UNIT TEST CODE GENERATION TOOL FOR LOWER LEVEL PROGRAMMING LANGUAGES

Kangana Mudiyanselage Parakrama Danesh Rasika Bandara

(179307J)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2020

UNIT TEST CODE GENERATION TOOL FOR LOWER LEVEL PROGRAMMING LANGUAGES

Kangana Mudiyanselage Parakrama Danesh Rasika Bandara

(179307J)

Thesis submitted in partial fulfillment of the requirements for the degree Master of Science in Computer Science specializing in Software Architecture

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2020

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis/dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

....

Kangana Mudiyanselage Parakrama Danesh Rasika Bandara Date

The above candidate has carried out research for the Masters thesis under my supervision.

....

Dr. Indika Perera Date

ABSTRACT

In the software development life cycle, there are a few, well-known, major phases and their sub-phases. Those are, namely, requirement engineering, feasibility study, design, development, testing, deployment and maintenance. Within those, the most likely sub-phase to be overlooked is the unit testing, in the testing phase. One of the main reasons for such negligence is that the cost it takes for unit testing. This cost is in the aspect of the effort, the amount of human resources to be put for unit testing. Most of the time, the project managers and the other responsible personnel, trade-off between carrying out unit testing and the cost it would take, and, either, neglect unit testing or carry out unit testing lightly.

In the case where carrying out unit testing lightly, it could be either writing trivial unit test code, or carrying out unit testing by debugging the code for various cases for functional units. In these cases, the code could not be tested enough at all. Or, there would not be any valid evidence to prove that a comprehensive unit testing has been carried out.

It is important to have a very good balance between carrying out comprehensive unit testing, keeping solid evidence of unit testing and saving the cost it would incur for unit testing. Considering all the aspects mentioned above, this research suggests a spreadsheet format as a unit test specification and offers a unit test code generator tool to generate unit test code based on the said unit testing specification. This research has considered using Microsoft Excel as the spreadsheet software to create the unit test specification, as it is widely used and popular. The unit test code is generated for the C++ programming language, which does not have that much of good unit testing frameworks. And, the unit test code is for the emerging unit test framework, Google Test.

The outcome of this research has been applied to five different industrial software system projects and six target functions of each of those projects, which ultimately sums up to 36 target functions. The results have been presented to an expert software architect and his judgement of the generated unit test code has been obtained.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Indika Perera, who guided me from the initial stages from this work to complete it successfully, providing the much-needed advice, material, and the knowledge.

Then, my sincere thanks go to my mother, brother, and my wife, not only being patient with my academic work committing my family time but by supporting me and encouraging me to continue this work till it succeeded.

At last but not the least, I would like to thank my colleagues, both in my MSc batch and in my office, Metatechno Lanka Company (Pvt.) Ltd., who helped me in various ways for making this work a success. Thank you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
API	Application Programming Interface
CPU	Central Processing Unit
DB	Database
GUI	Graphical User Interface
ID	Identification
IDE	Integrated Development Environment
JML	Java Modelling Language
JPF	Java PathFinder
MPOS	Mobile Point of Sales
NASA	National Aeronautics and Space Administration
NFC	Near Field Communication
OOP	Object Oriented Programming
POC	Proof of Concept
OS	Operating System
QA	Quality Assurance
RFID	Radio Frequency Identification
SAM	Secure Access Module
SQL	Structured Query Language
UI	User Interface
VM	Virtual Machine
XML	Extensible Markup Language

1 INTRODUCTION

1.1 Unit testing

1.1.1 What unit testing is

Unit testing is also known as ‘Developer testing’. Software development process has seven major phases as; requirement engineering, feasibility study, design, development, testing, deployment, and maintenance. For some of these phases, there are sub-phases. As an example, the design phase has two major sub-phases as, basic design and detailed design. Likewise, there are few sub-phases in the testing phase also. One of the important sub-phases of it is this unit testing. It is also considered as the foundation for other testing sub-phases such as integration testing and system testing [1]. Unit testing is carried out by, in most cases, the software developer of a source code himself. But, in some cases, one of the other software developers in the team may do it. However, it is a task for the software developers to carry out [2].

Unit testing is the place where it is tested for the qualification of integrating a certain code to a software system. That is why it is carried out before such integration. Therefore, in unit testing, it is expected that the developers must take the responsibility of their own code and carry out the testing of the units thoroughly before the integration happens [3]. By doing that, it will make the integration testing more efficient as no unit testing level bugs would be identified at the integration testing level. Unit testing is not there to replace the integration testing. Integration testing is also another sub-phase of the testing phase, which is carried out by separate QA (Quality Assurance) teams in the same organization which does the development, or a separate QA organization, joining with the development organization. Instead of replacing, as it is also discussed above, unit testing is there for making the life of QA easier by allowing them to focus on system and integration level defects in a software system [3] [4].

As it is implied by the name itself, unit testing is required to create small test cases which focus on the units, in most cases functions in the code [3]. There can be many unit test

cases for the same unit for unit testing, with different inputs to check how the unit behaves with such various inputs. As the focus is only on the targeted part of the code, the whole system is not needed to be executed for obtaining unit test results [3]. The execution of the targeted unit is enough for this, as however, it is a testing done before the integration happens.

1.1.2 Merits of unit testing

It is widely recognized that, execution of unit testing is one of the most effective ways to identify defects. As a result, it gives many advantages to the software development project and the software system or the product which that project outputs. Not only the researchers who perform research on software, but also the actual practitioners who are in software development field agree on this [2]. Some of those advantages can be listed as follows [3].

1. **Improves the quality of the software system:**

   There can be many defects in the source code units. Those defects must be identified and fixed before the code is released for the integration testing. Otherwise, the sub-phase where those defects would be identified could be the integration testing. There, the QA teams, or the QA organizations can have schedules and estimations for the integration testing effort, and they could spend most of that allocation, identifying unit level defects. This could easily limit the real integration level testing, both in the amount wise and the intensity wise. Ultimately, this leads to the reduction of the quality of the software system. Having unit testing executed improves the quality of a software system [5] [6].

2. **Reduces the software project cost:**

   Unit testing is the immediate sub-phase carried out in the testing phase, once after the code is produced. It is the earliest stage in testing phase to identify defects. In a software system, the earlier a defect is identified and fixed, the lesser the cost it would incur. The cost for identifying and fixing a defect in the integration testing level or after the deployment and in the maintenance phase, is orders of magnitude

greater than that of identifying and fixing it at the unit testing level [4] [7]. By unit testing, it technically means that, it is still the development going on. Unit testing could be a significant cost saver for a software development project [5].

3. **Accelerates the development:**

   If a code is not unit tested, it could impede the speed of the development of the software system, mainly when integrating with other developers' code. When at such integration, unit level defects could be there in the code and exposed when referring that code from a code of another developer. Here, it could easily waste some time to identify where the defect is and figure out that it is a defect that should be identified and fixed at the unit testing level. This process impedes the development. In contrast, having unit test performed accelerates the implementation of a software system by guaranteeing such unit level defects will not exist at the integration level.

4. **Ensures timely completion of a software development project:**

   When the schedule for the software development project is created, certain time frames are allocated for both development and testing phases. But, as a result of not performing unit testing, number of unit level defects could be identified at the integration testing. This could easily make the integration level testing lengthier, shifting the software development project deadlines further. Further, the time taken for identifying and fixing a defect in the integration testing level or after the deployment and in the maintenance phase, is orders of magnitude greater than that of identifying and fixing it at the unit testing level [7]. Time taken for such activities could also shift the software development project deadlines further. By unit testing, it technically means that, it is still the development going on and the unit level defects identified could be fixed within the given time frame in the schedule. Unit testing could ensure the timely completion of a software development project [6].

1.1.3 Negligence of unit testing

There is no doubt that there are considerable benefits of carrying out unit testing. Further, all the software developers who take part in a software development project are aware of those. But, in practice, only a small percentage of them carries it out. Even within the ones who practice it, it is not so consistent [3].

When the reasons for this are considered, there are two major ones as follows.

1. **A problem in the software development culture:**

   Even though software development field does not have a long history, from its beginning, there had been no concept of testing the code by the software developer himself. Further, for such kind of a task, the software developers had not been responsible. As a result, most of the software developers have struggled to come out of that mindset and therefore, adequate unit testing has not been carried out [8]. Software development activities and testing activities are completely opposite. It has been difficult for software developers to switch between development activities and testing activities. Development activities target to construct things and they need a focused mindset. In contrast, testing activities target to break things and they need a mindset ready for exploring more. When a software developer writes a fresh code and at the next moment, if he is forced to think about the ways that this code could not work or incomplete, it is simply artificial for him. This has been one of the major reasons for separating software development and testing.

   But, with the introduction of Agile software development practices, there is considerable adoption of unit testing. Agile is gradually becoming a successful, popular and a widely used practice for carrying out software development projects and it has rated unit testing very highly in it. Because of the recognition Agile has given to the unit testing, software developers are forced to carry out unit testing.

2. **Inadequate tools:**

   A task is practiced more, if there are easier ways to do it or, there are ways to carry out time consuming and repetitive parts efficiently. But there had not been enough of these for unit testing. Efficient and easy-to-use tools have not been enough for unit testing.

1.1.4 Problems in unit testing

There is no doubt that, even though there are highly considerable merits exist in unit testing, it has not been practiced or even practiced, very inconsistently, by the software developers. Most of the time, carried out unit tests are in low quality, leaving bugs in the systems [1]. In simple terms, the rate of adoption for unit testing has been slow [2].

When the reasons for this situation are analyzed, the main reason had been the inefficiency of some of the tasks in unit testing. Carrying out those tasks are tiresome, as well as those take considerable amount of time to carry out, even those are simple and monotonous. Most of the time, a significant portion of the cost for development has been spent for unit testing, and it is without adding any new features to the product [9]. Further, the software developers waste 10-15% of the time for development, for regression testing [1] [10] [11].

For increasing the practice of unit testing among software developers, there must be means to carry out such tasks in easier ways. Automating those tasks or, making the software developer intervention minimum for those tasks are very good approaches [3] [12].

In this research work also, it is tried to address these problems. There, test code writing has been almost completely automated. Further, for the tasks which must need human intervention such as creating the unit test specifications, automation has been used to minimize the software developer intervention. At the creation of unit test specification, the unit tester only needs to make the minimum inputs needed. All the other inputs are automated by using Microsoft Excel features and formulae. Further, in the unit test code generation tool, the UI (User Interface) has been created keeping this in mind.

1.1.5 Problems in unit testing frameworks

With the identification of the importance of unit testing, frameworks for unit testing such as JUnit and NUnit were also introduced. The main advantage of having those was that, those supported for establishing standards for both writing unit test code and running unit test cases. Earlier, there were not that much of standardization and therefore, software developers and software development organizations followed their own styles, which they thought that were suitable [3].

But these frameworks did not help to reduce the cost of the unit testing process. Following reasons could be identified for incurring high costs [3] [10].

1. There was not an automation of unit test code generation. All the unit test code had to be written by the software developers themselves, by hand.
2. Code coverage has been mainly categorized to three main categories based on the coverage degree of the code to be unit tested. Namely, C0 (line coverage), C1 (branch coverage) and C2 (condition coverage). With unit testing frameworks, even to achieve C0 coverage, which is the simplest among the complete code coverage degrees, it must be written a very large amount of test code [8].

1.1.6 Unit testing tool creation

A tool which is for unit testing should be able to drag more software developers towards carrying out unit testing. For that, the tool should provide solutions to get rid of / minimize the hurdles that exist which keep software developers away from performing unit testing. According to the Agitator software research team [3], for a unit testing tool to become such one, it should address following areas.

1. Automate the tasks in unit testing that do not require human intervention.
2. Provide input values for the components to be unit tested, software developers may not come up with.
3. Generate unit test code automatically, and the generated code should be able to be reused.

Further, unit testing consists of three main components and for a complete automation of unit testing, all these three components must be addressed [1]. Those are;

1. Selecting unit test data
2. Executing unit tests
3. Comparing the results obtained with the expected ones (Test oracle)

In this research work, it is focused more towards the software developer first coming up with a proper unit test specification, and getting it reviewed by an expert in the field before the automatic generation of unit test code. In the Google Test Unit Test Code Generator tool, providing input values is not made one of its tasks. Instead, input values are decided and finalized at the unit test specification completion, and the unit test code is generated for those inputs. The idea is that, deciding input values is a task that requires human intelligence and a software developer must be aware of, for what input he is carrying out the unit testing. But both the software developer and the reviewer could miss important unit test cases. For such cases, an additional support is given to the software developers by giving warnings of missing unit test cases, at the unit test code generator execution time. It is done based on a schema decided by both the unit test specification format and the unit test code generator tool.

If the unit testing tool generates a huge number of input values and the software developer has to go through those and select any interesting ones, it takes additional cost, depleting merits of using a unit testing tool.

Features of a good unit testing tool

According to the Agitator software research team [3], to become a unit testing tool a good one, there are four main aspects to be considered.

1. Usability

   The tool must not put steep learning curves on the software developers, just to understand and use it. Also, it should be able to be used both by the experienced software developers and the ones with no such lengthy experience. A unit testing

tool must not target a specific software developer skill level. It should support the usage of mock object creation [13] [14] [15]. There are complex objects in software code, which is not practical and time-consuming to implement again only for unit testing purposes. The unit testing tool must provide the facility to create alternate objects which has the same interface as of the real complex object and, only needed partial implementation done related to it.

2. Applicability

   Though unit testing tools are presented with using it on very simple code, the reality is greatly different. These real-world application code evolves larger, complex and often nasty, which makes even unit testing by a human, very difficult. A unit testing tool should be able to handle a code base of any level. Not only the complexity, but also the evolving of the code base must also be supported by a unit testing tool. It should not pass the unit test code changes due to the software code changes, to the software developer.

3. Scalability

   Even though at the beginning of a software system it has nice and clear designs, with the time, it becomes bulkier and more complex. New features will be added by several software developers, deviating from the initial design. A good unit testing tool should be able to hold up with this scaling up of the software systems.

4. Performance

   The complexity of software systems could easily affect the performance of those. Generally, as unit testing tools are based on the source code of the software systems to be unit tested, bad performances of those source code can affect badly on the performance of the unit testing tools also. A good unit testing tool could stand up well in such situations, without reducing its performance much.

The unit testing tools available have focused on the above aspects and have made trade-offs. That is because, the purpose of each unit testing tool is different, and the creators have focused more on the aspects that they have targeted in their unit testing tools.

Some tools target the complete automation of the unit testing for the source code written using OOP (Object Oriented Programming) concepts. They hope to operate without human intervention at all, apart from the initial trigger for starting the unit testing process. The argument of the creators of such unit testing tools is that most methods in these source codes are short and because of that, complete automation is possible. But, on the other hand, the object-oriented concepts such as encapsulation used in these source codes makes unit testing harder too. Anyway, they do not throw away the manual testing also. They agree the manual testing is needed to carry out more complex unit testing [1].

1.2 Google Test [16]

1.2.1 Introduction

Google Test unit testing framework can be used to perform unit testing on the software code written in the programming languages; C or C++. As the name implies, it is developed and provided by Google Inc. and, it is an open source framework which means that it is free to use, and anyone can modify/update it.

1.2.2 Features

Google Test has become an emerging tool for unit testing mainly because of the easiness for its usage and it provides features that are very useful for the software developers. Some of the useful features among them can be described as follows.

It helps to identify memory related problems in the target code. Normally, these problems arise only in some executions of the code, not always. Google Test gives the possibility to execute the same test repeatedly any number of times as of the liking of the user. Further, it provides the facility to bring debugger up at the first occurrence on a memory issue, based on the user's preference.

The target programs written in C or C++ programming languages are typically focused on the increased performance. Therefore, in some cases, software developers have disabled exception handling. In contrast to other competitive unit testing frameworks, Google Test provides inbuilt assertions to cater this.

Test execution has been made simpler with Google Test in contrast to the existing competitive unit testing frameworks. It is just a single macro call and there is no need to write additional bulky code.

Exporting a report regarding the unit text execution to a file (XML (Extensible Markup Language) format) has also been made easier with Google Test in contrast to the existing competitive unit testing frameworks. It is just a single command and there is no need to write additional bulky code.

To check whether the expected result is got from the test, there are two macros introduced by Google Test. One is EXPECT_EQ which does not terminate the test execution even if the test failed. The other is ASSERT_EQ which terminates the test execution once the test failed. These can be used accordingly by the software developers based on their requirement.

Google Test is included a unit test filtering feature where the user can select specific unit tests to execute or omit specific unit tests from executing.

Google Test also provides the facility to disable unit tests that are not to be executed, just by adding the keyword DISABLE before the unit test case name. However, framework outputs a warning message after the unit test execution indicating that there are disabled unit tests. Further, Google Test has provided the facility also to execute all the unit test cases including the disabled ones using an option at execution.

Google Test has provided macros, additionally for comparing numbers with floating points. It is useful because different CPUs (Central Processing Unit) and unit test execution environments store these in different manners and therefore, if a user used the default comparison macros, the unit test might fail. These special macros neglect

negligible differences in expected and actual float values and if the user needs, a precision can also be defined to pass the unit test.

Google Test has also provided macros for checking in error cases, whether a program exits with the expected error code and with the expected error message. This feature is useful for unit testing the error handling in the targeted code.

Google Test is included test fixtures. A test fixture can be defined as a separate C++ class. It can add several important methods which run immediately before and after every unit test is executed so that the user can perform initializations, exception handling, cleanup stuff, assertions and define any custom data members as appropriate.

1.3 Unit testing with mock objects [13] [14] [15]

1.3.1 Introduction

Though it is well understood the importance of carrying out unit testing, practically, when the target code gets more complex, it is very hard to unit test in simpler manners. As a result, unit test code becomes more complex and difficult to comprehend. On the other hand, it affects to the maintainability and the quality of the resultant unit test code.

This is where mock objects come into play. With the usage of mock objects, software developers can replace the complex domain code, which is irrelevant to the target unit test, with dummy implementations. These mock objects mimic the real functionality.

1.3.2 Advantages

Using mock objects is a very good technique to overcome many problems the software developers face when writing unit test code. The advantages of using those are as follows.

When writing unit test code, the software developers can focus only on the target domain code, without bothering about setting up domain code statuses which are irrelevant to the targeted unit test but needed for the successful execution of it. In some cases, domain code does not even expose the needed features for such a setup. Simply, mock objects facilitate software developers to unit test one feature at a time.

Mock objects help to improve both target domain code and the unit test code. If the usage of mock objects is complex for a domain code, it means that the domain code needs refactoring. By refactoring, domain code gets improved and at the same time, unit test code also gets improved. Further, mock objects reduce the unnecessary dependencies between modules and reduce the vagueness of the interaction between modules.

Also, there can be cases where mock objects fail as software developer did not expect. These are very good warnings that give the indication to the software developer that improvement is needed for either domain code or mock code.

With mock objects, there will be no untestable domain code. Unit tests can be written for all the domain code including emulating server failures. For some domain code which has states that are difficult/impossible to reproduce, using mock objects is the only mean to carry out unit testing.

Mock objects also make the unit test code much simpler, making it more comprehendible and maintainable. With the usage of mock objects, unit test code will have a common format and that helps the development team with the aspects of comprehensibility and maintainability.

There are some assertions to be carried forward with some new unit tests. But these can be in different unit tests and the software developer can easily forget and get omitted to include these in new unit tests. In contrast, with mock objects, these assertions can be included in mock objects themselves so that, such misses never happen. Further, if new assertions would be found by the software developers later, with mock objects, they only have to add it to the relevant mock object instead of sweeping through every unit test to include those.

When using mock objects, there will not be a need to manipulate domain code for the successful execution of unit tests. As an example, it does not need to manipulate encapsulation. A unit test should know about what the behavior of the domain code should

be and the result it should produce. It is perfectly OK not to know much about the structure of it.

If mock objects have been used, software developers are notified once a problem occurs. It is because mock objects can carry out the assertions at their every interaction with the domain code. Further, it could provide the exact error message with the real reason for the failure.

In contrast, when debugging, it is hard to identify the exact point which an error occurred as domain objects fail sometime after the actual error occurred. Also, without mock objects, the unit test code for checking the domain objects asserts after domain code executed. It also makes the identification of the exact point which error occurred, very difficult.

With mock objects, software developers can start writing unit test code related to DB (Database) operations before a working DB is implemented. On the other hand, mock code also provides an idea on the DB functionalities required by the domain code so that, those functionalities can be considered when creating the DB infrastructure.

1.3.3 Limitations

Of course, mock objects also have some limitations which to be concerned when used. But, most of them are common to any unit testing. Not only for mock objects. Some of those limitations are as follows.

Due to errors of software developers, mock objects might contain incorrect code. As an example, the original code returns a value in one type of units but erroneously, mock object code returns that value in another type of units. However, this is common to any unit testing, not only for mock objects.

In principle, unit testing checks the individual components. Not the interactions between those components. Therefore, mock objects can pass unit tests for the individual components but when considered with interactions, it could be wrong. This is also common to any unit testing, not only for mock objects. Even with excellent unit testing,

integration testing is also needed to be carried out, especially to test the interactions between individual components.

