International Journal of Engineering & Technology, *7 (3.34) (2018) 580-584*

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Construction of a Regression Test Automation System

Inhwa Choi[1], Wonshik Na[2*]

[1]Department of Software Engineering,Tomatosystem Co.Ltd Seoul, Republic of Korea
*[*2]Department of Beauty & Health, Namseoul University Cheonan, Republic of Korea*
[*]Corresponding Author E-mail: winner@nsu.ac.kr

Abstract

Background/Objectives: As testing plays an important role in software quality assurance, many studies are now being carried out in areas such as test method design, test case generation, test case management, GUI test automation, and integrated test automation in order to create tests that are more efficient as well as cost efficient.
Methods/Statistical analysis: This paper proposes a regression test automation system that can dynamically generate test cases in the regression test stage and automatically execute the generated test scenarios
Findings: The Mocha framework is extended to build an automation framework, and test algorithms and test algorithms are classified to extend and combine various test cases.
Improvements/Applications: Applying the proposed system to a UI development tool test saw more than twice the test cases being automatically generated, which led to a discovery of 37% more key defects.

Keywords: *regression test, test automation, integration test, testing, data-driven*

1. Introduction

The importance of software quality is becoming increasingly important as the scope of software and their complexity increases as a whole in society. As testing plays an important role in software quality assurance, many studies are now being carried out in areas such as test method design, test case generation, test case management, GUI test automation, and integrated test automation in order to create tests that are more efficient as well as cost efficient. However, the reality is that because test phase takes up a lot of time and cost that it accounts for 35-40% of the software development period, many software development projects are not able to invest the appropriate amount of time and resources in testing[1]. If the test cases are not managed and run properly during the development phase, the regression testing that is done after the development phase will be even costlier and take a longer time[2]. In case it is not possible to run a regression test, the time and costs are doubled because a full test must be carried out every time a correction is made, leading to the distribution of software that cannot be guaranteed to be of high quality[3].
To address this issue, this paper proposes a regression test automation system that can oversee the testing process from the development to the regression test phases at minimum cost. The proposed system incorporates test automation, improved reusability, and a risk-based regression testing to reduce the time and costs of testing. A test automation framework was developed by extending Mocha to automate tests that allow regression testing. A data-driven method was used to improve reusability, and the test cases generated by the data-driven method were combined to be used again. In order to combine the previous test cases, a structure was adopted that separates all test algorithms, data, and scenarios. In addition, a risk-based regression test was implemented to make regression testing more effective. This proved to be highly helpful in reducing testing time and costs, as it made it possible to dynamically combine test cases created beforehand when creating additional test cases for risk situations that occur after the development phase.
The composition of this paper is as follows. Chapter 2 introduces software testing techniques and regression testing based on the test case selection strategy. Chapter 3 introduces the proposed regression test automation system. Chapter 4 summarizes the results of applying the regression test automation system, followed by the conclusion in Chapter 5.

2. Related Studies

2.1. Software Test Techniques Based on the Test Case Selection Strategy

There are a variety of test techniques that can be used in the software development phase. The following provides an overview of the technique categories depending on the strategy of selecting test cases from test suites to test software [4].

2.1.1. Exhaustive Testing

The method of testing all test cases is called exhaustive testing. However, it is almost impossible to test all test cases so a selection is made for testing in general.

2.1.2. Operational Testing

Operational profile refers to the frequency with which users use software. Operational testing, also called operational profile-based testing, is the method of selecting test cases by considering operational profiles only.

2.1.3. Debug Testing

Debug testing is the method of selecting all test cases that are likely to have or cause defects. Trying to find as many defects as possible is common to most test methods.

2.1.4. Random Testing

The method of selecting test cases according to the probability distribution, and testing until they meet the specified end criterion is called random testing, or testing without a sub-domain.

2.2. Regression Testing

Regression testing is a type of software test that focuses on selective retesting in various versions of the software system [5]. The IEEE [6] defines regression testing as follows.
"Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements"
That is, regression testing is performed for the purpose of re-verifying whether a modification or addition to a software module has caused a defect or an error by affecting existing software. Figure 1 shows regression testing in the life cycle of software development and maintenance.

