comprehensive code grader. This code grader is an intelligent unit-based code grader, that gives responses to input in a more abstract format only done successfully through intelligent automation; much akin to the intelligent and dynamic unit testing tool JWalk. JWalk is one of the two tools utilized in the testing of our calculator program. It uses a dynamic method, learning the test cases prioritized by the tester and then automatically creating its own test cases in an exhaustive method. Working at a much faster pace then its JUnit counterpart, it becomes a powerful tool by coupling its unit testing basis with an intelligent.

Through this article, we first see evidence that coupling of unit testing and intelligent tools is possible, and very powerful. Many of the downfalls spoken on by pythia of unit testing and why they are not complete are also used as assertions of JWalk’s strength over JUnit. The simple nature of unit testing, though useful in some measure, is useful in determining how to encourage a student passed on their input, just as JUnit is useless in helping to implement test methodologies and automating the unit testing process. Here JWalk thrives by coupling its smart and dynamic intelligent tool with the flexible unit test to another powerful tool, and how this article directly shows the technology being utilized in our tests.

Relational Symbolic Execution of SQL code for Unit Testing of Database Programs

In the article, a plan was devised to undertake the testing of programming languages that communicating to databases, where on simple unit testing framework was considered useless. An example is running unit tests in PHP, specifically in portions where SQL is written. The SQL is not possible to traverse through for the Unit testing tool as it is designed in a structured format with infinite permutations resulting in error. In quick summary, a relational symbolic code was written for dealing with these situations, by replacing the SQL with a symbolic language that was traversable by the Unit testing tools. Again, relations can be built from the article to our own JWalk testing tool, where a specification of its own is drawn up.

JWalk is tasked with the difficult task of dynamically inferring the evolving specification, suggesting test-cases that the programmer usually forgets. These specifications are derived in a similar symbolic manner to that of Relational symbolic form used with SQL. These test paradigms cannot obviously be compromised at execution as they’re dynamic. Hence, an abstract language must be derived from where a more specified translation can be used for test case referral. Again, references can be drawn from the literature reviews and the real application of unit testing, showing these articles are more then just theory and used in daily life throughout many units testing tools.

Case Study Introduction

A group of us had decided on testing a calculator function written in java to see if it held viability in a production standard. The program itself was broken down into three separate functions and could be best tested from our evaluation using unit testing. The question had then become, what unit testing tool was the best fit for the endeavour? Many different unit testing tools can be found out in the world but differentiating between them can be the difficult part. For our team, we decided to run the test using two separate tools - that even though they were both unit testing tools, were as far away form each other as possible. This made the most sense as we could then use two separate tools, both could have varying degrees of success but the combination of which would really help properly affirm whether our calculator program was ready or not. As well, using two very different tools would be a great learning experience from which we could see which of these two tools was better if any was at all. The choices then were narrowed down to the two most different testing tools in unit testing possible; JWalk and JUnit.

The latter allowing for manual unit testing, where the tester manually writes his own test cases resulting in him formulating his own test suits and methodologies to implement. Here we assume the tester holds a high degree of experience and what he wants implemented is perfect, that his test cases will result in a large percentage of the errors being caught. To mimic this perfect test case, we needed to act as perfect testers of our own; thus, a strong test case needed to be implemented. We will speak on this implementation later.

The other tool to be used was JWalk, its name coming from the leisure activity of walking versus running. The name choice though strange at first glance, makes a lot more sense when understanding the tool. JWalk is described as a testing tool that supports the “lazy systematic unit testing” [1], testing paradigm. In other words, this is a test case for people who may not be so inclined to write their own test cases out manually. JWalk is very different from JUnit, as well as most testing tools. The tool gains understanding for what tests the tester wants to create by the few test cases the tester must manually write, and then is able to produce its own test cases dynamically based on what the tester is trying to weed out. These test cases are generated exhaustively and used to cover the test classes state space adequately. As well is it able to provide test classes as suggestion in case they’re forgotten by the tester. In addition, the tool works at a much faster pace then its counterpart. From an early inspection, it seems obvious that JWalk is the front runner as the stronger unit testing tool.

Setup

The setup process involved a course of action on what and how to implement the different test methodologies. The three function behave vary differently, and we will approach there testing in an incremental format, starting with values being inputted into the first function, and then tests on the second alongside the first before finally the implementing test cases for the third function which is integrated with the second and first function. An overall combined approach was agreed upon by the group, since there was not an issue of different input combinations from various points of entry, cause and effect graphing and its derived test cases become a nonfactor. Instead, we jump into black box design with boundary value analysis, before moving into equivalence class partitions and then onto error guessing.