Though mock objects bring the simplicity for unit testing, with some complex libraries, it could be very hard to emulate those with mock objects and it could be very costly. In such scenarios, mock objects should not be created. However, mock objects come in handy, if only a part of such complex libraries is to be emulated.

1.3.4 Google Mock [17]

As the Google Test framework, Google Mock is also a framework for C++ programming language, to write and use C++ mock classes. As the name implies, it belongs to Google Inc. It is a part of Google Test C++ testing framework and works seamlessly with it. In this research work also, Google Mock has been used to use mock objects in the generating of unit test code.

The idea of creating a mock framework for C++ programming language has been got from the mock frameworks created for other languages. The design of Google Mock has been done focusing on C++ programming language.

As any usage of mock objects in unit testing, Google Mock also improves both system designs and unit test cases. Further, it gives support for the software developers to write simple and comprehendible unit test code.

1.4 Problem statement

In this research, it has been identified four problems that exist in the unit testing domain.

One of those problems is that, though unit testing is an important phase of the software development life cycle, it needs much effort to be completed successfully and comprehensively. This effort takes the valuable time of the software developers. And, with the effort it takes, a considerable amount of human resources needs to be put. Also, this could lead to longer schedules, making the product delivery late. Considering these, the project teams could overlook unit testing, as it is thought that integration testing could compensate for the unit testing.

Another problem is that, the depth of the unit testing done by the software developers. They could either think of some trivial unit test cases and write unit test code for those, or, they could carry out unit testing just by debugging the code and changing the values of the variables at runtime. These would not serve the purpose of carrying out the unit testing at all.

Also, not keeping evidence for carrying out unit testing, especially in a presentable way to the clients (especially, the clients who have a knowledge about software development and the clients who are also software development organizations and have outsourced a part of their work) and also in a form which can be easily referred in the future, is another problem. Further, in cases where changes in source code happen, there would not be a proper track of the changes done for the unit test cases.

Finally, according to the research done in this research work on the unit test code generator tools, there are hardly any such tools for Google Test unit testing framework, which is a giant emerging one for lower level programming languages. Almost all such tools have been targeted higher level programming languages such as Java.

1.5 Research objectives

There are three main objectives in this research.

The first objective is to give a solution for the 2nd and 3rd problems mentioned in '*1.4 Problem statement*' section above. That objective is, creating a comprehensive format for creating unit test specifications. The developer should adhere to the guidelines to create it and include the unit test cases. The created unit test specification must be reviewed and finalized fixing issues, before carrying out unit testing. By this way, it could guarantee that the unit test cases are comprehensive and ample. This format is based on spreadsheets and, Microsoft Excel is used for creating it.

The second objective of this research attempts to provide a solution for the 1st and 4th problems mentioned in '*1.4 Problem statement*' section above. It is to automate unit test code writing to reduce the effort needed to be put for carrying out unit testing. But, here,

the unit test code generation is done based on a unit test specification created, adhering to the proposed unit test specification format in the research objective 1. By this way, it is guaranteed that all the decided unit test cases are carried out without missing any, and the unit test code is written very quick, reducing the unit test code writing time drastically. In this research, it is written unit test code for C++ programming language based on the Google Test unit testing framework. But this concept is not restricted only to these technologies.

Though how many reviews are done for the unit test specification, there could still be missing unit test cases as occurrence of human errors is inevitable. The third objective in this research is, introduce a methodology to reduce such errors using both unit test specification format and unit test code generator tool proposed. Here, it is targeted to do a POC (Proof of Concept) to show that a schema could be defined, and the tools could be improved further to reduce human errors.

1.6 Motivation

The main motivation for this research emerged because of the importance of the testing phase of the software development life cycle and, unit testing being one of the major parts of it. The idea of this research is to contribute to this area.

Also, the experience the researcher has gained working in the software development industry has motivated to carry out this kind of a research work. There, the researcher has seen and heard of the cases where, unit testing phase has been neglected, not carried out well, and, not keeping track of the unit tests carried out. Here, the idea is to provide a solution considering the problems to be addressed for carrying out comprehensive unit testing.

2 LITERATURE REVIEW

In this section, three of the research works related to the unit testing area have been selected and a description, what the researchers have achieved and what are missing in the research work for each research have been described. Finally, a comparison of these research works has been carried out.

2.1 Agitator software [3]

The researchers have created a tool to be used for unit testing, called Agitator. It is based on software agitation. When it is created, results of following two research areas have been adopted, relating automating unit testing.

1. Test input generation

   This is simply generating possible values to be used as the inputs of the unit test cases to be executed in unit testing.

2. Dynamic invariant detection

   An invariant is a property that is held, either at a point or at several points within a computer program. This research area is focused on detecting those invariants at a program's execution time.

This tool is for unit testing source code written in Java programming language.

2.1.1 What software agitation is

Software agitation simply is a technique that can be used for unit testing. In *Figure 2-1* below, the overview of software agitation is displayed.

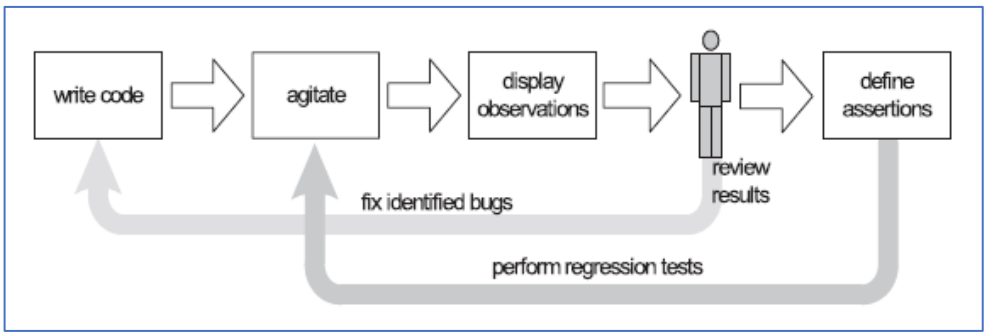


Figure 2-1 Software agitation workflow overview

Software agitation is invoked by the software developer upon finishing writing code to be unit tested. As the result of software agitation, a set of observations is presented to the software developer. Software developer checks the presented observations for bugs, and if there exist bugs, fixes those and agitates again until he gets a satisfied set of observations. When a satisfied set of observations is received, assertions for that source code is defined and agitation is performed again. These agitations with assertions defined are used to identify software degradations.

There are three main phases in the software agitation process. They are;

1. Creating instances of the classes to be unit tested.
2. Calling every function in those classes using input data generated by the Agitator software itself.
3. Recording the obtained results for further analysis.

2.1.2 What is achieved

Adoption of research work

In research world, there are ample resources for creating highly effective unit testing tools. But the problem is, those results are not presented to the software developers, in a form that they could use for unit testing. As a remedy, the researchers have adopted some important research ideas related to unit test automation, when they created the Agitator software. But, according to the researchers, it was a very challenging task.

One of the research works adopted by the researchers is that the work by Ernst et al. in Daikon Invariant Detector [18]. It detects invariants with a very high accuracy, by executing software systems with a bunch of different inputs. The researchers have identified that Daikon's work can be utilized in the software unit testing domain in two ways.

1. The predicted invariants can be used to observe the behavior of the software system. The software developer can use these to compare the software system's current behavior with the expected behavior. If it is different, it is a defect in the

software system and the software developer can fix it as early as possible in the testing phase.

2. The predicted invariants can be used as unit test cases in the unit test specification.

But, one of the limitations in Daikon Invariant Detector had been the wide range of input data it needed to generate a considerable number of invariants. Here, the Agitator software researchers have catered this by combining the results of several other research on automatic unit test input data generation.

Another application which is like the Agitator software has been proposed by the researchers Xie and Notkin [19]. But their application required pre-existing tests for its execution. In contrast, Agitator software is not expected such tests beforehand, which is a plus over that application.

Reducing grunt work

Though software agitation does not generate unit test code, it reduces some of the boring and repetitive work that are needed to be carried out before generating it. This helps the software developer on focusing more important and challenging tasks in the process.

Generating input data

Agitator software generates a wide variety of input data for every function targeted for unit testing. As also discussed in *1.1.6 Unit testing tool* , Agitator software could suggest input data which the software developer would not come up. Further, the researchers have provided test input factories to generate test input values. They have included a set of pre-defined factories in Agitator software. Also, users can define their own factories using the APIs (Application Programming Interface) provided. Of course, generated test input factories by a software developer can be used by the other software developers, making those as assets for the project.

Giving more control to software developer

The researchers imply that, as they do not decide the unit test specification inputs in the Agitator software, they have given the control to select the inputs from the generated values to the software developers. But they generate candidate unit test specification inputs for the software developers to select from.

Reducing deviation from development workflow

The software developers have used to a workflow and, unit testing should not disturb that. To reduce this disturbance, Agitator software's interface has been implemented on top of Eclipse IDE (Integrated Development Environment), as the researchers have carried out their work targeting Java programming language.

Presenting information to software developers in a clearly understandable way

When Agitator software displays invariants and assertions to the software developer, the Boolean Java expression format is used. This helps the software developer to understand the outcomes of the tool, very clearly. It will not be an extra burden on them.

Providing code coverage

Agitator software provides a code coverage view to the software developer. The coverage metric used is line coverage but including the coverage for every condition in a branch. This helps the software developer to identify the areas of the source code which are not covered by the tool and the needed input values to cover those areas. On the other hand, the unreachable paths in the source code can also be identified.

Lesser test code needed

If Agitator software is used, to get a good code coverage, lesser number of lines of test code is needed, compared to other existing unit testing tools. Though there is not a defined universal norm, the researchers have compared Agitator software's statistics with another software project, which is called 'Commons-collection library', a part of the Jakarta project [20]. In numbers, in Agitator software, 1 test code related line is for 2.3 application

line codes. In contrast, in Commons-collection library, 1.63 test code related lines are for 1 application line code. Further, even with these statistics, Agitator software had achieved 80.2% code coverage, while Commons-collection library had achieved only 77.77% code coverage. All these values indicate that with the usage of Agitator software, software developers have only to write lesser test code to achieve a better code coverage.

2.1.3 What is missing

Not generating unit test code

It is important to note that, software agitation does not generate unit test code. It just checks the code's behavior when unit testing would be performed on it. Once a code is in a satisfied level after several agitations, software developer has to create a unit test for each satisfied behavior. Selecting the inputs to be used (from the wide variety of inputs generated by Agitator software) and determining the correct output for a given input, which is also known as Test Oracle Problem, are still responsibilities of the software developer, with Agitator software.

No guidelines for unit test specification

In this research work, the researchers admit that a unit test specification must be created before the execution of the unit testing. But here, it exists as a task that would cost. But, apart from the possible input data that can be used for it, there is no help or guidelines provided for the software developer to create the unit test specification.

Have some manual work to be carried out

Even though the researchers have tried to make Agitator software as automatic as possible, there are still some work that needs to be carried out by software developers' hands. Mainly, three tasks can be identified as follows.

1. Check observations provided by Agitator software.
2. Decide for suitability of the observations and promote those as assertions.

3. Add assertions manually if the resultant quantity and quality of assertions are not satisfied.

Percentage of observations promoted to assertions are less

According to an experiment carried out by the researchers themselves, only 11% of the observations suggested by the Agitator software have been promoted as it is to assertions by software developers. As a result, the software developers have to check 89% of the useless observations. Of course, some of those are modified by the software developers and promoted as assertions. But it costs and the 11% percentage needs to be improved.

2.2 A complete automation of unit testing for Java programs [1]

At the writing of the referred research paper, this had been an on-going project. The researchers' main target had been the object-oriented programs, to carry out unit testing on them. Complexity and not having enough work done related to it had been the reasons for their selection. Further, they had also been trying to automate unit testing of those programs in a complete manner, covering all the aspects of unit testing. By this automation, the researchers' ultimate target had been to reduce the cost for unit testing.

They were trying to use following three technologies in this research work, for the three major components in unit testing.

1. **Genetic Algorithms (For selecting unit test data) [21]**
   The researchers had been targeting this technology to be utilized for generating unit test input data automatically. This is a technique that is used recursively to improve the quality of the final test input data which is to be used for the unit testing.
2. **JUnit (For executing unit tests) [22]**
   JUnit is an open source unit test framework built to be used for Java programming language. As in many such frameworks, test classes are needed for performing unit testing and the researchers have made to generate these test classes automatically in their work.

3. **JML (Java Modeling Language) (For test oracle) [23]**

   The researchers have used this technology to write program specifications within the source code itself. Those specifications have been utilized to obtain the unit test results, to decide whether a unit test case is passed or failed. Further, this has been used in test data generation for removing useless test data.

The complete automation of unit testing has been depicted in *Figure 2-2* below.

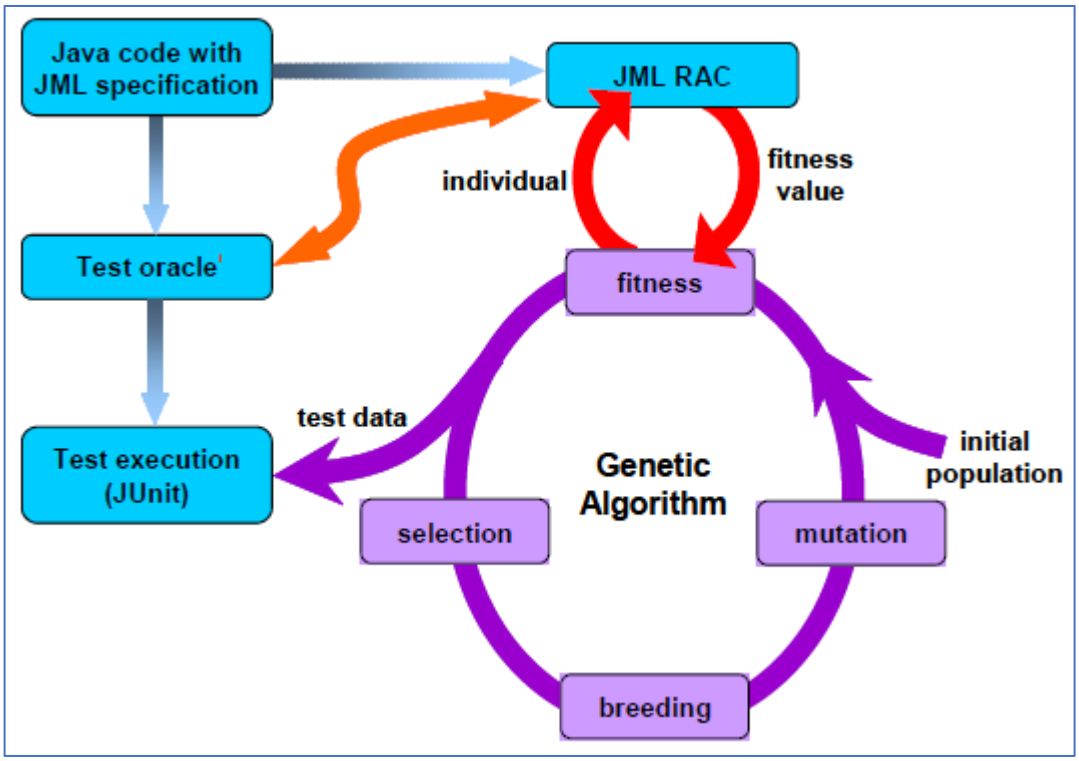


Figure 2-2 Unit test automation using Genetic Algorithms, JUnit and JML

The process followed here is as follows.

1. Check the method signature information in classes in Java source code and create the initial population of test data randomly.
2. Calculate fitness value of each test data based on the coverage on the Java source code.
3. Remove useless test data using the calculated fitness values and JML preconditions.

4. If the resultant set of test data does not satisfy the passing requirement to be fed into unit test execution, test data is improved using genetic operations (i.e. crossover and mutation) on it. Here, based on the fitness value of each test data, ones to become parents are selected. Then, those parents are combined using the crossover operation to create new offspring test data. Further, generated offspring test data is mutated to diversify the test data.
5. New set of test data is selected from the newly created offspring test data and the old set of test data.
6. This process is carried out repeatedly until finding a suitable set of test data.
7. Once the set of test data to be used in unit testing is finalized, it is used with JUnit for unit test execution.

2.2.1 What is achieved

Support for test data selection

There are two ways to select unit test data called black box and white box. In black box method, only the information available regarding the inputs and outputs of a function are considered when selecting test data. This is also called 'specification-based approach'. In contrast, in white box method, function's source code is also considered for this selection. This is called 'program-based approach'. However, both these methods are essential to be followed for a complete selection of unit test data and the researchers' work support both these methods.

Usage of JML for test oracles

As this work is targeted for Java programming language, the life of the software developer has been made easier by using JML for writing program specifications. The software developer can write these in this familiar specification language, instead of using a completely strange one. Further, the specifications are always with the source code itself so that, the maintenance of the unit test specifications are easier.

Applying Genetic Algorithms to OOP

In the previous work where, Genetic Algorithms have been applied to unit testing, the needed attention has not been given to OOP [24]. Here, the researchers have focused on applying Genetic Algorithms to OOP. They have provided improved solutions for performing genetic encoding of objects, performing genetic operations with objects, and calculating fitness values of objects.

Creating a POC tool for complete unit test automation of object-oriented programs

This tool has been developed targeting Java programming language and, it is capable of carrying out all the aspects of unit testing automatically, with a single mouse click.

Extending an existing test case generation tool

One of the authors of this research work had already built a unit test case generation tool [25]. In this work, the researchers have extended that tool to generate both test oracle class and test data class (with test data) for each unit test case.

2.2.2 What is missing

The project was on-going

Even though the researchers describe the concepts and technologies they are going to use in their research paper, the work was not completed.

Genetic algorithms not implemented

At the point when the researchers have written the research paper, they have not integrated Genetic Algorithms for unit test data selection, in their solution. Instead, they have only generated these test data randomly.

Need to change JML specifications if source code changed

Despite researchers say that it is fully automated, if a change in source code happened, it is needed to alter related JML specifications manually.

Learning curve exists for writing JML specifications

Though the JML specifications are written within the same source code as special comments, the language syntax is quite different from Java programming language and therefore, there exists a learning curve, which is tedious for the software developers, especially who are new to the paradigm. Also, there is a considerable amount of specifications have to be written too for each and every method.

Reduces source code readability

According to the examples given by the researchers, JML specifications have to be inserted within a same source code line, in between source code lines or above the method. Also, they have not specifically mentioned that, how to add these specifications and method comments at the same time. All of these makes the appearance of the source code more complex, reducing the readability.

2.3 Combining unit-level Symbolic Execution and system-level Concrete Execution for testing NASA (National Aeronautics and Space Administration) software [26]

The researchers have introduced a mean to automate testing and error detection of the software which are complex, and which need thorough consideration related to the safety aspect of its usage. They have targeted the software in NASA where the software has the said properties. But, according to the researchers, they were not hoping to be limited only for NASA and they claim that their approach can be utilized by such complex and safety critical software. For creating test cases needed for testing such software, the researchers have combined two techniques as follows.

1. Symbolic Execution [27]

   This can be done at many phases as it is developed in a general manner but, is most suitable to be done with the unit testing. In this way of analyzing software programs, instead of real values, symbols are used to represent variables. It generates a path condition for every path in the target program and determines the

feasible paths. Those feasible paths are solved to get the input data for the unit tests guaranteeing the complete coverage of the program.

Here, the researchers have used an existing model checking tool called JPF (Java PathFinder) and introduced a new framework (called Symbolic Java PathFinder or Symbolic JPF), doing an implementation of an interpreter on it.

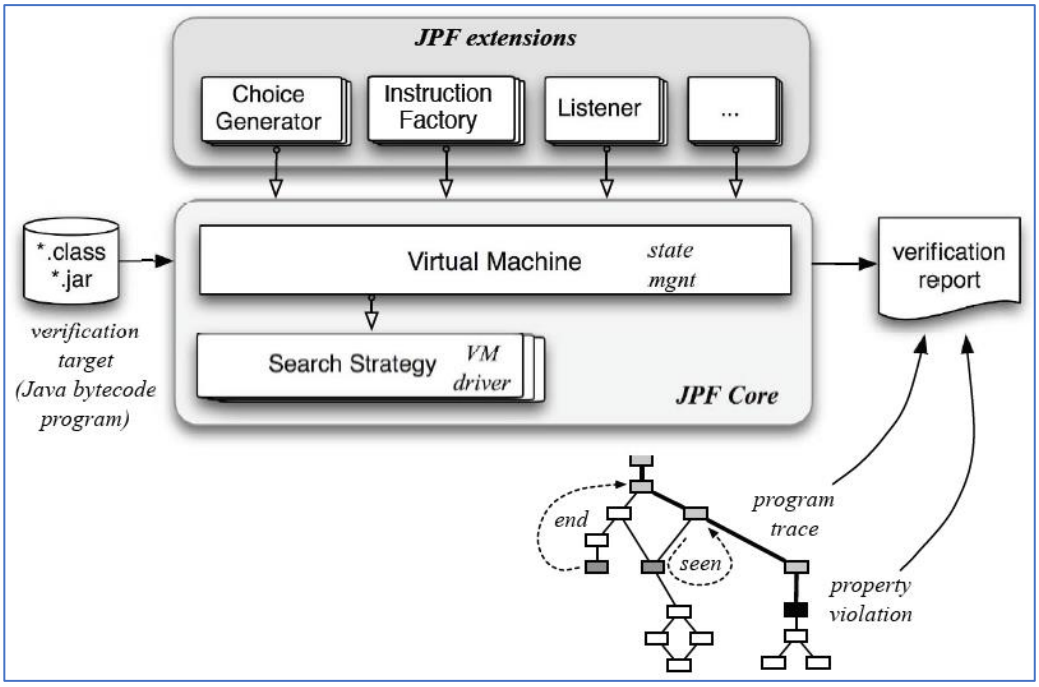


Figure 2-3 High-level structure of JPF

JPF consists of three main parts which carry out the following tasks in the tool.

i. VM (Virtual Machine): Carries out the state management of the execution.
ii. Search strategy: An object that carries out the execution in VM.
iii. JPF extensions: Configurable extensions to the JPF Core, which is the combination of VM and Search strategy.