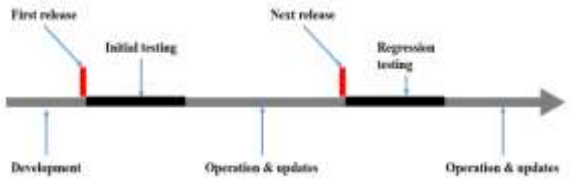

Figure 1: Regression testing in software development process

Figure 2: Regression testing and test case

Figure 2 shows regression testing and test case selection. As seen in the figure, testing is carried out in the software maintenance phase or in the evolution process when a change is made or if a new function is added to the program to be released.
Here, the test suite for the changed or added element, separate from the test suite used to test the original program P,must be reconsidered. Therefore, in order to test program P', test suit T' must be made from a selection of test suit T, which tested the existing functions. Test suit T'', a new set of test cases that test the new function, must also be created and used with test suit T' to test program P'. Here, the selection of the test suite T' might cause an issue in terms of the cost, duration and effort required for the testing [7-9].

3. The Proposed Regression Test Automation System

The regression test automation system consists largely of a Client and a Server. The Client consists of an integrated development environment and a user interface screen. The integrated development environment provides an environment for writing test codes based on the test framework. The user interface provides a pathway for entering test data and commanding the execution of the test. The server consists of a Test Manager, a Report Manager, a Data Repository, and a Code Repository. The Test Manager consists of a Test Case Manager that manages the combinations of risk-based test cases; a Test Case Generator that dynamically generates test cases to be run; and a Testing Manager that automatically performs regression testing. The Report Manager consists of a Result Manager that manages the results of regression testing, and a Report Generator that generates various reports. There are two repositories: the DBMS that uses MySQL to store test data, test cases, and test results, and the GIT Server that stores test costs and business logic. Figure 3 shows the overall configuration of the proposed regression test automation system.

Figure 3: The overall configuration of the system

3.1. Test Manager

The Test Manager, which consists of a Test Case Generator, a Combination Manager, and a Testing Manager, plays the most important role in the regression test automation system proposed in this paper

3.1.1. Test Case Generator

The Test Case Generator is a module that dynamically generates test cases to be automatically run. Most test automation frameworks automatically perform all of the implemented code and provide test results. However, if the number of test cases increases, or if there are test cases that do not need to be run anymore, the issue of time and costs might arise. Thus, this paper proposes a method of test automation where the user can determine the testing range using the Client GUI, and the Server Test Manager's Test Case Generator can dynamically generate the test codes and test data as needed by the set range. Figure 4 shows the test components generated by the Test Case Generator. The test item, test scenario, and test case set are the test execution units that can be determined by the user.

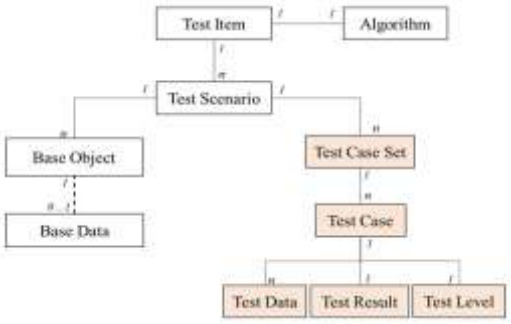

Figure 4: The test components

One test item consists of one test algorithm and one or more test scenarios. A test scenario consists of one or more test objects and test case sets, respectively. A set of test cases consists of one or more test cases, which in turn consists of test data, test results, and

test levels. The separation of test algorithms and test cases enables data-driven test case generation. The separation of test scenarios and test cases enables reuse of premade test cases when adding risk-based regression test cases.
The procedure for creating a test case in the Test Case Generator is as follows.
Step 1: The user imports the selected test range through the GUI.
Step 2: Test case sets corresponding to the selected range in the DBMS are imported.
Step 3: Test data is imported from the DBMS using the test data of the test case set.
Step 4: A test code list is created using the test code information of the test case set, to be forwarded to the Test Manager**.**