White box methodologies are not utilized, because of the basic simplicity and size of the calculator program. The decision and statement coverage serve no use as all the permutation of different paths, and all statements have been executed in the previous test cases. Since the testing process is long and drawn out, the test cases for the first module will be shown as the second test for boundary value and so forth on three other two modules are merely extensions of the tests on the first. Results for all three will be elaborated on in later portions of the writing.

Again, the boundary value analysis considers that we are storing values in integers in the calculator function. Thus, it creates a minimum and maximum test case inside the boundary (min: -2 147 483 648, max: 2 147 483 647) and another pair outside (min: -2 147 483 649, max: 2 147 483 648). These complete our test cases of the first module in the boundary value analysis test. We move to equivalence partitions, beginning with valid equivalence classes; firstly, of operations resulting in 0, with the four different operators, + - / \* being placed into there own sub classes.

Class partitions resulting in a positive and negative answer placed in their own classes, with subgroups based on operators. Within these groups even smaller were created for examples where both operands are negative, both are positive, and one is either or. Once the cases like these were completed, some of the left-over valid class partitions included dividing by 0, this class extends for any number dividing by 0 including 0 itself and is deemed as valid because it follows all valid inputs. Invalid class including using operators twice like 2 ++ 3 or leaving operators hanging like 2 -.

Decimal placed values as inputs were also considered part of invalid equivalence classes, though decimal places from results where the input did not have them were placed in there own class as well. Finally, error guessing techniques are utilized where, the input I decided to go with was that of the same equation but with differing inputs in terms of spaces; and example would be 2+ 3, and 2 + 3. Thus, concludes our test cases in a combined approach, as the white box methodologies are already accounted for by the large range in test inputs of the black box tests methodologies.

Conclusion:

In conclusion unit testing is a level of software testing where individual units and or components of a software are tested. It allows for early bug detection, refactoring, strong code design and teamwork. Our literature review consisted of ten articles compiled from various online scholarly resources based on the topic of Unit Testing. The unit test overall worked to a complete success, bugs were found and can easily be dealt with. The best part is having the knowledge of where exactly the issue is coming from, in Unit testing, searching for the problem is not a cause for concern in comparison to other testing frameworks. Implementing the combined searching technique also worked in our favor, we were to ensure all statements and decisions were covered, a variety of different inputs in different classes as well. Overall, the tests were implemented well, through incremental testing, we could see both the flexibility of unit testing and the integration testing created by stacking the separate modules on top of one another. The test cases were thorough and were able to cover so many classes, that decision and statement coverage had no real reason to be there.

Overall, the test cases were deemed a complete success. More test cases passed then failed, finding many more errors then issueless runs. Although many were easy to find some others were still surprising; examples of these can be shown in the boundary value analysis. Those test cases were predictable, the two test cases inside the integer cap boundary passed while the two that were outside failed. The decimal value input failed as the inputs and outputs were being held as integers and trying to place the decimal typed values would throw an error. All the invalid input classes failed, including the character letters and the multiple operators.

This compared to the much more surprising response during the execution of the error guessing inputs that showed equations without spaces between the operands and the operators, could not be used while having a space on the right operand or both, allowed the statement to be read just fine. The incremental test meant, that many more testcases had to be derived for the other features. Though, these features seemed daunting, the size of the other two function made this an easy endeavour. One software bug was found in the usage of precedence through the bracket () functions. Both JWalk and JUnit were effective, responding with accurate totals including 65% passed test cases, or 65% runs that failed. JUnit took a lot longer to run the same number of tests, though our decision on the better tool went to JUnit. Though, it would seem cleat that JWalk is better tool, context is one of the key situations that made JWalk unfavourable. The length at which it took to setup and get the JWalk running was a large amount of time. There was also a large learning curve, in comparison to the simple method in unit testing where we simply set up the unit test suite like sperate class and were on our way. The UI itself wasn’t very clean with JWalk either, very cluttered or this as well did not make it easy to pick up the newtool. Thus, overall, we all believe the JUnit was the stronger testing tool.

Citations:

1. [Lazy Systematic] AJH Simons, “The JWalk Download Centre,” *University of Sheffield Department of Computer Science*, 2017. [Online]. Available: <https://staffwww.dcs.shef.ac.uk/people/A.Simons/jwalk/download.html>[Accessed: 02-Apr-2020].
2. [JUnit da Best] T. Xie, N. Tillmann, and P. Lakshman, “Advances in Unit Testing: Theory and Practice,” *OverOps*, 20-Nov-2013. [Online]. Available: <https://blog.overops.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/>[Accessed: 02-Apr-2020].
3. [JUnit vs Walk] AJH Simons, “The JWalk Home Page,” *University of Sheffield Department of Computer Science*, 2013. [Online]. Available: <http://staffwww.dcs.shef.ac.uk/people/A.Simons/jwalk/>[Accessed: 02-Apr-2020].