The researchers have utilized the JPF Core as it provides techniques to identify imperfections in the source code. In addition, they have picked up some JPF

extensions and configured those to implement Symbolic Execution Framework. It has resulted in the Symbolic JPF.

They have expected following two tasks to be carried out by it.

i. Automating test case generation
ii. Checking source code properties (This is done at the same time where test cases are generated)

2. Concrete Execution

   This is done for the whole software. As the Symbolic Execution is run based on the attribute values, at Symbolic Execution, it is always checked that the associated attributes exist and valid. If that check failed, the Concrete Execution takes place. This way, both Symbolic Execution and Concrete Execution work hand in hand to execute the unit testing. This has also been used to increase the accuracy of the created unit test cases.

2.3.1 What is achieved

Obtained very good results by testing NASA software with this work

The researchers have used the outcome of their research work to test software in NASA and have been able to find defects in those. One of the defects they found has resulted in a design change in that tested software too. Further, at the test case generation phase using the Symbolic Execution Framework also, it had identified defects in the software.

Obtained improvements in symbolic execution approach

One of the problems in the prior work related to Symbolic Execution is that it needed to manipulate source code for initial analysis which takes extra effort. But in this work, such code manipulation is not needed as needed information is held dynamically and the analysis needed is done at runtime. Because of this, the symbolic data has also become up to date compared to the previous approaches.

Obtained high code coverage

The test cases generated by Symbolic JPF provide a very good code coverage even for the specifications for coverage set by the users.

Avoid generation of useless test cases

Typically, for avoiding generating test cases with input data not suitable for the context of the unit to be tested, it takes some cost as it is a manual process. The researchers have introduced two techniques in their work to automate this.

1. Starting Symbolic Execution at any needed time. It can be started after setting up the context needed (with the realistic input data) for the test by the Concrete Execution.
2. Running simulations to obtain the limitations based on the context for the input data. Those limitations are then used as pre-conditions for Symbolic Execution.

Generating test cases (which gives full coverage) quickly

According to the researchers, to generate test cases which give the full code coverage using Symbolic Execution Framework, it takes less than 1 minute compared to the approximate 20 hours (or more) for generating those manually. But this manually generated test cases also gives only a partial code coverage.

2.3.2 What is missing

Not automated Symbolic/Concrete execution switching

In this work, the users of the tool have to manually instruct it to perform the Symbolic Execution and to give the methods and inputs which are to be considered for Symbolic/Concrete Execution. Otherwise, they have to do some development work by creating a listener to automate this.

Correlation analysis for complex correlations not performed

Correlation analysis is performed to determine the constraints for the inputs of unit tests. In the researchers' case study, they have proven this concept only with simple correlation analysis.

Have to encode additional constraints separately

As there can be cases where un-realistic unit test cases would be generated, when using the researchers' approach, the users have to introduce additional constraints separately, which incurs an additional cost.

2.4 Comparison of unit testing tools

When the research work discussed here are compared based on some properties defined, the result is as follows. Note that any of those does not provide a unit test specification format or a report, though all of them generate unit test input data.

Key:

Tool 1: Agitator software

Tool 2: A complete automation of unit testing for Java programs

Tool 3: Combining unit-level Symbolic Execution and system-level Concrete Execution for testing NASA software

Table 2-1 Comparison of unit testing tools

#	Feature	Tool 1	Tool 2	Tool 3
1	Help or guidelines provided to create unit test specifications	No	No	No
2	Generating input data	Yes	Yes	Yes
3	Support for including input data that could easily miss	Yes	Yes	No
4	Deciding unit test inputs and outputs by the tool	No	Yes	Yes
5	Avoiding generation of useless unit test cases	No	Yes	Yes
6	Manipulating source code	No	Yes	No

7	Generating unit test code	No	Yes	Yes
8	Ability to provide a unit test report	No	No	No
9	Amount of manual work to be done	High	Low	Medium
10	Level of learning curve	Medium	High	Low

3 METHODOLOGY

Here in this section, mainly, the design, and the architecture of the two main outcomes of this research work are explained. The items to be explained are as follows.

- Unit test specification
- Unit test code generator tool
- Schema definition
- Unit testing procedure

3.1 Unit test specification

The unit test specification proposed is a comprehensive one which covers almost all the aspects of the unit testing phase in software development life cycle. After carrying out unit testing and filling the needed information in this specification, it could simply be used as a unit test report to be submitted to the needed parties such as clients and other project stakeholders and further, to be kept as a reference. In the specification, there are cells where the creator needs to manually insert data (cell background is in yellow color in the specification), and also, there are some cells where the data is automatically inserted (cell background is in green color in the specification), referring other sheets and areas of the specification. By having such cells, it improves the efficiency of creating the specification and, increases the accuracy of it by reducing the chances for human errors.

3.1.1 Format

The format discussed here is a comprehensive one suitable for unit testing, with several spreadsheet sheets which include important information. Though only a part of these sheets directly related to the unit test code generation, to have a complete specification, it is designed considering the information that is important (including code coverage).

Here onwards, the sheet wise explanation is done, mentioning the information included in those sheets.

Cover: This is the first page of the unit test specification. It gives the introduction to the specification with the following information.

- Targeted source code file name
- Latest date updated
- Version

Here, only the targeted source code file name is needed to be inserted manually. Other information is filled automatically.

UpdateHistory: This is the sheet where the changes for the unit test specification are tracked. By referring 'UpdateHistory' sheet, the reader can get an idea of the evolution of the specification. The information in this sheet is in the tabular form and that table has following content.

- History record sequence number
- Date when the change is done
- Version which the change is done
- Description of the change
- Person who did the change

Here, only the sequence number is increased automatically. Other information is needed to be inserted manually.

FunctionList: This is the sheet where the list of functions in the unit test specification is maintained. By referring this sheet, the reader can check whether a specific function is covered in this unit test specification and retrieve the sheet name of the specification of that function. The information in this sheet is in the tabular form and the table has following content.

- Record sequence number
- Name of the sheet which unit test cases are available
- Name of the unit test target function

Here, only the sequence number is increased automatically. Other information is needed to be inserted manually.

UnitTestResults: In this sheet, the summary of the unit test results is available. By referring this, the reader can get an idea of the status of carrying out unit testing on a specific source file. However, the only thing the unit test specification creator has to insert manually in this sheet is the 'Sheet Name'. All the other information in this sheet is automatically inserted, referring to the specific unit test specification sheets. The information in 'UnitTestResults' sheet is in the tabular form and the table has following content.

- Record sequence number
- Name of the sheet (No.x) where the relevant unit test specification is available
- Link to the specific unit test specification sheet
- Target function name
- Number of unit test cases
- Number of unit test cases passed
- Number of unit test cases failed
- Number of unit test cases not tested
- Percentage of unit test cases passed
- Totals in the whole unit test specification (Unit test cases, passed, failed, not tested)
- Percentage of unit test cases passed in the whole unit test specification

CoverageResults: In this sheet, the summary of the code coverage results after carrying out unit testing is available. By referring this, the reader can get an idea on how unit testing has covered the code in the aspects of line coverage, function coverage and branch coverage. Of course, there can be source code lines that are impossible to pass using a code coverage tool. In such cases, those lines are covered manually by observation and information related is inserted in this sheet. The information in 'CoverageResults' sheet is in the tabular form, in two tables named as 'Observation' table and 'Summary' table.

Observation table has following content.

- Record sequence number
- Target function name
- Reason for the impossibility of coverage using the coverage tool
- Number of code lines covered
- Number of complete target functions covered
- Number of branches (conditions) covered in the target function

Here, only the sequence number is increased automatically. Other information is needed to be inserted manually.

Summary table has following content related to lines, functions and branches in the target source code.

- The coverage done by the coverage tool
- The coverage done by observation
- Total coverage achieved by the coverage tool and observation
- Coverage that is there to be achieved in the target source code
- Percentage of coverage achieved

Here, the coverage done by the coverage tool and the coverage that is there to be achieved in the target source code need to be inserted manually. Other information is inserted automatically.

BugList: In this sheet, the list of bugs identified during the unit testing is available. By referring this, the reader can get an idea on the status of fixing bugs which were identified at the unit testing phase. In this sheet, part of the information is inserted by the person who carries out the unit testing. The rest is inserted by the person who fixes the bugs. The information in 'BugList' sheet is in the tabular form and the table has following content.

- Record sequence number
- Name of the unit test specification sheet (No.x) related to the bug

- Link to navigate to the unit test specification sheet (No.x) related to the bug
- Short name for the bug
- Description of the bug
- Person who identified the bug
- Date on which the bug is reported
- Status of the fixing of the bug
- Fix for the bug
- Person who fixed the bug
- Date on which the bug is fixed
- ID (Identification) of the commit to the source controlling tool
- Additional information regarding the bug or the fix

Here, only the sequence number is increased automatically. Other information is needed to be inserted manually. Further, as the status of the fixing of the bug, a status from the following statuses can be set by the user and the color of the relevant table row changes accordingly. This is to easily identify the status of the fixing of the bugs.

Table 3-1 BugList table row color changes based on status selection

#	Status	Color
1	Not Started	Red
2	Ongoing	Yellow
3	On Hold	Ash
4	Fixed	Green

No.x: This is the sheet to enter the unit test cases for each function in the source code file. The letter ‘x’ in the sheet name ‘No.x’ will be starting from 1 and incremented by 1 for each new function (E.g.: No.1, No.2, No.3, …). The unit test code generator tool accesses and reads these sheets in the unit test specification to create unit test code so that, it is important to adhere to the rules when creating these sheets and also, use the names and

types from the source code exactly as they are. The information in ‘No.x’ sheet is also in the tabular form in two tables for unit test cases and summary.

Table for unit test cases has following content.

- Unit test case number
- A short description of the unit test case
- Unit test case name
- Target function parameters and return value
- Global variables
- Function calls return value, call count and parameters
- Result after execution of the unit test case
- Date that the unit test is carried out
- Additional information regarding the unit test case

Here, only the sequence number is increased automatically. Other information is needed to be inserted manually.

Summary table has following content.

- Name of the source code file where the target function is available
- Name of the target function
- Total number of the unit test cases written in the unit test specification sheet
- Total number of passed unit test cases in the unit test specification sheet
- Total number of failed unit test cases in the unit test specification sheet
- Total number of unit test cases that are not tested in the unit test specification sheet

Here, source code file name and target function name need to be inserted manually. Other information is inserted automatically.

3.1.2 Color codes

For the ease of identification for both the unit test specification creator and reader, a color code has been used for representing the cells in the spreadsheet based on the information possess. The colors used are as in the following table.

Table 3-2 Unit test specification color codes

#	Color	Description
1	Yellow	The unit test specification creator has to insert data manually to the cells in this color.
2	Green	Data for the cells in this color is automatically generated so that, unit test specification creator should not try to edit content of those cells.
3	Blue	The table headers are colored in this color in the unit test specification.
4	Gray	To indicate where the new rows to be inserted for the tables in the unit test specification, this color has been used.
5	Light Orange	To highlight the input value header section in the 'No.x' sheets, this color is used.
6	Light Green	To highlight the output value header section in the 'No.x' sheets, this color is used.

3.2 Unit test code generator tool

In this research work, it has been developed a software tool for generating unit test code based on the unit test specification suggested. For this tool, the sheets 'No.x' are used for the generation of unit test code. This tool is for generating unit test code automatically for Google Test framework to test source code written in C++ programming language. This tool is only one possibility and, tools for any test framework and any programming language can be built, which generate unit test code based on the unit test specification in this research work.

3.2.1 Architecture

The architecture of the unit test code generator tool is a simple one, which serves the purpose very neatly. It basically gets the necessary inputs from the user, analyzes the relevant content in the unit test specification, and generates the unit test code based on the unit test specification content. From here onwards, the basic design of the tool is described with the help of class diagrams and sequence diagrams.

Main class structure

The main class structure of the tool is as follows. The rest of the classes help to hold the structures needed in the unit test code generation process and are used accordingly in the needed places.

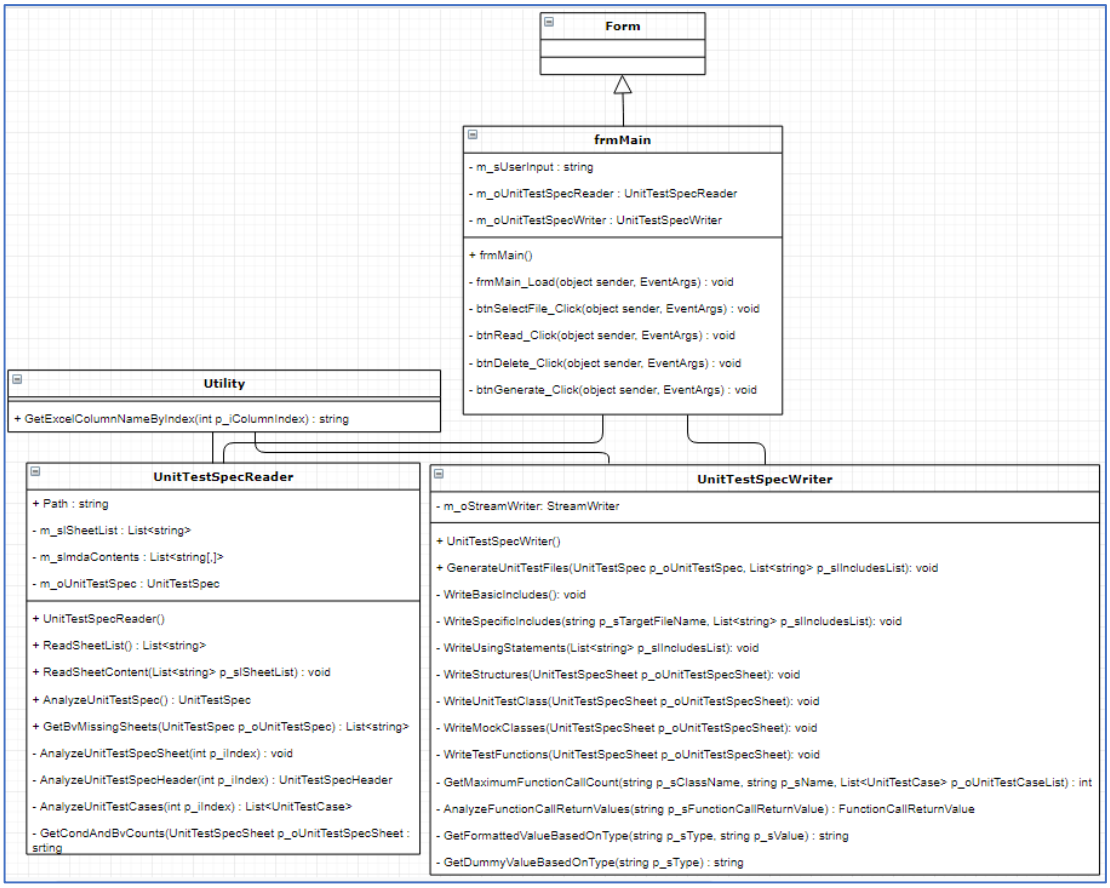


Figure 3-1 Main class structure of Google Test Unit Test Code Generator tool

The ‘frmMain’ class in inherited from the basic ‘Form’ class provided by C# programming language. ‘frmMain’ class refers ‘UnitTestSpecReader’ and ‘UnitTestSpecWriter’ classes for reading the unit test specification and generating unit test code, respectively. ‘UnitTestSpecReader’ and ‘UnitTestSpecWriter’ classes refer ‘Utility’ class to access utility functions needed for function operations.

Classes

Classes of this tool can be divided into two sections. One set of classes are mainly for the operational purpose of the tool as the other set of classes are for holding data needed for unit test code generation.

Operational purpose

The classes for operational purpose and brief descriptions of them are as follows.

Table 3-3 Classes for operational purpose

#	Class	Description
1	Form	This is the base class inherited for the creation of the ‘frmMain’ class which is for the UI of this tool.
2	frmMain	This is the class for the UI of this tool. This includes the member variables needed for the unit test code generation process and the functions for event execution.
3	UnitTestSpecReader	This class reads the unit test specification content into the tool. Further, it analyzes the read sheets and feed the read data into the classes to hold structures needed in the unit test code generation process. Apart from that, it does the check for the existence of ample test cases for boundary value checks.
4	UnitTestSpecWriter	This class refers the class objects which hold the information needed to generate unit test code and generate unit test code files.

5	Utility	This class is a static class to hold the utility functions needed for the entire tool. It is referred by both 'UnitTestSpecReader' and 'UnitTestSpecWriter' classes, which drive the operation of the tool.

Data holding purpose

The classes for data holding purpose and brief descriptions of them are as follows.

Table 3-4 Classes for data holding purpose

#	Class	Description
1	UnitTestSpec	Hold the data related to the whole unit test specification file.
2	UnitTestSpecSheet	Hold the data related to a 'No.x' sheet of the unit test specification.
3	UnitTestSpecHeader	Hold the data in the header section of a 'No.x' sheet of the unit test specification.
4	TargetFunction	Hold the data in the target function related sections of a 'No.x' sheet of the unit test specification.
5	GlobalVariable	Hold the data in the global variable related sections of a 'No.x' sheet of the unit test specification.
6	FunctionCall	Hold the data in the function call related sections of a 'No.x' sheet of the unit test specification.
7	UnitTestCase	Hold the unit test case wise data in a 'No.x' sheet of the unit test specification.
8	GlobalVariableValue	Hold the global variable value related data in a unit test case in a 'No.x' sheet of the unit test specification.
9	FunctionCallValue	Hold the function call value related data in a unit test case in a 'No.x' sheet of the unit test specification.

10	FunctionCallReturnValue	Hold the function call return value related data in a unit test case in a 'No.x' sheet of the unit test specification.
11	Parameter	Hold the parameter related data in both the header section and a unit test case (both target function and function call) in a 'No.x' sheet of the unit test specification.

Sequence diagrams

In this tool, there are two major sequences. One for reading the 'No.x' sheet list of the unit test specification and the other for reading the contents of the 'No.x' sheets selected by the user and generating the test code based on the information on those sheets.

Read the sheet list

This sequence is for user selection of the unit test specification and reading the 'No.x' sheets into the unit test code generation tool.

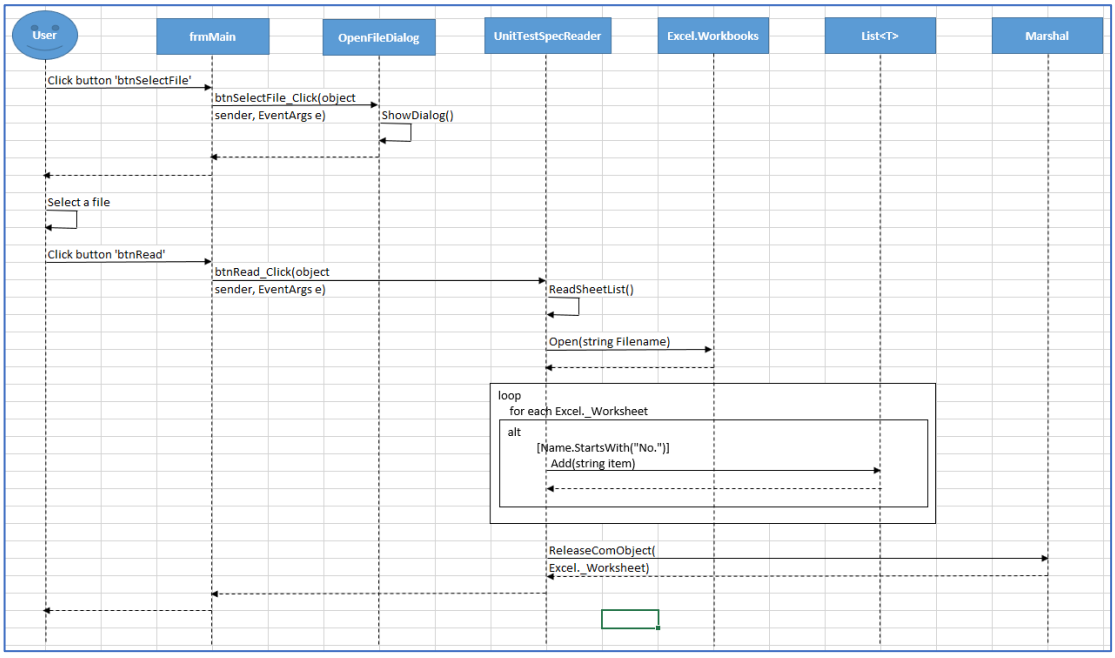


Figure 3-2 Sequence diagram - Read the sheet list

When the user clicks on the ‘Select file’ button, a dialog box to select a unit test specification file is opened. Then, the user selects a unit test specification file. After that, the user clicks on the ‘Read button’. As a result, ‘ReadSheetList()’ function in the ‘UnitTestSpecReader’ class is called. From there, the Excel workbook is opened, the sheet list is obtained and returned. The obtained sheet list is looped in the ReadSheetList() function and filter the ‘No.x’ sheets in it. Then, the Excel workbook and Excel application are closed, and the filtered sheet list is returned to the ‘frmMain’ class. At ‘frmMain’ class filtered sheet list is added to the list to be displayed to the user in the tool UI.