3.1.2. Test Case Manager

Regression testing must be repeated to verify that code modifications did not affect past code. It can be difficult to detect defects by repeatedly performing the same test with the same test case each time. Test cases must be added by predicting defects that may occur from code modification, and repetitive testing must be carried out by selecting the range that might be affected. This paper presents risk-based testing as an effective method of running regressive testing, and presents a Test Case Manager based on the following considerations.
Step 1: Prioritize test cases based on the frequency and severity of defects, and the impact they have on customers.
Step 2: The test cost is managed separately from the business logic.
Step 3: If the expected risk is detected, the test data and Test cost should be separated so that the test case can be extended into various forms.
Step 4: If the expected risk is detected, each test cost must be implemented separately so that premade test cases can be combined to create a risk-based test case.
In consideration of the above, all test cases are configured so that upon creation, priority can be automatically or manually assigned according to the rules. The test code is written separately so that codes can be added, modified, and deleted can be done without affecting the business logic. In the GIT Server repository shown inFigure 3, it can be seen that the test cost and business logic are put in separate spaces. Each test case consists of one test code and a number of test data, making it easy to add, modify, delete, and combine data. Below is a simple example that shows an independent test cases and a combination of test cases. Figure 5 shows the result of performing a test case to test the results of a simple API test (right side of the figure).

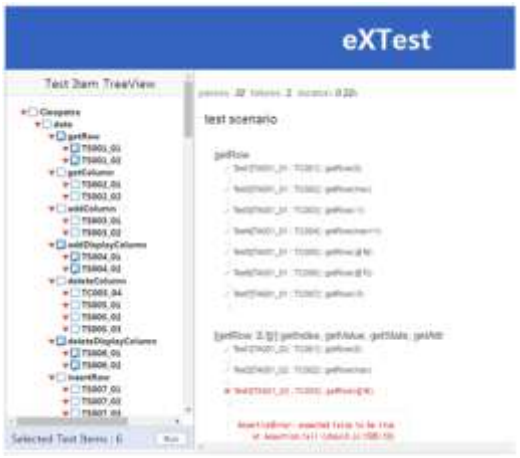

Figure 5: The result of performing a test

The API 'getRow' introduced in the example is an API used in tabular UI components composed of columns and rows. If you specify the index as a parameter of 'getRow', it returns the row at the specified index. Testing for getRow is done in two ways. One is a simple test for the getRow function and the other is a combination of getRow and another API.Since the first simple test is intended to perform independently without combining with other APIs, the various index values become test data, and the test algorithm is conceived from the modified test data applied to getRow. Figure 6 corresponds to the test algorithm of getRow. The description of the test code is covered in detail in Section 3.3.The combination test of the second getRow is a combination test case newly added after the completion of the first test. It is one of the test cases made after a risk associated with value was detected after running the first test. The getRow combination test is a test case created by reusing the test case that successfully returned the row object in the first getRow test and combining it with another API test case.The test results are shown as '[getRow combination]' in the bottom right of the figure. It can be seen that there was no failure in the first getRow test case, but that there was a failure in the second getRow combination. The defect that was detected was a high risk priority that could stop the program from running.

3.1.3. Testing Manager

The Testing Manager is responsible for executing the test cost list forwarded by the test case generator. It imports the test cost to be performed from the GIT server, and substitutes the test data in the test case set to execute the test program. After the execution of the test program, the execution result and the performance information are forwarded to the Report Manager's Result Manager so that the test execution results can be accumulated.

3.2. Report Manager

In order to understand the effect of code modification through regression testing, the history of text execution needs to be managed and the cumulative results analyzed and provided to the user in a form that is easy to understand. It is the Report Manager's Result Manager and Report Generator that support this function. The Result Manager receives the performance information from the Testing Manager and stores the test results. The results are compared with the accumulated results and analyzed to provide users with useful information.