Generate unit test code

This sequence is where user selects which sheets to be kept from the ‘No.x’ sheets loaded to the tool for generating unit test code. Tool reads the ‘No.x’ sheet content and generates the unit test code.

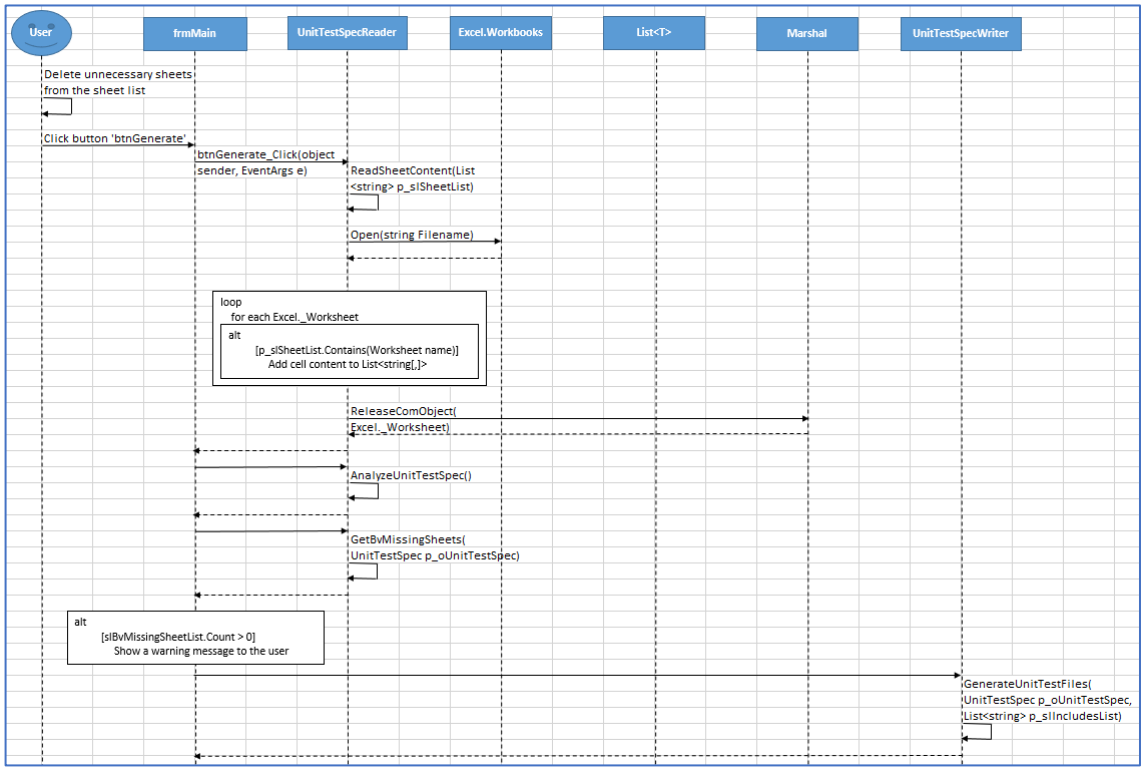


Figure 3-3 Sequence diagram - Generate unit test code

User deletes the sheets that he does not want to generate unit test code and clicks ‘Generate’ button. As a result, ‘ReadSheetContent()’ function in the ‘UnitTestSpecReader’ class is called. From there, the Excel workbook is opened, cell contents are obtained looping the sheets and the Excel workbook is closed. Then, the cell content obtained is analyzed by the ‘AnalyzeUnitTestSpec ()’ function in the ‘UnitTestSpecReader’ class and checks whether the ample amount of unit test cases exist for boundary value checks by ‘GetBvMissingSheets()’ function which is also in the ‘UnitTestSpecReader’ class. If the unit test cases for boundary values are inadequate, a warning message is displayed to the user. Finally, unit test files are generated by the ‘GenerateUnitTestFiles ()’ function in the ‘UnitTestSpecWriter’ class.

3.3 Unit test code

For the unit test code generated from the tool, a format is defined. By having such a format, it improves test code’s readability. Apart from that, it is easier and efficient to have such a format when the tool is developed, as it becomes easier and possible to pay more attention, section wise.

3.3.1 Format

Here in this section, the definition of the format for the unit test code for a source code file is explained. A unit test code output from the unit test code generator tool will be as follows.

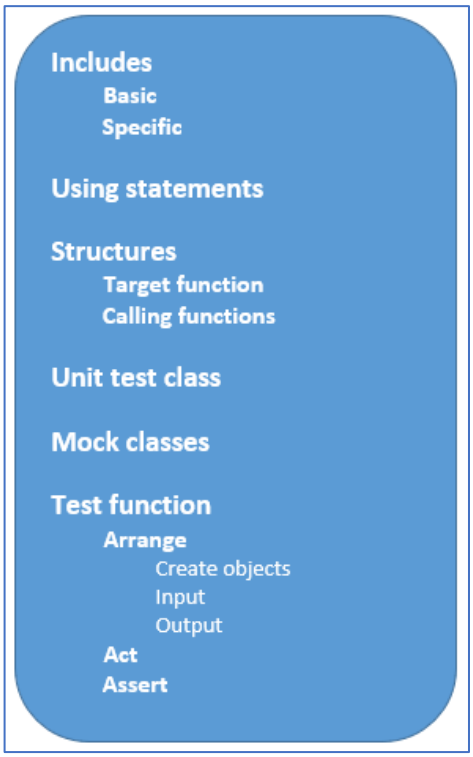


Figure 3-4 Unit test code format

The items in the above format are briefly described below.

Includes: The include statements which include necessary header files and source files for referring the base functions.

Basic: The include statements which are basic in the aspect of Google Test unit test code. These are common for all the unit test files generated so that, those are fixed for all the unit test code files generated.

Specific: The include statements which are relevant to the specific unit test code file only. These change from base source file to source file.

Using statements: Using statements written here include both basic using statements which are relevant to Google Test and ones relevant to specific includes done for the unit test code file.

Structures: A structure is defined to hold the components related to the functions which are to be used in the unit test code.

Target function: This structure is to hold the components of the function to be tested. The components in this structure are as follows.

- Parameter – The parameters of the function to be tested.
- Return value – The return value of the function to be tested.

Calling function: This structure is to hold the components of a function that is called in the function to be tested. The components in this structure are as follows.

- Parameter – The parameters of a calling function.
- Return value – The return value of a calling function.
- Call count – Number of times a certain calling function is called in the target function.

Unit test class: This is the class for having logics which are needed to execute before and after each test function.

Mock classes: This class includes the mock calls to the calling functions in the target function for unit testing. The function definitions in this class hook the function calls relevant to those functions so that, various measurements can be done in the test functions.

Test function: This is the function where a certain unit test case is executed in the unit test code. The number of test functions created should be equal to the number of unit test cases in the relevant 'No.x' sheet in the unit test specification. For improving the unit test code's readability, the body of a test function is virtually divided in to three sections as follows.

Arrange: This section is to assign values to the variables according to the values in the unit test specification for the relevant unit test case. For improving the unit test code readability, this section is also divided in to three sub sections as follows.

Create objects: In this section, the class objects and mock class objects are created to be used in the same test function.

Input: In this section, the input values for the variables in the relevant unit test case are assigned. The values considered for this assignment are the values in the 'Input' section of the relevant 'No.x' sheet. There, three types of assignments are done as follows.

- Target function parameters – Assign input values for the target function parameters.
- Global variables – Assign input values for global variables used in the target function.
- Calling function return values – Assign the expected return values from the calling functions in the target function.

Output: In this section, the output values for the variables in the relevant unit test case are assigned. The values considered for this assignment are the values in the 'Output' section of the relevant 'No.x' sheet. There, four types of assignments are done as follows.

- Global variables – Assign output values for the global variables used in the target function.
- Calling function call count – Assign how many times a certain calling function is called within the target function.
- Calling function parameters – Assign the values for parameters in the calling function.
- Target function return value – Assign return value expected from calling target function.

Act: This section is to execute the target function for unit testing. If the target function returns a value after executing, it is also assigned to the relevant variable in target function structure.

Assert: This section is to execute the assertions, meaning that, to check whether the values obtained from executing the target functions are the same as the expected values. Here in this unit test code generation tool, two types of assertions are considered.

- Global variable assertions – Check whether the values obtained for global variables from executing the target functions are the same as the expected values. The number of such assertions are same as the number of global variables used in target function.
- Target function return value assertion – Check whether the return value obtained from executing the target function is the same as the expected return value.

3.4 Schema definition

The schema defined here is to reduce possible misses of unit test cases due to human errors. The schema should be included to check the unit test specification test cases whether those cover the whole function, whether the test case inputs are valid, whether checks on conditions have properly been done (e.g. boundary value checks), etc.

Though the schema should be defined comprehensively and implemented in the unit test code generator tool completely, here is this research work, one aspect of schema definition has been done as a POC. It is, checking whether the unit test specification is included the ample amount of unit test cases for each condition check that is in the target function, in the aspect of boundary value checks.

The definition used here is, for each condition check, there should be three additional unit test cases for boundary value checks. Those are;

- A unit test case with boundary value + 1
- A unit test case with boundary value = condition value
- A unit test case with boundary value – 1

If there are not ample unit test cases for boundary value checks exist in the unit test specification, a warning message should be displayed to the user informing the target function list where boundary value checks are not enough.

3.5 Unit testing procedure

In this section, the procedure to carry out unit testing with the usage of the tools introduced in this research work is explained.

- Create the unit test specification using the unit test specification format introduced in this research work. Here, following spreadsheet sheets need to be filled in that format.
    - Cover
    - Update history
    - Function list
    - Unit test results
    - No.x sheets
- Review the created unit test specification, carry out the review fixes and finalize.
- Read the finalized unit test specification into the Google Test Unit Test Code Generator tool, which is the unit test code generator tool introduced in this research work.
- Remove the No.x sheets that are not needed to generate unit test code, add the 'includes' statements needed to be written in the unit test code and generate unit test code files.
- Add the generated unit test code files to Google Test unit testing execution project and do modifications to the unit test code if there is any, such as missing 'includes' statements.
- Execute unit testing using Google Test unit testing framework.
- Obtain the code coverage got by the unit test execution using a code coverage tool.
- Create the unit test report from the unit test specification, completing the remaining following sheet inputs.
    - No.x sheets (Test result, test date and remarks if any)
    - Bug list
    - Coverage results

The procedure can be depicted as in *Figure 3-5* below.

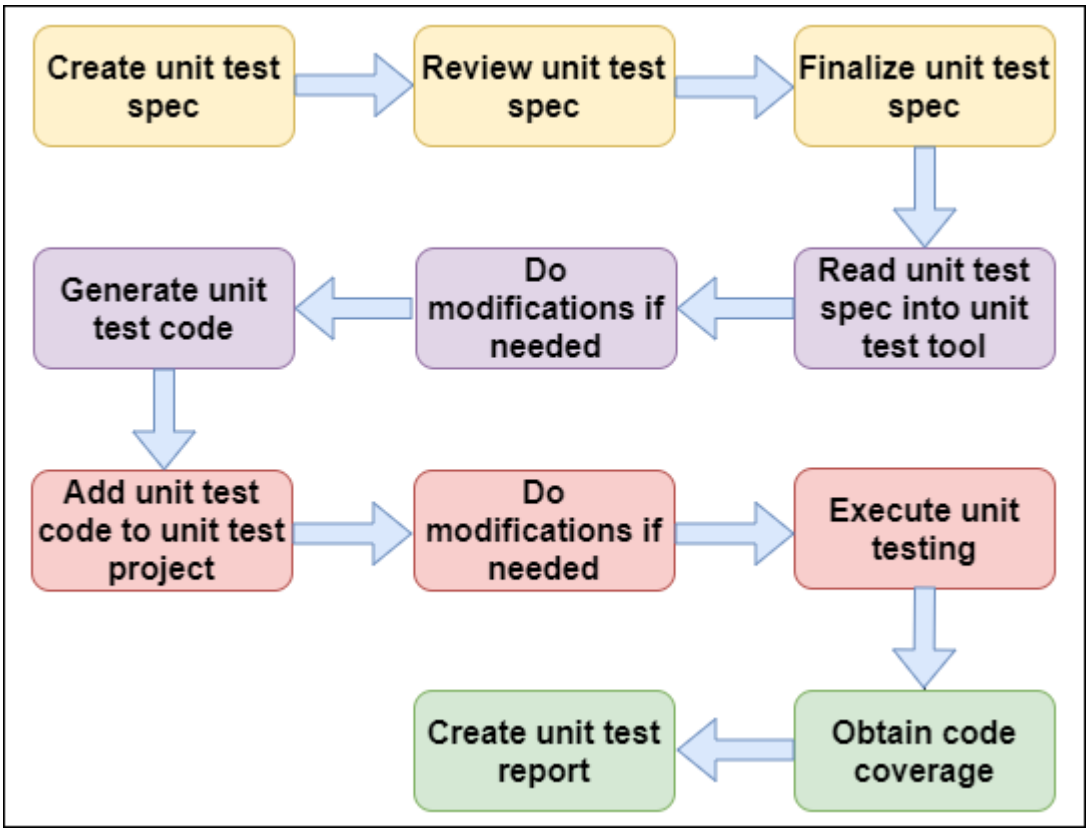


Figure 3-5 Unit testing procedure

4 IMPLEMENTATION

Regarding the implementation related to this research work, there are three main aspects to be explained as follows.

1. **Unit test specification**

   This section explains how the provided facilities in Microsoft Excel have been used to create the desired unit test specification.

2. **Google Test Unit Test Code Generator tool**

   Here, the software development project related details and the class wise explanation of the implementation methodologies and the algorithms used have been provided.

3. **Google Test unit test project**

   The unit test execution project related details have been explained in this section.

4.1 Unit test specification

As the unit test specification must be created by a team member, it takes some time incurring a cost to the project. When creating the unit test specification format, it has been considered to minimize the inputs to be made by the unit test specification creator and the unit tester so that, some cost saving can be achieved. On the other hand, it improves the accuracy of the unit test specification as input errors are minimized. To achieve this, Microsoft Excel formulae have been used efficiently in this process.

Here in this section, the implementation aspect of the unit test specification is described for the items which needs explanation in each sheet. For this, snapshots of a sample unit test specification are used with the explanations.

4.1.1 Cover sheet

The ‘Cover’ sheet of a completed unit test specification will be as in *Figure 4-1* below.

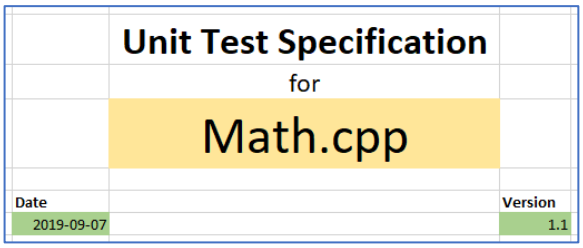


Figure 4-1 'Cover' sheet of unit test specification

Values for both ‘Date’ and ‘Version’ in the ‘Cover’ sheet need to be the latest ones. The values for these are not to be inserted manually. The values are retrieved from the special cells arranged in the ‘UpdateHistory’ sheet and the relevant cells in this sheet are automatically updated based on those values.

Further, to display the values here in the correct formats, the cell formatting has been carried out as in *Table 4-1* below.

Table 4-1 Cover sheet cell formatting

#	Item	Category	Remarks
1	Date	Date	Type: YYYY-MM-dd (2012-03-14)
2	Version	Number	Decimal places: 1

To implement this, Microsoft Excel’s ‘Format Cells’ feature has been used.

4.1.2 UpdateHistory sheet

The ‘UpdateHistory’ sheet of a completed unit test specification will be as in *Figure 4-2* below.

No.	Date	Version	Changes	Changed By
1	2019-08-23	1.0	Create new	Nimal Ranawaka
2	2019-09-07	1.1	Add 'No.2' sheet, Update 'No.1' sheet	Kasun Batagala
3				
4				
5				
6				
7				
Add new rows above this row				

Figure 4-2 'UpdateHistory' sheet of unit test specification

Values for ‘No.’ column are inserted automatically by using an Excel formula from the 2nd row onwards (Value of the above cell + 1). By having this, if the unit test specification creator copies a row from the 2nd row onwards and inserted it to the table as a new row, the value of the ‘No.’ column is automatically inserted in the correct way.

Though, the values should be inserted (apart from the ‘No.’ column) manually, to display the values here in the correct formats, the cell formatting has been carried out for ‘Date’ and ‘Version’ column values as in ‘*Table 4-1 Cover sheet cell formatting*’.

The needed ‘Date’ and ‘Version’ values to be automatically inserted in the ‘Cover’ sheet are arranged in this ‘UpdateHistory’ sheet. Here, a table (which is hidden in the unit test specification) is created as in *Figure 4-3* below.

Last Date	Last Version
2019-09-07	1.1

Figure 4-3 'UpdateHistory' sheet - Last date & last version

In this table also, the cell formatting has been carried out for ‘Last Date’ and ‘Last Version’ column values as in ‘*Table 4-1 Cover sheet cell formatting*’. The last row values of the ‘Date’ and ‘Version’ in the ‘UpdateHistory’ table are retrieved to the above table using the ‘LOOKUP’ formula in Microsoft Excel [28].

These cells which contain last date and last version values are refereed by the ‘Cover’ sheet’s date and version cells to display the latest date and latest version.

4.1.3 FunctionList sheet

A ‘FunctionList’ sheet of a completed unit test specification will be as in *Figure 4-4* below.

No.	Sheet Name	Function Name
1	No.1	getSumOfSquares
2	No.2	getSquare
3		
4		
5		
6		
7		
Add new rows above this row		

Figure 4-4 'FunctionList' sheet of unit test specification

Here, the values for 'No.' column are inserted automatically.

4.1.4 UnitTestResults sheet

A 'UnitTestResults' sheet of a completed unit test specification will be as in *Figure 4-5* below.

No.	Sheet Name	Link	Function Name	Total	OK	NG	Not Tested	Percentage
1	No.1	Go to sheet	getSumOfSquares	4	4	0	0	100.00%
2	No.2	Go to sheet	getSquare	2	0	2	0	0.00%
Add new rows above this row								
			TOTAL	6	4	2	0	66.67%

Figure 4-5 'UnitTestResults' sheet of unit test specification

Here, the values for 'No.' column are inserted automatically.

Though the values for 'Sheet Name' column are to be inserted manually by the unit test specification creator, instead of typing, the needed value is to be selected from a drop-down list which includes the list of the sheet names existing in the 'FunctionList' sheet. For the implementation of this, 'Data Validation' feature of Microsoft Excel has been used.

Apart from that, the sheet name column cells will be colored in red, if there are NG test cases in the relevant unit test specification. To implement this feature, 'Conditional Formatting' feature of Microsoft Excel has been used [29].

For the implementation of the values in the 'Link' column, a Microsoft Excel formula has been used. That is 'HYPERLINK' formula. The main challenge in implementing it was that the formula had to get the value in the relevant 'Sheet Name' cell to create the

hyperlink to the target unit test specification sheet. To achieve that, a special formatting has been used [30] [31].

The values for the columns; 'Function Name, Total, OK, NG, Not Tested' are automatically inserted by using another Microsoft Excel formula called 'INDIRECT' [32].

To display the percentage value in the correct format, the cell formatting has been carried out as in *Table 4-2* below.

Table 4-2 UnitTestResult sheet cell formatting

#	Item	Category	Remarks
1	Percentage	Percentage	Decimal places: 2

With the above cell formatting, the formula to calculate the percentage should only be 'No. of OK unit test cases / No. of total unit test cases'.

The values for the columns; 'TOTAL Total, TOTAL OK, TOTAL NG, TOTAL Not Tested' are automatically inserted by using the Microsoft Excel summation formula 'SUM' for a range of cells. And, to avoid missing out newly added rows in the 'UnitTestResults' sheet table for the summation, additional one cell is also included in the target value range for this summation as in *Figure 4-6* below.

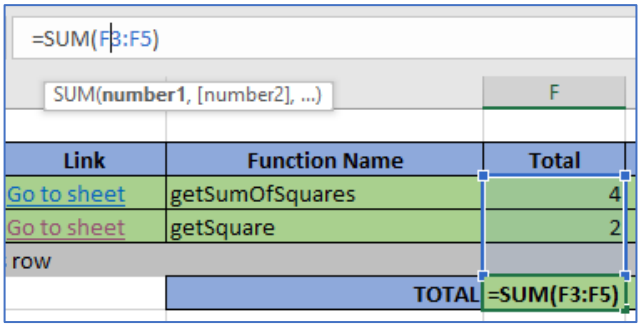


Figure 4-6 Microsoft Excel - Include additional one cell to the range

TOTAL percentage also has the cell formatting as described in '*Table 4-2 UnitTestResult sheet cell formatting*'. The formula to calculate the percentage should be only 'Total no. of OK unit test cases / Total no. of total unit test cases'.