Attribute(fail)	Attribute combination (fail)	Attribute combination (fail)
All(0%)	All, TextAlign, VerticalAlign(0%)	All, VerticalAlign(20%)
AutoSkip(0%)	AutoSkip, EditMask, MaxLength(75%)	AutoSkip, MaxByteLength(50%)
BackgroundColor(0%)	BackgroundColor, RowColor(37.5%)	
EditMask(52.08%)	EditMask,ImeMode,InputMode(85.71%)	AutoSkip, DataType, EditMask, MaxLength(75%)
MaxLength(0%)	MaxLength,SpinButton,SpinRangeMax ,SpinRangeMin(25%)	SpinButton,SpinLoop(0%)
RowColor(0%)	RowColor, RowOverColor(50%)	

Figure 6: An example of a defect when a test case is combined

The comparative analysis is based on the history of defect occurrence, the risk priority, and the error type. The history of defect occurrence is analyzed according to the following criteria.
If a former defect has been modified
If a new defect occurred
If the occurrence and modification of defects is repeated
If the defect is detected no more than 5 times
The risk priority and the error type in which a defect occurs when a test case is combined are analyzed based on the following.
Defects by risk priority
Defects by error type
The analyzed results are generated in the form desired by the user by the Report Generator. Figure 6 is a simple example of when a defect occurs when a test case is combined.

3.3. Client

3.3.1. Integrated Development Environment

The Client provides a user interface that allows users to easily run tests and create test cases. There are two types of interfaces that are provided: one is an integrated development environment for test engineers who develop test codes, and the other is a graphical user interface that supports testing and creating test cases.
The test code is developed using Visual Code based on a test framework extension of Mocha

Figure 7: The test algorithm for the getRow

Figure 7 shows the test algorithm for the API getRowexemplified in Section 3.1.2. Since it is an extension of the Mocha framework, the entire code structure consists of before, after, beforeEach, afterEach, and it functions. This structure is automatically provided whenever an algorithm is developed. The before function is executed when the test for one Test Case Set introduced in Figure 4 starts, and the after function is executed after the test for one Test Case Set is finished.The beforeEach function is executed before the test case comprising the Test Case Set is executed, and the afterEach function is executed after the test case is executed.Each test case is executed in the function. The beforeEach, afterEach, and it functions are provided testCaseData as the common parameter, and this value is comprised of the test data received from the DBMS and the test result. The API getRow used as an example takes an integer index value such as 0,1, and 2 as a parameter. That said, it can be seen that line 33 in the figure above is written with getRow(testCaseData.P1). This is because the index values can be changed, and are dynamically fetched from the DBMS and reflected in the testCaseData.P.Checking the test results is also a dynamic fetching and comparing process, as the expected results can change according to the test data.This corresponds to line 35, where the code for testCaseData.P2 fetches the expected results and runs a comparison

3.3.2. Graphical User Interface

Engineers who write test algorithms can implement the test algorithm through the integrated development environment described above and run it to check the results immediately. However, because not all testers can develop and run programs through the integrated development environment, a Graphical User Interface that is easy to use for those responsible for performing the tests is provided. The system's Client provides three important GUIs. The first is a GUI that supports optional regression test automation, which is represented in the tree structure in the left of Figure 5. If this is not selected and the run button is pushed, the entire test is automatically performed and the result is displayed on the right side. If a specific item is selected, only the corresponding item is tested and the result is presented. The second important interface is that for entering test data. The system proposed in this paper uses a data-driven method that allows even people without knowledge of programming language to create test cases as long as they know the test data. Therefore, a GUI that supports selecting test items and inputting data is provided. Last but not least, GUI is provided that allows users to combine test cases. Selecting a test item will show the relevant test items that can be combined. These can be selected for the TestCase Generator to combine, and the user is provided an input screen where they can input the expected results for the combination

4. Application Result

The regression testing system proposed in this paper aims to build a system that can perform automated tests, improve reusability, and execute risk-based regression testing to drive the costs and time needed for testing. The Mocha framework was extended to automate tests and results are saved in MySQL to enable regression testing. A data-driven method was used to improve reusability, so that one algorithm could be used to create multiple test cases. The test cases added during regression testing improve reusability and usability by making it possible to select and combine existing test cases. The proposed system was applied to a UI development support software developed by an SME. The tests were applied at the start of the tool development phase until the end, as well as the regression testing after the development was completed. The test was an integration test that excluded the GUI test. To gauge the accuracy, 126 APIs of the GUI development tool were analyzed.
The test cases consist of the test algorithm, target object, test data, and the expected result. The test algorithm can be used for all target objects. Target objects are applied with different levels of test cases according to their importance. The exception and boundary value tests were applied to those of high level importance; boundary value tests were applied to those of mid-level importance; and only the test data corresponding to the representative values were applied to those of low level importance. The results of application are as follows.
Basic algorithm: 126
Test data: 2968
Test cases applied to target objects of high level importance: 2968
Test cases applied to target objects of mid-level importance: 1654
Test cases applied to target objects of low level importance: 876
Test cases created by combining test cases: 2456
An integration test is a test that can be performed by combining several APIs and target objects. Therefore, the test algorithm and data created during the development phase could be used, and the expected results could be managed separately. Running the generated test cases detected 37% more defects than from the default test.

5. Conclusion

Testing is an important step to ensure the quality of software. However, it takes a lot of time and financial resources to manage and execute test cases from the start of the software development stage to the end[10,11]. If test cases are not managed during the development phase, even more time and resources will be required in the regression testing performed in the following phase[12,13]. This paper proposes a test automation system that aims to solve this issue by testing and managing tests at minimum cost, from the development stage to the regression testing stage. Applying the proposed test automation system to a UI development tool test of an SME, it was found that the system could generate an average of 29 test cases with one algorithm, as well as automatically create more than 2 times the number of combination test cases based on the default test case. Through combination test cases, 37% more defects were detected.

Acknowledgment

Funding for this paper was provided by Namseoul University.

References

[1] Architectures of Test Automation,- arcor.de http://www.kaner.com/pdfs/testarch.pdf
[2] Khaled M. Mustafa, Rafa E. Al-Qutaish, Mohammad I. Muhairat. (2009). Classification of Software Testing Tools Based on the Software Testing Methods. Computer and Electrical Engineering, 2009. ICCEE '09. Second International Conference on
[3] H Kaur, G Gupta. (2013).Comparative study of automated testing tools: selenium, quick test professional and testcomplete.Journal of Engineering Research and Applications 2013
[4] B. Beizer(1990).Software Testing Techniques, Van Nostrand Reinhold, New York, NY, USA
[5] Gregg Rothermel, Sebastian Elbaum, A.G. Malishevsky, P.Kallakuri, and X. Qui. (2004). On Test Suie Composition and Cost-Effective Regression Testing, ACM Transactions on Software Engineering and Methodology
[6] M. Balcer, W. Hasling, and T. Ostrand, Automatic generation of test scripts from formal test specifications. (1989).In Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification
[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. (1978). Hints on Test Data Selection, Help for the Practicing Programmer.Computer, Volume: 11, Issue:4, April 1978, 34-41
[8] T. Goradia. (1993). Dynamic impact analysis: A cost-effective technique to enforce error-progagation, In Proceedings of the ACM International Symposium on Software Testing and Analysis, 11-181
[9] Artzi, J Dolby, SH Jensen, A Moller. (2011). A Framework for Automated Testing of JavaScript Web Applications.Software Engineering. ICSE, 2011 33rd International Conference on
[10] A Kumar, S Saxena. (2015) Data Driven Testing Framework using Selenium WebDriver. International Journal of Computer Applications, 0975 – 8887
[11] P Laukkanen. (2006). Data-driven and keyword-driven test automation frameworks, HELSINKI UNIVERSITY OF TECHNOLOGY, 2006
[12] S Amaricai, R Constantinescu. (2014). Designing a Software Test Automation Framework.Informatica Economica, 2014
[13] Rohan R. Kachewar. (2011). K model for designing Data Driven Test Automation Frameworks and its Design Architecture “Snow Leopard”.International Journal of Computer Applications