4.1.5 CoverageResults sheet

An 'Observation' table in the 'CoverageResults' sheet of a completed unit test specification will be as in *Figure 4-7* below.

Observation

No.	Function Name	Reason	Lines	Functions	Branches
1	getSumOfSquares	'new' statement cannot be hooked.	2	0	4
2	getSumOfSquares	Log statement is not covered due to a problem in log functionality.	0	0	1
3	getSquare	Log statement is not covered due to a problem in log functionality.	0	0	1
4					
5					
6					
7					
Add new rows above this row					

Figure 4-7 'Observation' table - 'CoverResults' sheet of unit test specification

Here, the values for 'No.' column are inserted automatically.

A 'Summary' table in the 'CoverageResults' sheet of a completed unit test specification will be as in *Figure 4-8* below.

Summary

	Lines	Functions	Branches
Hits	95	2	139
Observation	2	0	6
Total	97	2	145
Total (Should be)	97	2	145
Percentage	100%	100%	100%

Figure 4-8 'Summary' table - 'CoverResults' sheet of unit test specification

The 'Total' for each coverage category (lines, functions, and branches) are obtained by summing up the values in the above two lines (i.e. 'Hits' and 'Observation') in the 'Summary' table.

'Percentage' for each coverage category also has the cell formatting as described in '*Table 4-2 UnitTestResult sheet cell formatting*'. The formula to calculate the percentage should be only 'Total / Total (Should be)'.

4.1.6 BugList sheet

A 'BugList' sheet of a completed unit test specification will be as in *Figure 4-9* below.

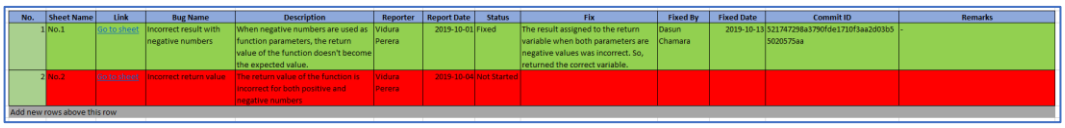


Figure 4-9 'BugList' sheet of unit test specification

Here, the values for 'No.' column are inserted automatically.

As the formats of the values for the 'Report Date' and 'Fixed Date' columns, the cell formatting has been used as mentioned in '*Table 4-1 Cover sheet cell formatting*'.

And, the color changing of the cells based on the status is implemented by using 'Conditional Formatting' feature of Microsoft Excel [29]. The conditions used are as in *Figure 4-10* below.

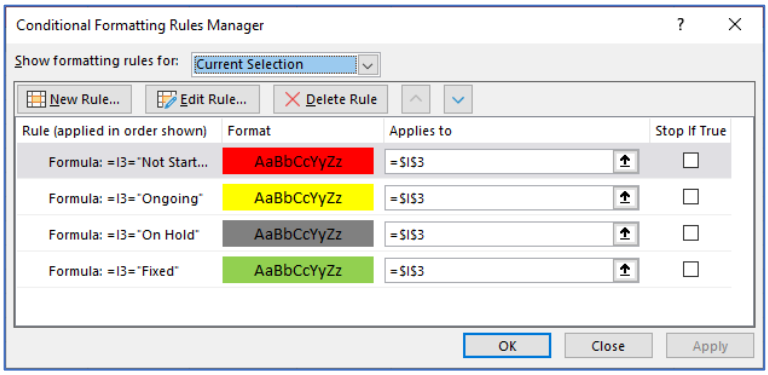
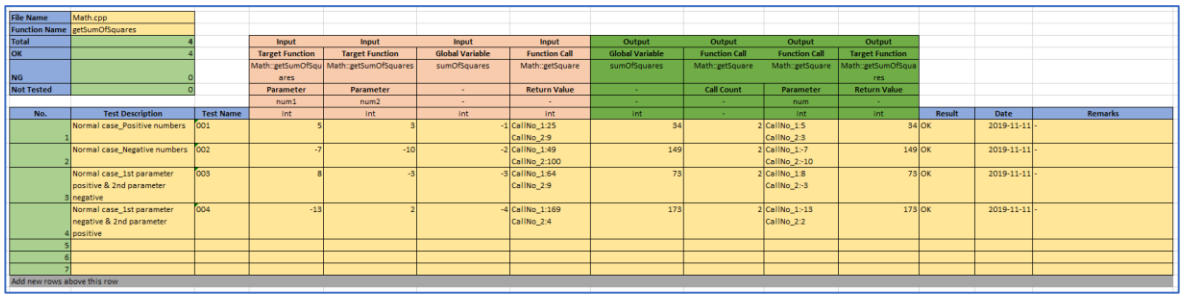


Figure 4-10 'Conditional Formatting' in 'BugList' sheet

4.1.7 No.x sheet

A 'No.x' sheet of a completed unit test specification will be as in *Figure 4-11* below.



File Name	Math.cpp												
Function Name	getSumOfSquares												
Total	4		Input	Input	Input	Input	Output	Output	Output	Output			
OK	4		Target Function	Target Function	Global Variable	Function Call	Global Variable	Function Call	Function Call	Target Function			
NG	0		Math::getSumOfSquares	Math::getSumOfSquares	sumOfSquares	Math::getSquare	sumOfSquares	Math::getSquare	Math::getSquare	Math::getSumOfSquares			
Not Tested	0		Parameter	Parameter	-	Return Value	-	Call Count	Parameter	Return Value			
			num1	num2	-	-	-	-	num	-			
No.	Test Description	Test Name	int	int	int	int	int	-	int	int	Result	Date	Remarks
1	Normal case_Positive numbers	001	5	3	-1	CallNo_1:25 CallNo_2:9	34	2	CallNo_1:5 CallNo_2:3	34	OK	2019-11-11	-
2	Normal case_Negative numbers	002	-7	-10	-2	CallNo_1:49 CallNo_2:100	149	2	CallNo_1:-7 CallNo_2:-10	149	OK	2019-11-11	-
3	Normal case_1st parameter positive & 2nd parameter negative	003	8	-3	-3	CallNo_1:64 CallNo_2:9	73	2	CallNo_1:8 CallNo_2:-3	73	OK	2019-11-11	-
4	Normal case_1st parameter negative & 2nd parameter positive	004	-13	2	-4	CallNo_1:169 CallNo_2:4	173	2	CallNo_1:-13 CallNo_2:2	173	OK	2019-11-11	-
5													
6													
7													
Add new rows above this row													

Figure 4-11 'No.x' sheet of unit test specification

Here, the values for ‘No.’ column are inserted automatically.

As the formats of the values for the ‘Test Name’ and ‘Date’ columns, the cell formatting has been used. For ‘Test Name’ column, ‘Text’ category has been selected. For, ‘Date’ column, the values have been selected as mentioned in ‘*Table 4-1 Cover sheet cell formatting*’.

To calculate Total, OK and NG unit test counts, the ‘COUNTIF’ formula of Microsoft Excel [33] has been used.

And for calculating ‘Not Tested’ unit test case count, the Total, OK and NG unit test counts are used. The formula is; ‘Not Tested unit test count = Total unit test count – (OK unit test count + NG unit test count)’.

4.1.8 Check for boundary values

Unit test cases should be written with the following schema definitions to get the warnings when boundary value checks are not enough.

In the ‘Test Description’ of the unit test case for the condition check, at the end, ‘_Condition’ string must be added. It is to identify the condition checks by the unit test case generator tool. Then, in the ‘Test Description’ of the unit test case for the boundary value checks, at the end, ‘boundary value’ string must be included after ‘_’. An example is as follows when the condition check is ‘if (num >= 75)’.

File Name	Math.cpp							
Function Name	isPassed							
Total	5		Input	Output				
OK	0		Target Function	Target Function				
NG	0		Math::isPassed	Math::isPassed				
Not Tested	5		Parameter	Return Value				
			num	-				
No.	Test Description	Test Name	int	int	Result	Date	Remarks	
1	Normal case_Passed_Condition	001	80	TRUE				
2	Normal case_Passed_Boundary value high	002	76	TRUE				
3	Normal case_Passed_Boundary value equal	003	75	TRUE				
4	Normal case_Failed_Boundary value low	004	74	FALSE				
5	Normal case_Failed	005	60	FALSE				
Add new rows above this row								

Figure 4-12 Unit test cases for boundary value checks

4.2 Google Test Unit Test Code Generator tool

The IDE used to create this tool is Visual Studio 2017. It is selected because of the support it gives when creating GUI (Graphical User Interface) applications. And, the 2017 version is selected because it is a newer stable version of Visual Studio that has been released recently, and it is better to create a new C# tool with a newer IDE. Further, as the edition of it, the Community edition has been chosen. The main reason was that, it was a free edition. Also, the features offered by this edition are enough for the tool creation. Most importantly, it gives the chance for the tool to get improved, as future developers can access the code base and do advance improvements, free of charge.

The tool was named as 'Google Test Unit Test Code Generator'. As the UI of this tool is a simple one, decided to create a Visual Studio project with the project type 'Windows Forms App (.NET Framework)'.

Further, the .NET Framework used was 4.7.2, which is a newer and stable release of it, at the time of this tool development. Also, this .NET Framework supports for Windows 10 OS (Operating System), which is the development environment and the target running environment of this tool [34].

The programming language used for creating this tool was 'C#'. It is because C# is one of the easiest languages to implement graphical routines, which are needed in this kind of GUI applications.

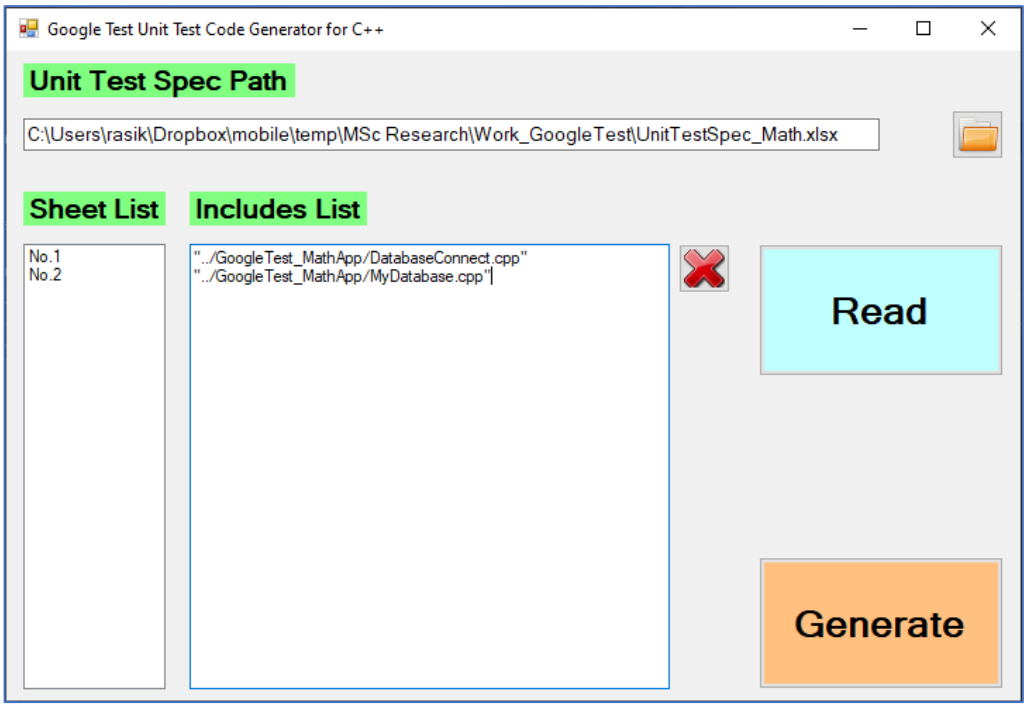


Figure 4-13 UI of GoogleTestUnitTestCodeGenerator tool

The classes at the implementation level and their functionalities are as follows.

4.2.1 frmMain class

This is the class related to the only UI in this tool. This class is inherited from the Form class, which comes as a default class with C# language. The inheritance has also been automatically generated when creating a new form in C#, Visual Studio.

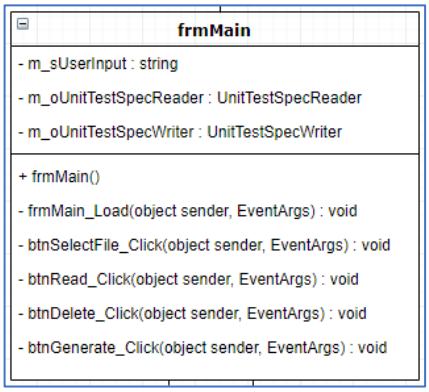


Figure 4-14 frmMain class

The main functionalities this class provides can be listed as follows.

- Preparation of the tool's main UI to be displayed to the user.

- Selecting the unit test specification file (As the unit test specification file is a Microsoft Excel file, the files displayed in the open file dialog box are restricted only for Excel files [35].)

- Obtaining the list of sheets which are relevant for unit test code generation (E.g. No.1, No.2, etc.) from the unit test specification. As this process could take a little while, following precautions have been taken at the development.
    - Disabling the **Read** Button and **Generate** Button while the process is being executed to avoid unnecessary user interaction.
    - Changing the mouse cursor while the process is being executed to indicate the user that the tool is busy executing a process [36].

- Deleting loaded unit test specification sheet names.

- Executing unit test code generation. As this process could take a little while, following precautions have been taken at the development.
    - Disabling the **Generate** Button while the process is being executed to avoid unnecessary user interaction.
    - Changing the mouse cursor while the process is being executed to indicate the user that the tool is busy executing a process [36].

  Further, the includes list user has inserted is read from the **Includes List** Textbox. The unit test code generation happens in two major steps as follows.

    - The unit test specification is analyzed, and the analyzed result is assigned to a **UnitTestSpec** (refer *Table 3-4 Classes for data holding purpose*) object.

- Generate unit test case files using both **UnitTestSpec** object and the includes list.

4.2.2 UnitTestSpecReader class

This is the class for reading unit test specification in this tool.

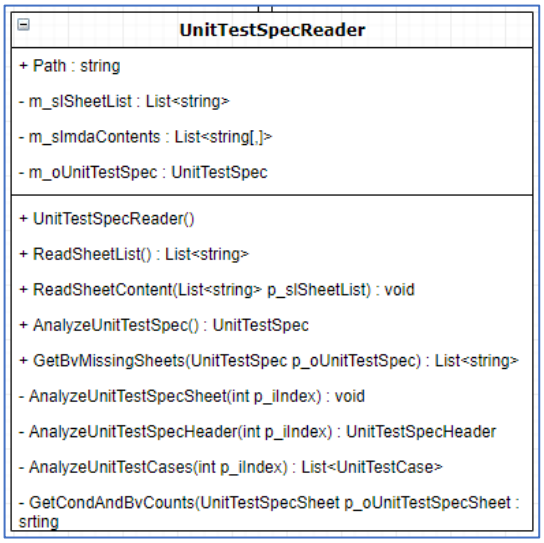


Figure 4-15 UnitTestSpecReader class

The main functionalities this class provides can be listed as follows.

- Opening unit test specification in the program to read the sheet names. Here, a proper closing of this file is needed. Otherwise, this tool could keep a process to hold the file in the opened state. The closing of the unit test specification has been carried out in this tool as in *Figure 4-16* below [37].

```
public List<string> ReadSheetList()
{
    Excel.Application oExcelApplication = new Excel.Application();
    Excel.Workbook oExcelWrokbook = oExcelApplication.Workbooks.Open(m_sPath);
    m_slSheetList = new List<string>();

    foreach (Excel._Worksheet oExcelWorksheet in oExcelWrokbook.Sheets)
    {
        m_slSheetList.Add(oExcelWorksheet.Name);
    }

    GC.Collect();
    GC.WaitForPendingFinalizers();
    Marshal.ReleaseComObject(oExcelWrokbook.Sheets);
    oExcelWrokbook.Close();
    Marshal.ReleaseComObject(oExcelWrokbook);
    oExcelApplication.Quit();
    Marshal.ReleaseComObject(oExcelApplication);

    return m_slSheetList;
}
```

Figure 4-16 'ReadSheetList' function in 'UnitTestSpecReader' class

- Reading the contents of the unit test sheets into the unit test code generator tool. Here, it is important to identify the cell range in unit test specification sheet to read the contents from. Otherwise, the program will loop through the entire sheet, grading down the performance. To achieve this, the code snippet in *Figure 4-17* below has been used.

```
Excel.Range oExcelRange = oExcelWorksheet.UsedRange;

int iRowCount = oExcelRange.Rows.Count;
int iColumnCount = oExcelRange.Columns.Count;
```

Figure 4-17 Obtain Microsoft Excel sheet's used range

- Obtaining the unit test specification sheets which do not include enough boundary value test cases.
- Analyzing unit test specification content and assigning those into relevant variables in the program. To avoid unnecessary processing, iterating columns in the program is done only if unit test case details are available. It is decided by checking whether texts exist in 2nd and 3rd columns. The logic related is as in *Figure 4-18* below.

```
List<UnitTestCase> oUnitTestCaseList = new List<UnitTestCase>();

int iRowCount = m_slmdaContents[p_iIndex].GetLength(0);
int iColumnCount = m_slmdaContents[p_iIndex].GetLength(1);

for (int i = 9; i < iRowCount; i++)
{
    if (!m_slmdaContents[p_iIndex][i, 1].Equals("") &&
        !m_slmdaContents[p_iIndex][i, 1].Equals("Add new rows above this row"))
    {

        UnitTestCase oUnitTestCase = new UnitTestCase();
        oUnitTestCase.TargetFunctionParameterList = new List<Parameter>();

        for (int j = 1; j < iColumnCount; j++)
        {
```

Figure 4-18 Iteration used in 'AnalyzeUnitTestCases' function

4.2.3 UnitTestSpecWriter class

This is the class for creating the unit test code based on the read data at the **UnitTestSpecReader** Class.

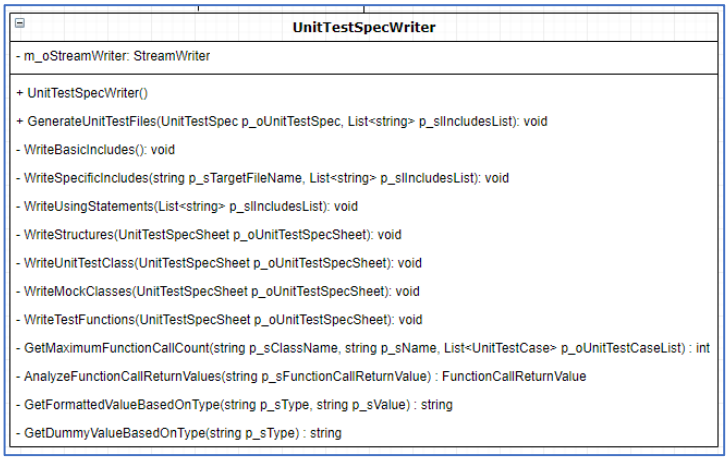


Figure 4-19 UnitTestSpecWriter class

The main functionalities this class provides can be listed as follows.

- Creating the location to save the unit test code files.
- Creating unit test code files.
- Writing the content in the unit test code files. The content is as follows.
    - Basic includes
    - Specific includes

- Using statements
- Structures to hold target function and calling functions
- Unit test class
- Mock classes

  If there are function calls exist in the target function, separate mock classes are written for each of those function call. Here, **Google Mock** is used, and the code is according to its specifications (Refer *1.3.4 Google Mock*), within the mock classes. The code snippet for generating test code for mocking a function call is as in *Figure 4-20* below.

```
p_oStreamWriter.Write("\tMOCK_METHOD" + iMethodIndex + "(" +
                oFunctionCall.Name + ", " + oFunctionCall.ReturnType + "(");

int iParameterCount = 0;
if (oFunctionCall.ParameterList != null)
{
    foreach (Parameter oParameter in oFunctionCall.ParameterList)
    {
        p_oStreamWriter.Write(oParameter.Type + " " + oParameter.Name);

        if (oFunctionCall.ParameterList.Count > 1 && iParameterCount + 1 < oFunctionCall.ParameterList.Count)
        {
            p_oStreamWriter.Write(", ");
        }

        iParameterCount++;
    }
}
p_oStreamWriter.WriteLine("));");
```

Figure 4-20 Mocking function calls in code

A sample output of the Mock classes section is as in *Figure 4-21* below. Highlighted area is the result of the above code snippet.

```
/* Mock classes */
class MockMath : public Math {
public:
    MOCK_METHOD1(getSquare, int(int num));
};
```

Figure 4-21 Mock classes sample output

- Test functions

- Calculating maximum call counts for calling functions.
- Obtaining return values for function calls, based on the call iteration.
- Formatting values obtained from unit test specification suiting to their data types.

4.2.4 Utility class

This is the class for holding the static utility functions needed in this tool.

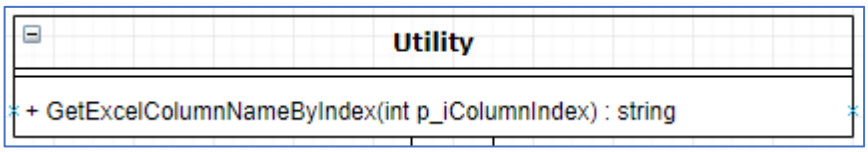


Figure 4-22 Utility class

Though the Microsoft Excel's row name is directly related to the index, column name is different. It has been created with English letters (i.e. A… AA… AAA…). As a result, displaying an error of data in a cell with the column index (instead of column name) is not user friendly to the user. This class converts column indices to column names and return. The logic of the function is as in *Figure 4-23* below.

```
private const int NO_OF_LETTERS_IN_ALPHABET = 26;
private const int ASCII_VALUE_OF_A = 65;

public static string GetExcelColumnNameByIndex(int p_iColumnIndex)
{
    string sColumnName = string.Empty;
    int iDividend = p_iColumnIndex;
    int iModulous = 0;

    while (iDividend > 0)
    {
        iModulous = (iDividend - 1) % NO_OF_LETTERS_IN_ALPHABET;
        sColumnName = Convert.ToChar(ASCII_VALUE_OF_A + iModulous).ToString() + sColumnName;
        iDividend = (int)((iDividend - iModulous) / NO_OF_LETTERS_IN_ALPHABET);
    }

    return sColumnName;
}
```

Figure 4-23 'GetExcelColumnByIndex' function logic

4.2.5 Other classes

The classes other than the ones explained above are for data holding purpose. They are mainly with properties. Those classes hold data related to following items.

- Unit test specification
- Unit test specification sheet
- Unit test specification sheet header

- Unit test case
- Target function
- Parameter
- Global variable
- Global variable value
- Function call
- Function call value
- Function call return value

4.3 Google Test unit test project

Microsoft has embedded the facility to create Google Test unit test projects inside Visual Studio 2017 Community Edition. Therefore, it has become easier to run Google Test unit test code from this IDE itself.

As discussed in *1.3 Unit testing with mock objects* , using Google Mock makes it possible to mock the functions used in the function which is targeted to unit test. To utilize this in Visual Studio 2017, it is needed to install a NuGet package. In this research work, NuGet package gmock v 1.7.0 has been used [38].

5 EVALUATION

As the evaluation of this research work, the unit test specification format and the unit test code generator tool were used in five C++ language-based projects in the researcher's company. Those were used for the unit testing of six functions from each of those projects by the related software developers and for obtaining their feedback. Additionally, the expert judgement of the generated unit test code has also been obtained. Here in this section, introductions for the selected projects have been given. A unit test targeted function from each project has been selected for the detailed description. A description of that function, unit test specification created, and the unit test code generated for each of those projects have been mentioned. The other five target functions and the expert judgement for the completeness of the unit test code generated are presented under each project. The feedback given by each software developer who carried out the unit testing has been summarized at the end.

5.1 Project 1 – LNBTI Door Access Control

5.1.1 Description

This project has been done for a private university in Sri Lanka for their door access control purposes. Door access controlling have been used at each critical place in the university such as; the main entrance, lecture rooms, library, etc. Further, the access for each university member can be configured and modified based on the updated privileges he/she has and based on the date and time.

The software related to this project have been installed in a Raspberry Pi which has a Linux based OS. A separate Raspberry Pi has been used for each door. Additionally, for central monitoring and controlling, a central server has also been installed. MySQL DBs have been installed in both Raspberry Pi and the central server. DBs in each Raspberry Pi synchronize with the DB in the central server.

Door access controlling in this project has been done using NFC (Near Field Communication) cards. Specifically, NFC technology by Sony (Sony FeliCa cards) has been used for this project.

5.1.2 Target function

This is one of the core functions of the project as this function opens the targeted door for the authorized university members after checking the permission for his/her NFC card, when a card tapping is detected on the card reader.

In the function, it performs a door check for checking whether the targeted door is registered in the system and a card check for checking whether the targeted card is registered in the system and issued to a university member, before considering opening the door based on the various access groups. Only if those checks passed, the function moves forward to check whether the targeted card has the permission to open the door.

An image of the target function is as in *Figure 5-1* below.

```
/**
Name            :   CheckCardPermission()
Summary         :   Opens the door for authorized users by detecting card taps on the reader
Parameter       :   None
Return value    :   None
Created         :   2017/07/19 priyantha
*/
int CheckCardPermission()
{
    ComLog *logObj = getLogObject();

    Controller doorController; //creating object for door controlling
    char *host = getIpAddress();

    unsigned char cardID[CARD_ID_LEN];
    long _ret = DCM_S_RESULT_NG;

    DB_Access dbCheckDoor;
    _ret = dbCheckDoor.db_checkdoor(host); // Check whether the door is registered
    if(  _ret == DCM_S_RESULT_NG ){
        logObj->logError(__LINE__, __FILENAME__, __FUNCTION__, "Door IP %s is not registered. Please register the door..", host);
        return DCM_S_RESULT_NG;
    }
    else if( _ret == DCM_E_DB_ERROR ){
        return DCM_E_DB_ERROR;
    }

    logObj->logTrace(__LINE__, __FILENAME__, __FUNCTION__, "START : readCardWithoutSAM()");
    _ret = readCardWithoutSAM(cardID); //Get card ID from NFC card
    logObj->logTrace(__LINE__, __FILENAME__, __FUNCTION__, "END : readCardWithoutSAM()");

    if(_ret != SCARD_S_SUCCESS){
        logObj->logError(__LINE__, __FILENAME__, __FUNCTION__, "ERROR : readCardWithoutSAM() failed!!!");
        return _ret;
    }
    logObj->logDebug(__LINE__, __FILENAME__, __FUNCTION__, "END : readCardWithoutSAM() success - card ID: <%s>", cardID);
```

Figure 5-1 Project 1 - LNBTI Door Access Control - Target function

5.1.3 Unit test specification

As the target function has several different function calls included, the unit test specification has become wider. Specially, different types of log functions have been used in the target function and each log function has several parameters.

On the other hand, as it has several checks performed, there are several checks to be performed in unit testing also. Every access group has been checked separately for opening the door in target function. As a result, the number of unit test cases have also been increased for the target function.

An image of the unit test specification is as in *Figure 5-2* below.

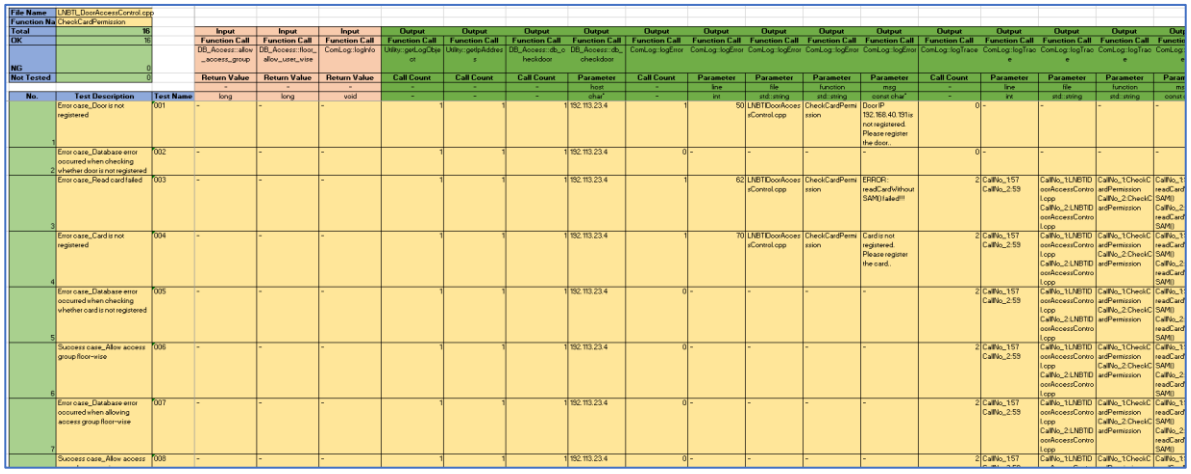


Figure 5-2 Project 1 - LNBTI Door Access Control - Unit test specification

5.1.4 Unit test code

The unit test code generated for the target function is also lengthier as the other function usage and the number of checks are higher in it. As the result, in the unit test code, number of structures, number of mock methods and of course number of test functions have been increased. Further, multiple calls of the same functions are checked in several unit test cases in this unit test code.

An image of the unit test code is as in *Figure 5-3* below.

```
    MOCK_METHOD0(getLogObject, ComLog*());
    MOCK_METHOD0(getIpAddress, char*());
    MOCK_METHOD1(readCardWithoutSAM, long(unsigned char* cardID));
};

class MockDB_Access : public DB_Access {
public:
    MOCK_METHOD1(db_checkdoor, long(char* host));
    MOCK_METHOD1(db_not_issued_card, long(unsigned char* host));
    MOCK_METHOD1(allow_access_group_floor_wise, long(unsigned char* host));
    MOCK_METHOD1(allow_access_group_bm_user_wise, long(unsigned char* host));
    MOCK_METHOD1(door_allow_user_wise, long(unsigned char* host));
    MOCK_METHOD1(allow_access_group, long(unsigned char* host));
    MOCK_METHOD1(floor_allow_user_wise, long(unsigned char* host));
};

class MockComLog : public ComLog {
public:
    MOCK_METHOD4(logError, void(int line, std::string file, std::string function, const char* msg));
    MOCK_METHOD4(logTrace, void(int line, std::string file, std::string function, const char* msg));
    MOCK_METHOD4(logDebug, void(int line, std::string file, std::string function, const char* msg));
    MOCK_METHOD4(logInfo, void(int line, std::string file, std::string function, const char* msg));
};

class MockController : public Controller {
public:
    MOCK_METHOD0(openDoor, void());
};

/*
    Error case_Door is not registered
*/
TEST_F(UnitTest_CheckCardPermission, 001) {

    /* Arrange */

    // Create objects
    MockUtility mockUtility;
```

Figure 5-3 Project 1 - LNBTI Door Access Control - Unit test code

5.1.5 Target functions and expert judgement

The expert judgement for the completeness of the unit test code for the above target function and five more additional target functions have been mentioned in *Table 5-1* below.

Table 5-1 Project 1 - LNBTI Door Access Control - Unit test code completeness

Target Function Name	Completeness (%)	Remarks
CheckCardPermission()	90	Input and output value comparison has not been carried out for some calling functions.
getMyIpAddress()	80	Invalid inputs for some conditions have not been highlighted. Condition coverage can be improved.
InitializeDoorLock()	100	-

InitializeCard()	90	Input and output value comparison has not been carried out for some calling functions.
SwithInputDetect()	100	-
getTimeAndDate()	100	-

5.2 Project 2 – NHRM NFC Base

5.2.1 Description

This project is for one of the modules of a system with several modules for a private university in Sri Lanka. The modules of the system are for various sections in the university such as; access controlling, attendance monitoring, gym usage, library usage, vending machine usage, etc.

To use in the system, every university member has been issued an NFC card. In the system, modules for the interactions with the system users, NFC related modules and DBs are available. There is a need of a module in the system for performing the interaction between these independent modules and carry out the necessary actions for the system users. The target project here is the one to build that module.

This project is implemented using C++ programming language. But, the modules for the user interactions have been implemented using C# programming language. Wrapper modules have been used for the interaction between these modules. For, NFC related modules and DB related modules, direct access is possible as those are also implemented using C++ programming language.

5.2.2 Target function

The function selected here is a one which NHRM NFC Base module calls the NFC related modules. Every university member is included in one or more permission groups for various access permission grants within the university. Those permission groups are saved in the NFC card issued to the university members. What this target function does is,

extracting permission group related data from the NFC card and convert that data into a usable format.

The system is designed so that, every university member using the system is assigned a unique university code. Within this target function, before acquiring permission group related data from the NFC card, university code is validated. This is done by matching the university code stored in the system DB and the university code saved in the university member's NFC card. Only if the card owner is a member of the expected university, the permission group related data is obtained. Otherwise, the process is terminated.

Permission group related data is saved in the NFC card in the little-endian format so that, the obtained bytes are converted to the exact values within this target function. Other than that, different types of logging functions have been used in the function.

An image of the target function is as in *Figure 5-4* below.

```
int NHRMNFCBaseModule::nfc_GetPermissionGroups(char *sessionId, unsigned short *pszPermissionGroups)
{
    NHRMNFCBaseNHRMNFCBaseModule->nfcLogTrace(__LINE__, __FILENAME__, __FUNCTION__, "nfc_GetPermissionGroups In");
    Error::ErrorType _ret = Error::NFC_E_OTHER;

    unsigned char permission[PERMISSIONGROUP_L + 1];
    memset(permission, 0, sizeof(permission));
    int length = PERMISSIONGROUP_L;
    int l_noBitShift = 0;
    int l_permissionGrpIndex = 0;

    char Buffer[PERMISSIONGROUP_L + 1];
    memset(Buffer, 0, sizeof(Buffer));
    memset(pszPermissionGroups, 0, sizeof(pszPermissionGroups));

    unsigned short result = 0;
    unsigned short index = 0;
    unsigned short permissionGrps[MAX_ACCESS_GROUPS_COUNT_CARD];
    memset(permissionGrps, 0, sizeof(permissionGrps));

    if (UNICODE_EMPTY_CHK_FLAG == 1)
    {
        _ret = Error::NFC_E_UNIVERSITY_CODE_MISMATCH;
        NHRMNFCBaseNHRMNFCBaseModule->nfcLogError(__LINE__, __FILENAME__, __FUNCTION__, "University Code is Empty -> NFC_E_
        return _ret;
    }

    try
    {
        _ret = sfnfcGu.sf_GetPermissionGroups(permission, &length);//due to felica interfce requirement send 128 as length
        if (_ret == Error::NFC_S_SUCCESS)
        {
            /***********************************************
            "permission" array has little endian values, so get the bytes
            and convert them to exact values. And break the for loop
            if "0" returns from "permissionGrps" array. Because that's the
            end of array.
            ***********************************************/
            for (int i = 0; i < PERMISSIONGROUP_L; i++)
            {
                l_noBitShift = i % 2;
                result += permission[i];

                if (l_noBitShift == 0)
                {
```

Figure 5-4 Project 2 - NHRM NFC Base - Target function

5.2.3 Unit test specification

The target function does not have that much of function calls. But, as there are different types of log functions have been used in the target function and each log function has several parameters, the unit test specification has become somewhat wider.

Also, the target function does not perform multiple checks in it. Therefore, the number of unit test cases are lesser in the unit test specification. Additionally, there is a global variable check as a global variable exists in the target function. Further, as this global variable is of integer data type, boundary value checks have also been performed.

An image of the unit test specification is as in *Figure 5-5* below.

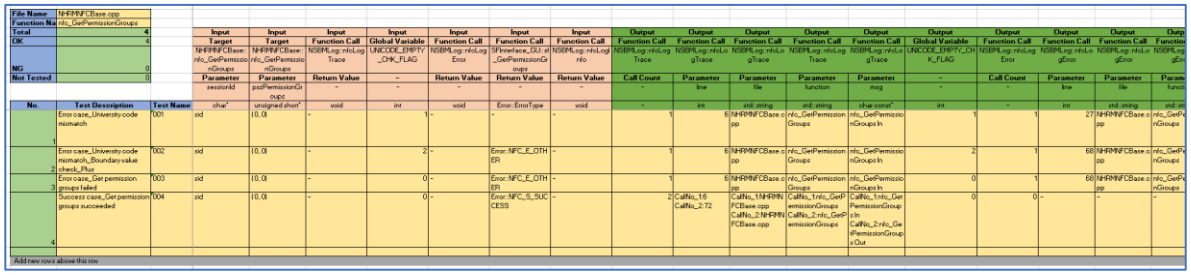


Figure 5-5 Project 2 - NHRM NFC Base - Unit test specification

5.2.4 Unit test code

Unit test code generated for the target function is not much lengthier as the other function usage and the number of checks is lesser in it. As the result, in the unit test code, number of structures, number of mock methods and number of test functions are also lesser. Further, global variable checks and multiple calls of the same functions are checked in this unit test code.

An image of the unit test code is as in *Figure 5-6* below.

```
struct {
    int in_line;
    std::string in_file;
    std::string in_function;
    char const* in_msg;
    int callCount;
} Chk_nfcLogInfo;

/* Unit test class */
class UnitTest_nfc_GetPermissionGroups : public ::testing::Test {
public:
    void SetUp() {
        Chk_nfcLogTrace.callCount = 0;
        Chk_nfcLogError.callCount = 0;
        Chk_sf_GetPermissionGroups.callCount = 0;
        Chk_nfcLogInfo.callCount = 0;
    }

    void TearDown() {

    }
};

/* Mock classes */
class MockNSBMLog : public NSBMLog {
public:
    MOCK_METHOD4(nfcLogTrace, void(int line, std::string file, std::string function, char const* msg));
    MOCK_METHOD4(nfcLogError, void(int line, std::string file, std::string function, char const* msg));
    MOCK_METHOD4(nfcLogInfo, void(int line, std::string file, std::string function, char const* msg));
};

class MockSFInterface_GU : public SFInterface_GU {
public:
    MOCK_METHOD2(sf_GetPermissionGroups, Error::ErrorType(unsigned char* permission, int* length));
};

/*
    Error case_University code mismatch
*/
TEST_F(UnitTest_nfc_GetPermissionGroups, 001) {
```

Figure 5-6 Project 2 - NHRM NFC Base - Unit test code

5.2.5 Target functions and expert judgement

The expert judgement for the completeness of the unit test code for the above target function and five more additional target functions have been mentioned in *Table 5-2* below.

Table 5-2 Project 2 - NHRM NFC Base - Unit test code completeness

Target Function Name	Completeness (%)	Remarks
sf_GetPermissionGroups()	90	Boundary value check for unrealistic value is carried out.
nfc_PayBusFare()	100	-
nfc_CardInitialize()	70	Invalid inputs for some conditions have not been highlighted.

		Input and output value comparison has not been carried out for some calling functions.
nfc_GetName()	90	Boundary value check for unrealistic value is carried out.
DBConnectDb()	80	Condition coverage can be improved.
checkCardSyncStatus()	90	Input and output value comparison has not been carried out for some calling functions.

5.3 Project 3 – MPOS (Mobile Point of Sales) SWB

5.3.1 Description

This project is for one of the modules (MPOS module) of a system with several modules for schools in Sri Lanka. The system is for automatic attendance management and in-school student payment management. To achieve these purposes, several modules are included in this system such as; attendance module, MPOS module, top-up module, wristband issuance module, monitoring module, NFC module, RFID (Radio Frequency Identification) module etc.

Both RFID and NFC technologies have been used for this system. For automatic attendance detection, RFID has been used. For the payments within the school, NFC has been used. Both RFID and NFC chips have been embedded to a wristband which is to be worn by the students. As the NFC technology, the technology by NXP (NXP Mifare chips) have been used for this project.

5.3.2 Target function

The target function selected from this project is of a sub-module of the MPOS module to access the NFC related modules.

There are two transaction limits set for the students in this system for the in-school payments. Those are, maximum amount per transaction and maximum total amount per day. Those limits are saved in the NFC chip and before every payment, those limits are obtained from the NFC chip and current payment amounts are validated with those. Payment accepting or rejecting is done based on these validations.

The target function is for saving those transaction limits in the NFC chip. The transaction limits obtained from the system are forwarded to the NFC related modules to save in the NFC chip using this target function.

An image of the target function is as in *Figure 5-7* below.

```
Error::NFCErrorType SampathNFC::setTxLimits(int p_perTxLimit, int p_DailyTxLimit)
{
    SampathNFCLog->LogTrace(__LINE__, __FILENAME__, __FUNCTION__, "setTxLimits In");
    Error::NFCErrorType _ret = Error::NFC_E_OTHER;
    TxLimits l_txLimits;
    memset(&l_txLimits, 0, sizeof(l_txLimits));

    l_txLimits.perTxLimit = p_perTxLimit;
    l_txLimits.dailyTxLimit = p_DailyTxLimit;

    _ret = nmInterfaceSch.nm_SetTxLimits(l_txLimits);
    if (_ret != Error::NFC_S_SUCCESS)
    {
        SampathNFCLog->LogError(__LINE__, __FILENAME__, __FUNCTION__, "NMInterface_Sch::nm_SetTxLimits Error %ld", _ret);
        SampathNFCLog->LogTrace(__LINE__, __FILENAME__, __FUNCTION__, "setTxLimits Out %ld", _ret);
        return _ret;
    }
    else
    {
        SampathNFCLog->LogInfo(__LINE__, __FILENAME__, __FUNCTION__, "NMInterface_Sch::nm_SetTxLimits Success");
    }

    SampathNFCLog->LogTrace(__LINE__, __FILENAME__, __FUNCTION__, "setTxLimits Out %ld", _ret);
    return _ret;
}
```

Figure 5-7 Project 3 - MPOS SWB - Target function

5.3.3 Unit test specification

The target function does not have that much of function calls. But, as different types of log functions have been used in the target function and each log function has several parameters, the unit test specification has become somewhat wider.

Also, the target function only performs a single check in it. Therefore, only two unit test cases exist in the unit test specification. There are checks only related to function calls. A

single log function is called more than once in a single run of the function so that, there are output value checks for more than one function call for that function.

An image of the unit test specification is as in *Figure 5-8* below.

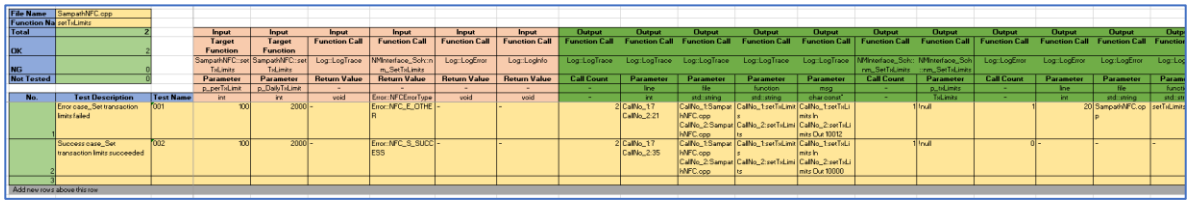


Figure 5-8 Project 3 - MPOS SWB - Unit test specification

5.3.4 Unit test code

The unit test code generated for the target function is not much lengthier as no other function usage and only one check exists in it. As the result, in the unit test code, number of structures and number of mock methods are also lesser. And, there are only two test functions exist in the unit test code. Further, only one function is checked for multiple calls of that same function in this unit test code.

An image of the unit test code is as in *Figure 5-9* below.

```
    char const* in_msg;
    int callCount;
} Chk_LogError;

struct {
    int in_line;
    std::string in_file;
    std::string in_function;
    char const* in_msg;
    int callCount;
} Chk_LogInfo;

/* Unit test class */
class UnitTest_setTxLimits : public ::testing::Test {
public:
    void SetUp() {
        Chk_LogTrace.callCount = 0;
        Chk_nm_SetTxLimits.callCount = 0;
        Chk_LogError.callCount = 0;
        Chk_LogInfo.callCount = 0;
    }

    void TearDown() {

    }
};

/* Mock classes */
class MockLog : public Log {
public:
    MOCK_METHOD4(LogTrace, void(int line, std::string file, std::string function, char const* msg));
    MOCK_METHOD4(LogError, void(int line, std::string file, std::string function, char const* msg));
    MOCK_METHOD4(LogInfo, void(int line, std::string file, std::string function, char const* msg));
};

class MockNMInterface_Sch : public NMInterface_Sch {
public:
    MOCK_METHOD1(nm_SetTxLimits, Error::NFCErrorType(TxLimits p_txLimits));
};

/*
    Error case_Set transaction limits failed
*/
TEST_F(UnitTest_setTxLimits, 001) {
```

Figure 5-9 Project 3 - MPOS SWB - Unit test code

5.3.5 Target functions and expert judgement

The expert judgement for the completeness of the unit test code for the above target function and five more additional target functions have been mentioned in *Table 5-3* below.

Table 5-3 Project 3 - MPOS SWB - Unit test code completeness

Target Function Name	Completeness (%)	Remarks
setTxLimits()	100	-
pay()	80	Invalid inputs for some conditions have not been highlighted.
topUp()	80	Invalid inputs for some conditions have not been highlighted.

getSystemDateTime()	80	Input and output value comparison has not been carried out for some calling functions.
getNFCId()	100	-
SetCardStatus()	100	-

5.4 Project 4 – SCADA DAM

5.4.1 Description

The product which includes this target module is for a well-known Japanese company. The solution is for monitoring and managing alert units. An alert unit is a device produced by the same company to connect multiple sensors and monitor/manage those sensors.

Though an alert unit can monitor and manage the sensors connected to that device, the Japanese company did not have a product to monitor and manage multiple alert units from a single place. This product gives a solution to that problem and it has mainly three modules called; data acquisition module, web application and web API. Additionally, it is included several sub modules to carry out various needs such as, sending notifications to the users, installing the product, configuring the product, etc. Most of them are Windows services.

The project selected here is for the data acquisition module. It also creates a Windows service and it is implemented using the C++ programming language. Data acquisition module is responsible for obtaining the alert unit information and the information of the sensors connected to alert units when the system requested for those information.

5.4.2 Target function

The function selected here is from the initialization section of the data acquisition module. Within this function, four of the information stored in the configuration file are read into the module and all of those are related to the local DB used by the module.

As there are only configuration file information reading happening in this target function, the checks performed here is only to check whether the requested information is correctly

read into the module. Therefore, a single function is called multiple times to perform that check.

An image of the target function is as in *Figure 5-10* below.

```
int Configuration::Init()
{
    DWORD nRet = -1;

    // IP
    TCHAR readBuffer[SERVICE_INIFILE_FIELD_LENGTH + 1];
    memset(readBuffer, 0, sizeof(readBuffer));
    GetPrivateProfileString(L"LOCAL_DB_SETTING", L"LOCALDB_URL", L"", readBuffer, sizeof(readBuffer), Configuration::INI_FILE_NAME);
    nRet = GetLastError();
    if (nRet != 0) {
        return nRet;
    }
    sprintf_s(Configuration::DBACCESS_URL, "%ls", readBuffer);

    // USER NAME
    char encodedData[SERVICE_INIFILE_FIELD_LENGTH + 1] = { 0 };
    memset(readBuffer, 0, sizeof(readBuffer));
    memset(encodedData, 0, sizeof(encodedData));
    GetPrivateProfileString(L"LOCAL_DB_SETTING", L"LOCALDB_USERNAME", L"", readBuffer, sizeof(readBuffer), Configuration::INI_FILE_NAME);
    nRet = GetLastError();
    if (nRet != 0) {
        return nRet;
    }
    sprintf_s(encodedData, "%ls", readBuffer);
    //strcpy_s(Configuration::DBACCESS_USERNAME, base64_decode(encodedData).c_str());
    strcpy_s(Configuration::DBACCESS_USERNAME, encodedData);

    // PASSWD
    memset(readBuffer, 0, sizeof(readBuffer));
    memset(encodedData, 0, sizeof(encodedData));
    GetPrivateProfileString(L"LOCAL_DB_SETTING", L"LOCALDB_PASSWORD", L"", readBuffer, sizeof(readBuffer), Configuration::INI_FILE_NAME);
    nRet = GetLastError();
    if (nRet != 0) {
        return nRet;
    }
    sprintf_s(encodedData, "%ls", readBuffer);
    //strcpy_s(Configuration::DBACCESS_PASSWORD, base64_decode(encodedData).c_str());
    strcpy_s(Configuration::DBACCESS_PASSWORD, encodedData);

    // DB NAME
    memset(readBuffer, 0, sizeof(readBuffer));
    GetPrivateProfileString(L"LOCAL_DB_SETTING", L"LOCALDB_DBNAME", L"", readBuffer, sizeof(readBuffer), Configuration::INI_FILE_NAME);
    nRet = GetLastError();
    if (nRet != 0) {
```

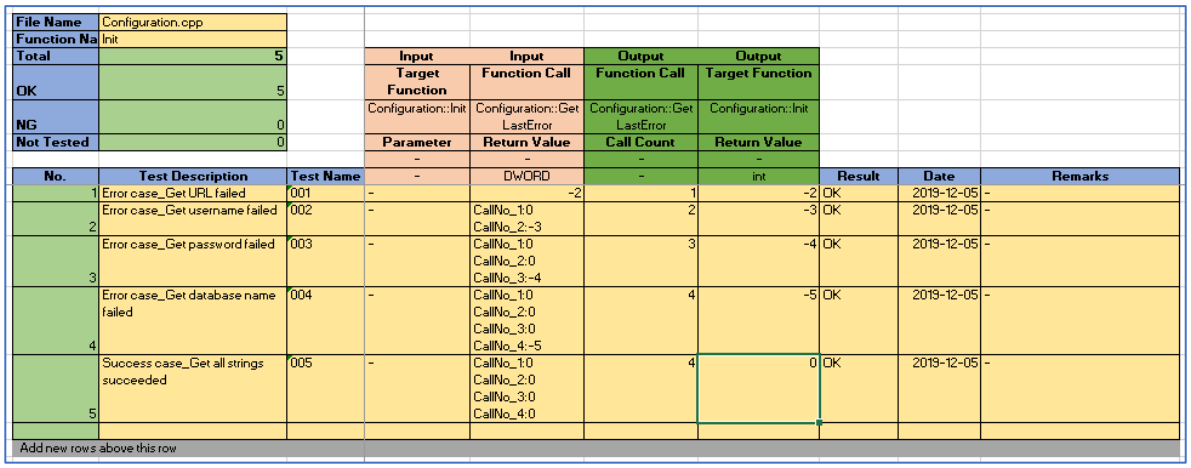
Figure 5-10 Project 4 - SCADA DAM - Target function

5.4.3 Unit test specification

The target function only has a single calling function which is to be considered in unit testing. Also, it does not have parameters. As the result, unit test specification for this function is very small and compact.

But, the specialty of this function is that, it calls the same function multiple times. But, from unit test case to unit test case, number of calls also differ as if an error occurred, a function return happens.

An image of the unit test specification is as in *Figure 5-11* below.



File Name	Configuration.cpp								
Function Na	Init								
Total	5		Input	Input	Output	Output			
OK	5		Target Function	Function Call	Function Call	Target Function			
NG	0		Configuration::Init	Configuration::Get LastError	Configuration::Get LastError	Configuration::Init			
Not Tested	0		Parameter	Return Value	Call Count	Return Value			
			-	-	-	-			
No.	Test Description	Test Name	-	DWORD	-	int	Result	Date	Remarks
1	Error case_Get URL failed	001	-	-2	1	-2	OK	2019-12-05	-
2	Error case_Get username failed	002	-	CallNo_1:0 CallNo_2:-3	2	-3	OK	2019-12-05	-
3	Error case_Get password failed	003	-	CallNo_1:0 CallNo_2:0 CallNo_3:-4	3	-4	OK	2019-12-05	-
4	Error case_Get database name failed	004	-	CallNo_1:0 CallNo_2:0 CallNo_3:0 CallNo_4:-5	4	-5	OK	2019-12-05	-
5	Success case_Get all strings succeeded	005	-	CallNo_1:0 CallNo_2:0 CallNo_3:0 CallNo_4:0	4	0	OK	2019-12-05	-
Add new rows above this row									

Figure 5-11 Project 4 - SCADA DAM - Unit test specification

5.4.4 Unit test code

The unit test code generated for the target function is in average length. It only has a single function call which is considered in unit testing. As the result, in the unit test code, number of structures and the number of mock methods has become very minimal.

But, as the same function is called multiple times in the target function, number of unit test cases has been increased. Because of that, the unit test length has become average. Also, this unit test provides a good example on how Google Test perform return value checks for each call of the same function, separately.

An image of the unit test code is as in *Figure 5-12* below.

```
/* Includes - Basic */
#include "gtest/gtest.h"
#include "gmock/gmock.h"

/* Includes - Specific (TODO: Add correct ones) */
#include "..\GoogleTest_MathApp\SCADA_DAM\Configuration.cpp"

/* Using */
using ::testing::_;
using ::testing::Return;

/* Structures */
struct {
    int retVal;
} TargetFunc;

struct {
    DWORD retVal[4];
    int callCount;
} Chk_GetLastError;

/* Unit test class */
class UnitTest_Init : public ::testing::Test {
public:
    void SetUp() {
        Chk_GetLastError.callCount = 0;
    }

    void TearDown() {

    }
};

/* Mock classes */
class MockConfiguration : public Configuration {
public:
    MOCK_METHOD0(GetLastError, DWORD());
};

/*
    Error case_Get URL failed
*/
TEST_F(UnitTest_Init, 001) {

    /* Arrange */
```

Figure 5-12 Project 4 - SCADA DAM - Unit test code

5.4.5 Target functions and expert judgement

The expert judgement for the completeness of the unit test code for the above target function and five more additional target functions have been mentioned in *Table 5-4* below.

Table 5-4 Project 4 - SCADA DAM - Unit test code completeness

Target Function Name	Completeness (%)	Remarks
Init()	100	-
csdjNetStart()	80	Invalid inputs for some conditions have not been highlighted. Condition coverage can be improved.

connectWithTimeout()	100	-
csdjNetConnect()	90	Input and output value comparison has not been carried out for some calling functions.
csdjNetDisconnect()	80	Invalid inputs for some conditions have not been highlighted.
csdjNetClear()	80	Invalid inputs for some conditions have not been highlighted.

5.5 Project 5 – Sony FeliCa NFC

5.5.1 Description

The module related to this project is to read data from and write data to NFC cards which are based on the NFC technology by Sony (Sony FeliCa). This module is implemented to be used as a common module for the NFC related products developed. To perform communications with NFC cards with increased security such as payments, a separate hardware module called SAM (Secure Access Module) is used. Sony FeliCa NFC product is built in such a way that, it supports both communications with SAM and without SAM.

The module has two sub modules. One is the NFC Driver, which directly communicates with the NFC cards. Therefore, NFC Driver does not relate to the domain information of the product. The other sub module is the NFC Interface, which sits between the NFC driver and whichever the module requests the data from or writes the data to NFC cards. NFC Interface is the sub module which relates with the domain information of the product and therefore, has modifications product to product. The target function here is from NFC Interface sub module.

5.5.2 Target function

This function is to get the credit balance stored in the NFC card. The areas in the NFC card where the data is stored are accessed using services. First, this function creates the keys to access the service for obtaining the credit balance. Then, the service is authenticated. After that, the corresponding data blocks where the needed data is stored

are read into the function. As the resultant data comes as a character array, finally, the card balance is converted to an integer value using a utility function and the result is returned from the function.

This function does several checks for return values by the calling functions. Those are; whether an error occurred when the service is authenticated, whether an error occurred when the data is read from the NFC card and whether the length of the data read from the NFC card is too short to be used in the function. Other than that, functions for several logs of different log types have also been used.

An image of the target function is as in *Figure 5-13* below.

```
Error::ErrorType SFInterface_GU::sf_GetCardBalance(int *cardBalance)
{
    SFInterfaceLogSFInterface_GU->nfcLogTrace(__LINE__, __FILENAME__, __FUNCTION__, "sf_GetCardBalance In");
    Error::ErrorType ret = Error::NFC_E_OTHER;

    // Service count
    unsigned char serviceCount = 0x01;

    // Service key list
    Services services[1] = { Purse2_3_DI_WS };
    unsigned char serviceKeyList[4 * 1];
    createServiceKeyList(services, serviceKeyList, serviceCount);

    // Mutual authenticate service
    ret = MAuthV2WithFeliCa(SYSTEM_CODE, serviceCount, serviceKeyList);

    // Error occured in mutual authenticating
    if (ret != Error::NFC_S_SUCCESS) {
        SFInterfaceLogSFInterface_GU->nfcLogError(__LINE__, __FILENAME__, __FUNCTION__, "Mutual authenticate Error -> NFC_E_OTHER");
        return Error::NFC_E_OTHER;
    }

    // Block number and block list
    unsigned char no_of_blocks = 0x01;
    unsigned char block_list[] = { 0x80, 0x00 };

    // Variables to get data from Sony FeliCa card
    unsigned long readLen = CARD_BLOCK_SIZE * BALANCE_BLOCK_COUNT;
    unsigned char readData[CARD_BLOCK_SIZE * BALANCE_BLOCK_COUNT];
    memset(readData, 0, CARD_BLOCK_SIZE * BALANCE_BLOCK_COUNT);

    // Read from Sony FeliCa card
    ret = ReadDataBlock(no_of_blocks, block_list, readLen, readData);

    // Error occured in reading from Sony FeliCa card
    if (ret != Error::NFC_S_SUCCESS) {
        SFInterfaceLogSFInterface_GU->nfcLogError(__LINE__, __FILENAME__, __FUNCTION__, "Reading from Sony FeliCa card Error -> NFC_E_CARD_READ");
        return Error::NFC_E_CARD_READ;
    }

    // Read length is too short
    if (readLen < CARD_BLOCK_SIZE * BALANCE_BLOCK_COUNT) {
        SFInterfaceLogSFInterface_GU->nfcLogError(__LINE__, __FILENAME__, __FUNCTION__, "Read length is too short Error, Length: %lu", readLen);
        return Error::NFC_E_CARD_READ;
```

Figure 5-13 Project 5 - Sony FeliCa NFC - Target function

5.5.3 Unit test specification

The number of unit test cases here is average because, it only has the function call return value checks other than the success case. But as it has several function calls and as those

functions have several parameters each, the length of the unit test specification (especially on the output side) has been increased.

An image of the unit test specification is as in *Figure 5-14* below.

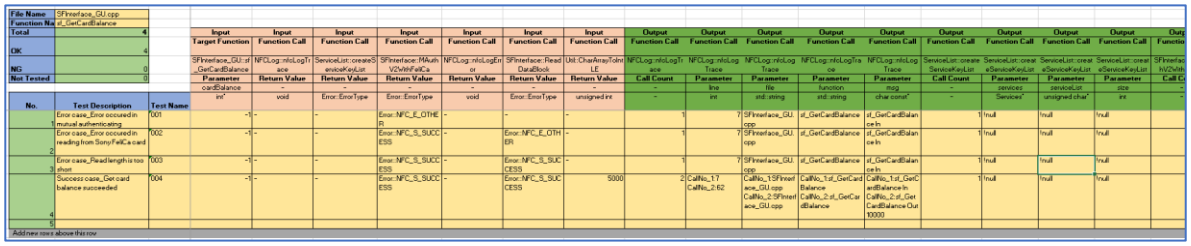


Figure 5-14 Project 5 - Sony FeliCa NFC - Unit test specification

5.5.4 Unit test code

As the unit test specification is also in an average size, unit test code length has also become average. And, this is for all the major sections in the unit test code such as; structures, mock methods and unit test functions. Content wise, the techniques used are also the ones that were used in the previously discussed functions.

An image of the unit test code is as in *Figure 5-15* below.

```
/* Includes - Basic */
#include "gtest/gtest.h"
#include "gmock/gmock.h"

/* Includes - Specific (TODO: Add correct ones) */
#include "../GoogleTest_MathApp/NHRMNFCBase/SFInterface_GU.cpp"

/* Using */
using ::testing::_;
using ::testing::Return;

/* Structures */
struct {
    int* in_cardBalance;
    Error::ErrorType retVal;
} TargetFunc;

struct {
    int in_line[2];
    std::string in_file[2];
    std::string in_function[2];
    char const* in_msg[2];
    int callCount;
} Chk_nfcLogTrace;

struct {
    Services* in_services;
    unsigned char* in_serviceList;
    int in_size;
    Error::ErrorType retVal;
    int callCount;
} Chk_createServiceKeyList;

struct {
    unsigned char in_systemCode[2];
    const unsigned char in_serviceCount;
    const unsigned char in_serviceList[];
    Error::ErrorType retVal;
    int callCount;
} Chk_MAuthV2WithFeliCa;

struct {
    int in_line;
    std::string in_file;
    std::string in_function;
```

Figure 5-15 Project 5 - Sony FeliCa NFC - Unit test code

5.5.5 Target functions and expert judgement

The expert judgement for the completeness of the unit test code for the above target function and five more additional target functions have been mentioned in *Table 5-5* below.

Table 5-5 Project 5 - Sony FeliCa NFC - Unit test code completeness

Target Function Name	Completeness (%)	Remarks
sf_GetCardBalance()	90	Input and output value comparison has not been carried out for some calling functions.
sf_ConnectReader()	100	-
sf_DisconnectReader()	100	-
sf_ConnectCard()	90	Boundary value check for unrealistic value is carried out.

sf_GetCardID()	100	-
sf_SetBusRoute()	90	Input and output value comparison has not been carried out for some calling functions.

5.6 Feedback

The software developers of each of the projects discussed above got hands-on experience of the suggested unit test specification format and the Google Test Unit Test Code Generator tool and they gave feedbacks on their experience of using those. Further, they suggested some improvements for a better and an efficient usage of the tools.

Here in this section, feedbacks of all the software developers have been summarized as they also had common points about the tools.

5.6.1 Positives

Easiness and fastness to write error-free unit test cases

The proposed unit test specification format has made writing unit test cases very easy. When creating the unit test specification header, already written header sections for other functions can be reused easily, if they are similar. Therefore, it becomes a simple column copy & paste and some slight modifications on the pasted columns.

When it comes to writing unit test cases also, it can be easily reused previously written unit tests. Here also, it becomes a simple row copy & paste and some slight modifications on the pasted rows.

Other than that, most of the items in the unit test specification can be filled by just copying & pasting directly from the source code itself. It not only makes creating the unit test specification easier and faster, but also it also reduces the chances of errors occurred in the unit test specification due to the typos by the software developer.

Some of items that can by directly copied and pasted from the source code to the unit test specification can be listed as follows.

- File name
- Function names
- Parameter names
- Variable types
- Return values
- Parameter values

Usage of proper IDEs or source code editors also help for identifying call counts of functions and usages of variables within the target function. Visual Studio and Notepad++ are two examples for such tools. When highlighted a function or a variable inside a function, it shows all the other places the highlighted item has been used in it. Therefore, it is useful when creating unit test specifications easily, fast and to become error-free.

Better comprehension of the source code

As the unit test specification format has covered most of the aspects of the target function, the software developer gets more comprehension of the source code. Most of the time, software developers tend to write source code without analyzing much in the cases such as global variable usage, usage of the functions created by other software developers, etc.

But, when he writes the unit test specification using this unit test specification format, he gets a clear idea on how various elements have been used in the target function. It is useful to write better, quality source code.

Identification of bugs when writing unit test specification

It is obvious that when a unit test code is created and run, some of the bugs existing in the target function can be identified. But, based on the format of the unit test specification format, the software developers could identify bugs and mistakes in the target function done by themselves, even when creating the unit test specification.

Accuracy and usability of the unit test code generated

Unit test code generated by the Google Test Unit Test Code Generator tool is very accurate. Therefore, software developers do not have to do any modifications on it, if the includes list has been given correctly to the tool before generating the unit test code. The generated unit test code can be used for unit testing as it is.

Reminder on checking boundary values is useful

Especially when there are condition checks in the target function, software developers write the test cases to pass and fail that check. But, when the condition check is based on a numerical value, the boundary values should also be checked when unit testing is done. Boundary values are the closest values on both the sides of the value in the condition check.

However, checking the boundary values is easily forgotten by the software developers when doing unit testing. But, here in the Google Test Unit Test Code Generator tool, for numerical value based condition checks, it checks whether the software developer has included unit test cases for boundary value checking in the unit test specification. If not, the tool gives a warning message to the software developer informing the lack of boundary value checking.

5.6.2 Negatives

Unit test specification's No.x sheet becomes lengthy

For some unit test target functions, unit test case table becomes lengthy when there are many target function parameters, global variables or parameters in calling functions. This happens especially for the output section of the table and when there are many function calls which have many parameters. Here, it becomes harder when creating the unit test specification and error prone. Also, navigating through the unit test case table becomes difficult.

As a remedy for this case, ‘Grouping’ feature of Microsoft Excel has been added to both input and output column sections in the unit test case table. By adding this, software developer can expand/collapse the groups based on where he is working in the unit test case table.

When a group is collapsed, the unit test specification’s No.x sheet looks as in *Figure 5-16* below. If the unit test table is lengthy, software developer can hide one column section as in *Figure 5-16* below and work in the other column section.

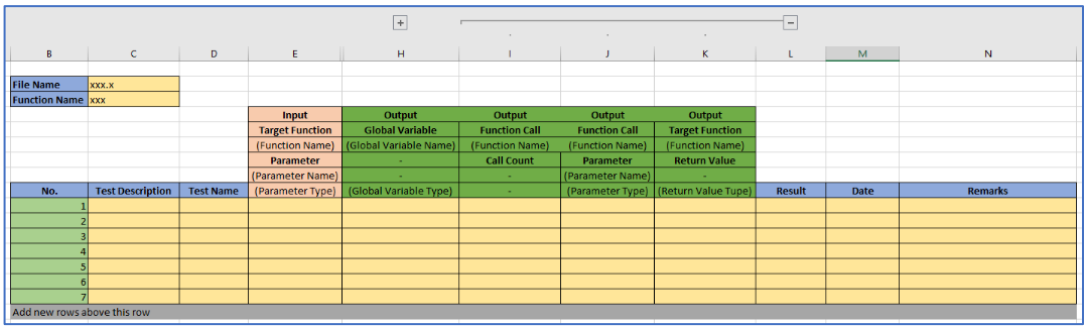


Figure 5-16 Column grouping in Microsoft Excel

Column headers and test case details get hidden when navigating through unit test case table

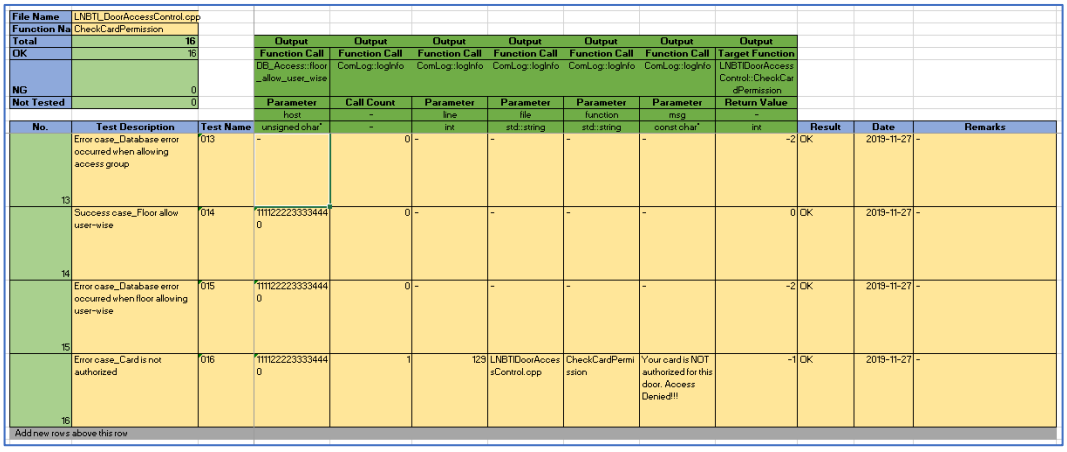
When there are many columns in the unit test case table, software developer has to scroll right to view the information at the right side of the table. But, when doing this, target function information and unit test case information get hidden and after scrolling right, it is difficult to check that information again.

On the other hand, when there are many rows in the unit test case table (i.e. many unit test cases have been written), software developer has to scroll down to view the information at the bottom of the table. But, when doing this, column headers get hidden and after scrolling down, it is difficult to check that information again.

In both the above cases, software developer has to scroll sideways many times to get an idea of a unit test case and it makes the unit test case writing inefficient and time consuming.

As a remedy for this case, ‘Freeze Panes’ feature of Microsoft Excel has been added to both columns and rows in the unit test case table.

When the needed columns and rows are frozen, the unit test specification’s No.x sheet looks as in *Figure 5-17* below, after navigating through the data section of the unit test case table. Here, software developer can easily get an idea about the unit test case and unit test case creation process also becomes efficient and speedy.



File Name	LNBTI_DoorAccessControl.cpp												
Function Na	CheckCardPermission												
Total	16		Output	Output	Output	Output	Output	Output	Output				
OK	16		Function Call	Function Call	Function Call	Function Call	Function Call	Function Call	Target Function				
			DB_Access::floor_allow_user_wise	ComLog::logInfo	ComLog::logInfo	ComLog::logInfo	ComLog::logInfo	ComLog::logInfo	LNBTIDoorAccessControl::CheckCardPermission				
NG	0												
Not Tested	0		Parameter	Call Count	Parameter	Parameter	Parameter	Parameter	Return Value				
			host	-	line	file	function	msg	-				
No.	Test Description	Test Name	unsigned char*	-	int	std::string	std::string	const char*	int	Result	Date	Remarks	
13	Error case_Database error occurred when allowing access group	013	-	0	-	-	-	-	-2	OK	2019-11-27	-	
14	Success case_Floor allow user-wise	014	111122223333444 0	0	-	-	-	-	0	OK	2019-11-27	-	
15	Error case_Database error occurred when floor allowing user-wise	015	111122223333444 0	0	-	-	-	-	-2	OK	2019-11-27	-	
16	Error case_Card is not authorized	016	111122223333444 0	1	123	LNBTIDoorAccessControl.cpp	CheckCardPermission	Your card is NOT authorized for this door. Access Denied!!!	-1	OK	2019-11-27	-	
Add new rows above this row													

Figure 5-17 Freeze panes in Microsoft Excel

Cannot generate unit test code for the whole project at once

From the Google Test Unit Test Code Generator tool, only the unit test code for a single source code file can be generated at a time. If all the unit test code for a project could be generated at once, unit test code generation process becomes easier and efficient.

To achieve this, the folder selection capability needs to be added to the Google Test Unit Test Code Generator tool. Currently, only a single unit test specification can be selected there so that, only the unit test code for a single source code file can be generated at once. This feature is planned to be added to the tool in a next phase.

5.6.3 Expert judgement for completion

Most of the point outs from the expert to highlight the incompleteness of the unit test code generated by the Google Test Unit Test Code Generator tool are due to a limitation of the

tool and due to the incompleteness of the schema defined to reduce mistakes when writing unit test cases. The project wise average completeness for the selected target functions for each project is as in *Table 5-6* below.

Table 5-6 Project-wise average completeness

#	Project	Average Completeness (%)
1	LNBTI Door Access Control	93.33
2	NHRM NFC Base	86.67
3	MPOS SWB	90.00
4	SCADA DAM	88.33
5	Sony FeliCa NFC	95.00
	Total average completeness	**90.67**

6 CONCLUSION

In this research work, it was able to introduce a spreadsheet format for unit test specifications. After filling this specification, it automatically become a unit test report, including all the information from unit test results to coverage results. Additionally, a tool was provided to generate unit test code based on the unit test specification created. The generated unit test code targets source code with C++ programming language and Google Test unit test framework.

However, this is the first step of this work and there is room for enhancement. Although there are a lot of benefits of using this unit test specification and unit test code generator tool, there are some limitations to overcome and future enhancements that could be done to have a better output.

6.1 Comparison with other tools

When the work presented in this research is compared with the research works discussed under literature review, the result is as follows. The research work presented in this research is the only one which provides a unit test specification format and a report, though it is the only one which does not provide unit test input data.

Key:

Tool 1: Agitator software

Tool 2: A complete automation of unit testing for Java programs

Tool 3: Combining unit-level Symbolic Execution and system-level Concrete Execution for testing NASA software

Tool 4: Google Test Unit Test Code Generator

Table 6-1 Comparison with other tools

#	Feature	Tool 1	Tool 2	Tool 3	Tool 4
1	Help or guidelines provided to create unit test specifications	No	No	No	Yes
2	Generating input data	Yes	Yes	Yes	No

3	Support for including input data that could easily miss	Yes	Yes	No	Yes
4	Deciding unit test inputs and outputs by the tool	No	Yes	Yes	No
5	Avoiding generation of useless unit test cases	No	Yes	Yes	No
6	Manipulating source code	No	Yes	No	No
7	Generating unit test code	No	Yes	Yes	Yes
8	Ability to provide a unit test report	No	No	No	Yes
9	Amount of manual work to be done	High	Low	Medium	Medium
10	Level of learning curve	Medium	High	Low	Low

6.2 Benefits

- As the unit test code generator tool is a separate software application, it executes separate, without putting software developers' work into a hold. The software developer can separately select the unit test specification path, select the needed unit test specifications to generate unit test code, input the include statements to be written in the generated unit test code, start unit test code generation and switch back to a different work. Once unit test code generation is finished, the tool will notify it to the software developer, and he can use the generated unit test code as he wishes. Therefore, the time it takes to generate unit test code really does not matter that much.

- As the generated unit test code files are complete, after the generation, those can be easily executed (as itself) in the unit test execution environment. Though in this case Microsoft Visual Studio 2017 has been used, it could be used with any unit test setup.

- This work is useful for the software development projects which is required a unit test report (including unit test results, coverage results and bug information) at the end of unit testing. Most of the clients who have outsourced a part of a system to another software development company request for it. As such companies are also aware of the software development field, they can refer this report to measure the quality of the unit test carried out.

- Almost zero modifications are needed to the generated unit test code using this tool. If the software developer did not input the includes before the unit test code generation, he has to modify the unit test code output to add those includes. Apart from that, if there are any special includes to be made for the proper execution of unit test code, those should also be included by the software developer, as the tool cannot guess those. However, generated unit test code files include a TODO comment saying that the software developer should check the includes section of the unit test code. Other than includes, there are no modifications to be done for the resultant unit test code.

6.3 Limitations

When a function is called once in a function to be unit tested, the values of the parameters used for that function could be compared (Values before function call and after function call). However, if that function is called more than once, it was not able to compare the parameter values distinguishing each call. According to the research done, it is a limitation in Google Test unit testing framework and therefore, made it as a limitation for Google Test Unit Test Code Generator tool. As the result, it compares the values of the parameters of used functions in the target function, only if that function is called once.

6.4 Future work

- Currently, files with unit test cases can be generated for one unit test specification at a time. Plan to make it possible to generate unit test files for all the unit test specifications in a folder at once. This will address the evaluation feedback that

the software developers wanted to input the unit test specifications of a whole project to the unit test code generator tool at once.

- Checking whether an ample amount of boundary value unit test cases have been written is done as a POC for defining a schema and applied it in the Google Test Unit Test Code Generator tool. This schema to suggest unit test case improvements can be improved to cover most of the human errors that can be occurred at the unit test case writing and even after reviewing.

- Though in this research work, it is not suggested to hand over unit test case input generation completely to tools, it is OK to get such inputs from a tool, review those properly and modify or add inputs accordingly. It can be integrated a unit test input generation tool for this work so that, users will be provided the option whether to use automatic test input generation to have a base to decide input data. In this case, generated test input data can be automatically inserted to the unit test specification also. Unit test specification creator can review and amend those later accordingly.

- For the completeness of the tool, a code coverage tool can be integrated.

- To create the unit test specification, Microsoft Excel has been used as it is widely used in the software development industry. However, it is a paid one. For the users who do not use Microsoft Excel and want to use this code generator tool, better to have a version of this tool which works with free spreadsheet software such as the Calc software in Open Office and Libre Office. It can be created such a version of this tool.

REFERENCES

[1] Y. Cheon, M. Y. Kim and A. Perumandla. “A Complete Automation of Unit Testing for Java Programs,” in proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), Las Vegas, Nevada, June 27-29, 2005, pp. 290-295.

[2] P. Runeson. (2006, Jul./Aug.). “A Survey of Unit Testing Practices.” *IEEE Software.* [On-line]. 23(4), pp. 22-29. Available: https://ieeexplore.ieee.org/abstract/document/1657935 [Mar. 12, 2019].

[3] M. Boshernitsan, R. Doong and A. Savoia, “From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing,” in ISSTA’06, July 17–20, 2006, Portland, Maine, USA.

[4] J. A. Whittaker. (2000, Jan./Feb.). “What Is Software Testing? And Why Is It So Hard?” *IEEE Software.* [On-line]. 17(1), pp. 70-79. Available: https://ieeexplore.ieee.org/abstract/document/819971/authors#authors [Mar. 12, 2019].

[5] M. Ellims, J. Bridges and D.C. Ince. “Unit Testing in Practice,” in proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE '04), Saint-Malo, Bretagne, France, November 2-5, 2004. Available: https://ieeexplore.ieee.org/abstract/document/1383101 [Mar. 12, 2019].

[6] N. H. Petschenik. “Building Awareness of System Testing Issues,” in proceedings of the 8th International Conference on Software Engineering (ICSE ’85), Los Alamitos, CA, USA, 1985. IEEE Computer Society Press, pp. 182–188.

[7] R. Vanmegen and D. B. Meyerhoff. (1995). “Costs and Benefits of Early Defect Detection-Experiences from Developing Client-Server and Host Applications.” *Software Quality Journal.* 4(4), pp. 247–256.

[8] H. Zhu, P.A.V. Hall and J.H.R. May. (1997, Dec.). “Software Unit Test Coverage and Adequacy.” *ACM Computing Surveys (CSUR).* [On-line]. 29(4). Available: https://dl.acm.org/doi/abs/10.1145/267580.267590 [Mar. 12, 2019].

[9] B. Beizer. *Software Testing Techniques, 2nd ed.,* Van Nostrand, Reinhold, 1993.

[10] M. Ellims, J. Bridges and D.C. Ince. (2006, Mar.). “The Economics of Unit Testing.” *Empirical Software Engineering*. [On-line]. 11(1), pp.5-31. Available: https://link.springer.com/article/10.1007/s10664-006-5964-9 [Mar. 12, 2019].

[11] D. Saff and M. D. Ernst. “An Experimental Evaluation of Continuous Testing During Development,” in proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA, 2004 Jul. pp. 76–85.

[12] S. Berner, R. Weber and R. K. Keller. “Observations and Lessons Learned from Automated Testing,” in proceedings of the ICSE’05, St. Louis, Missouri, USA, May 15-21, 2005.

[13] T. Mackinnon, S. Freeman and P. Craig. *Extreme Programming Examined: Endo-Testing: Unit Testing with Mock Objects*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 287-301.

[14] D. Thomas and A. Hunt. (2002, Aug.). “Mock Objects.” *IEEE Software.* [On-line]. 19(3), pp. 22-24. Available: https://ieeexplore.ieee.org/abstract/document/1003449 [Mar. 12, 2019].

[15] N. Tillmann and W. Schulte. “Mock-Object Generation with Behavior,” in proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE ’06), Tokyo, Japan, December 4, 2006. Available: https://ieeexplore.ieee.org/abstract/document/4019611 [Mar. 12, 2019].

[16] A. Sen, IBM Developer. (2010, May.). “A quick introduction to the Google C++ Testing Framework,” [Online article] Available: https://developer.ibm.com/articles/au-googletestingframework/

[17] GitHub. (2019, Aug). “Googletest Mocking (gMock) Framework,” [Online article]. Available: https://github.com/google/googletest/blob/master/googlemock/README.md

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. “Dynamically discovering likely program invariants to support program evolution,” in ICSE ’99: 21st International Conference on Software Engineering, 1999, pp. 213-224.

[19] T. Xie and D. Notkin. “Tool-assisted unit test selection based on operational violations,” in ASE ’03: International Conference on Automated Software Engineering, 2003, pp. 40-48.

[20] Apache Commons (2019, Jul.). “Apache Commons Collections,” [Online article] Available: http://jakarta.apache.org/commons/collections/

[21] M. Srinivas and L. M. Patnaik. “Genetic Algorithms: A Survey.” *IEEE Computer*, vol. 27(6), pp. 17-26, Jan. 1994.

[22] K. Beck and E. Gamma, “Test infected: Programmers love writing tests,” Java Report, vol. 3, no. 7, 1998, pp. 37–50, [Online] Available: http://junit.sourceforge.net/doc/testinfected/testing.htm

[23] G. T. Leavens, A. L. Baker, and C. Ruby. *JML: A Notation for Detailed Design*, Boston: Kluwer Academic Publishers, 1999, pp. 175-188.

[24] P. Tonella, “Evolutionary testing of classes,” in proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA, July 2004, pp. 119–128.

[25] Y. Cheon and G. T. Leavens, “A runtime assertion checker for the Java Modeling Language,” in Proceedings of the International Conference on Software Engineering Research and Practice, Las Vegas, Nevada, June 2002, pp. 322–328.

[26] S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, S. Person and M. Pape. “Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software,” in proceedings of the ACM/SIGSOFT International Symposium on Software

Testing and Analysis, ISSTA 2008, Seattle, Washington, USA, July 20-24, 2008, pp. 15-26.

[27] J. C. King. “Symbolic Execution and Program Testing.” *Communications of the ACM*, vol. 19(7), pp. 385-394, 1976.

[28] EXCELJET. (2019). “Get value of last non-empty cell,” [Online article]. Available: https://exceljet.net/formula/get-value-of-last-non-empty-cell

[29] Tech on the Net - Excel. (2019). “MS Excel 2010: Change the fill color of a cell based on the value of an adjacent cell,” [Online article] Available: https://www.techonthenet.com/excel/questions/cond_format9_2010.php

[30] Ablebits.com. (2018, Jun.). “Excel: How to add a hyperlink to another sheet,” [Online article] Available: https://www.ablebits.com/office-addins-blog/2014/05/15/excel-insert-hyperlink/

[31] Microsoft Community. (2012, Jul.). “Hyperlink based on the text value of a cell,” [Online article] Available: https://answers.microsoft.com/en-us/msoffice/forum/all/hyperlink-based-on-the-text-value-of-a-cell/6eede485-4d84-4657-95a0-dc4371e4dfd0

[32] Ablebits.com. (2019, Feb.). “How to use INDIRECT function in Excel - formula examples,” [Online article] Available: https://www.ablebits.com/office-addins-blog/2015/02/10/excel-indirect-function/#INDIRECT-another-sheet

[33] Ablebits.com. (2018, May.). “COUNTIF in Excel - count if not blank, greater than, duplicate or unique,” [Online article] Available: https://www.ablebits.com/office-addins-blog/2014/07/02/excel-countif-examples/

[34] Microsoft .NET Blog. (2018, Apr.). “Announcing the .NET Framework 4.7.2,” [Online article] Available: https://blogs.msdn.microsoft.com/dotnet/2018/04/30/announcing-the-net-framework-4-7-2/

[35] Stack Overflow. (2013, Jun.). “Open File Dialog, One Filter for Multiple Excel Extensions?,” [Online article] Available: https://stackoverflow.com/questions/17116045/open-file-dialog-one-filter-for-multiple-excel-extensions

[36] Stack Overflow. (2015, Apr.). “How can I make the cursor turn to the wait cursor?,” [Online article] Available: https://stackoverflow.com/questions/1568557/how-can-i-make-the-cursor-turn-to-the-wait-cursor

[37] Coderwall. (2019, Mar.). “Read Excel File in C#,” [Online article] Available: https://coderwall.com/p/app3ya/read-excel-file-in-c

[38] GMock for Windows. (2019). “gmock,” [Online article] Available: https://www.nuget.org/packages/gmock/