

Analytic Methods of Systems and Software Testing

Neha Kaul



Analytic Methods of Systems and Software Testing

Analytic Methods of Systems and Software Testing

Neha Kaul



www.arclerpress.com

Analytic Methods of Systems and Software Testing

Neha Kaul

Arcler Press

224 Shoreacres Road

Burlington, ON L7L 2H2

Canada

www.arclerpress.com

Email: orders@arclereducation.com

e-book Edition 2021

ISBN: 978-1-77407-915-7 (e-book)

This book contains information obtained from highly regarded resources. Reprinted material sources are indicated and copyright remains with the original owners. Copyright for images and other graphics remains with the original owners as indicated. A Wide variety of references are listed. Reasonable efforts have been made to publish reliable data. Authors or Editors or Publishers are not responsible for the accuracy of the information in the published chapters or consequences of their use. The publisher assumes no responsibility for any damage or grievance to the persons or property arising out of the use of any materials, instructions, methods or thoughts in the book. The authors or editors and the publisher have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission has not been obtained. If any copyright holder has not been acknowledged, please write to us so we may rectify.

Notice: Registered trademark of products or corporate names are used only for explanation and identification without intent of infringement.

© 2021 Arcler Press

ISBN: 978-1-77407-806-8 (Hardcover)

Arcler Press publishes wide variety of books and eBooks. For more information about Arcler Press and its products, visit our website at www.arclerpress.com

ABOUT THE AUTHOR



Neha Kaul is an experienced software consultant and technical author currently residing in Paris, France. She is the author of four technical books: Object Oriented Programming with Java, Logging Frameworks in Java, Applications of Data Mining in Engineering, Management and Medicine and Software Security: Building Secure Software Applications. She received her double Master's Degree in Computer and Communication Networks and Information Technology from Telecom SudParis and University Paris-Saclay in 2016. She is a recipient of the prestigious Telecom Scholarship for Excellence provided by Fondation Telecom, France. She received the Bachelor of Engineering degree in Computer Engineering from the University of Pune, India in 2011. From 2011 to 2014, she was employed as a Software Engineer with Geometric Ltd, Pune, India. Her major avenues of research include Advanced Java/J2EE frameworks, Automation Testing, Software Quality, Software Security, Logging Frameworks, Project Management and Leadership.

TABLE OF CONTENTS

<i>List of Figures</i>	<i>ix</i>
<i>List of Tables</i>	<i>xi</i>
<i>List of Abbreviations</i>	<i>xiii</i>
<i>Dedication</i>	<i>xv</i>
<i>Preface</i>	<i>xvii</i>
Chapter 1 Introduction to Testing	1
1.1. What Is Testing?	2
Chapter 2 History of Software Testing	5
Chapter 3 Software and Systems Testing	15
3.1. What Is Software Testing?	16
3.2. Basic Terminologies In Software Testing.....	17
3.3. The Importance of Testing Software	18
Chapter 4 Importance of Testing: Case Studies	27
4.1. Case 1: Ariane 5 Flight 501 Rocket Launch Failure	28
4.2. Case 2: Knight Capital Group Error	31
4.3. Case 3: Nasa’s Mars Climate Orbiter.....	46
4.4. Case 4: Nissan Vehicle Recall	53
4.5. Case 5: Multidata Systems Disaster	58
4.6. Case 6: Therac-25 Medical Accelerator (1985).....	74
4.7. Case 7: Heathrow Terminal 5 Opening	107
4.8. Case 8: Cairns Hospital Software Glitch.....	108
4.9. Case 9: Uber Was Sued Due To A Software Bug.....	113
Chapter 5 Methods of Software and Systems Testing	117
5.1. Objectives of Software Testing.....	118

5.2. Methods of Software Testing.....	124
5.3. Levels of Software Testing.....	142
5.4. Classification of Test Types	149
5.5. Types of Testing And Their Application To Test Levels.....	153
5.6. Factors To Be Considered When Choosing A Testing Technique.....	156
5.7. Steps To Perform Dynamic Testing.....	159
Chapter 6 Software Testing Frameworks	163
6.1. Junit.....	164
6.2. Testng Framework	205
6.3. Selenium Framework	231
6.4. Mockito Framework.....	257
Bibliography.....	293
Index	299

LIST OF FIGURES

Figure 2.1. Testing activities in STEP.

Figure 4.1. Rise in dark pools.

Figure 4.2. Therac-25 modes.

Figure 4.3. Therac-25 assembly.

LIST OF TABLES

Table 2.1. Stages of growth of testing

Table 2.2. Different between testing and debugging

LIST OF ABBREVIATIONS

AECL	Atomic Energy of Canada Limited
AMD	angular momentum desaturation
BDD	behavior-driven development
BH	horizontal bias
CGR	Companie Générale de Radiologie
CIO	Chief Information Officer
ETG	electronic trading group
IAEA	International Atomic Energy Agency
IQ	intelligence quotient
ISTQB	International Software Testing Qualification Board
MeV	million electric volt
NYSE	New York Stock Exchange
PAHO	Pan American Health Organization
RLP	retail liquidity program
SMARS	smart market access routing system
SRI	inertial reference system
STEP	systematic test and evaluation process
TCM	trajectory change maneuver
TDD	test-driven development
TLD	thermoluminescent dosimetry

DEDICATION

To my sister Nidhi, thanks for your unconditional love and support.

PREFACE

Software is an indispensable and integral part of our quotidian lives. We rely a great deal on software for the completion of our routine activities and tasks. We do not usually ponder over the real-life impact of software in the world. The magnitude of involvement of software in our lives is overwhelming. Software systems and applications are used on a massive scale in several crucial and non-critical fields such as healthcare, education, medicine, research, engineering, business, finance, banking, etc. Software systems and applications help us perform a significant number of tasks in a simpler and cost-effective way, thereby revolutionizing the world and improving the quality of life for most. It can be said that software has evolved into one of man's basic necessities.

It is inevitable that the role and relevance of software will continue to rise in the near and distant future. Software systems and applications will surely become more complex and like today continue to be indispensable to solving the world's biggest problems. Consequently, with the inevitable rise of complex software, the challenges associated with testing such software are bound to surge. Effective testing is linked directly to the efficiency and quality of the software. Proper and detailed testing is a crucial part in the life cycle of any software. It is a way to ensure the validity and the quality of any software system. Furthermore, in depth testing helps instill confidence in the quality of the software to its stakeholders. As software systems and applications are privy to a pool of possible issues ranging from defects in the software to potential malicious attacks, it is prudent to evaluate the software as thoroughly as possible depending on its criticality. Comprehensive testing and analysis of any software solution is paramount in order to ensure that no harmful and risky vulnerabilities in the software can be exploited which could have adverse effects on the software and its users.

Ergo, it is imperative that the construction of software systems is robust and secure. One of the chief reasons for software failure is the lack of quality testing. Inadequate testing of software, in most cases results in varying types of losses such as monetary, reputation, clients, etc. Developing software applications of good quality is the precursor to ensuring efficient testing of its quality. A combination of quality and efficient testing is the key to strengthening software systems and making it less susceptible to potential internal issues and external

threats. Naturally, it is safe to assume that testing takes up a significantly large percentage in the determination of the success or effectiveness of a software.

In this book, we will look deeply into the history of testing in the field of software. The risks of inadequate testing mechanisms will be explored in detail. Different mechanisms of software testing will be examined and guidelines for implementation of test strategies in software will be presented in this book. The book covers software testing patterns and software design that helps build robust software applications. Practical implementations of code that explains different test scenarios have been provided in this book.

Introduction to Testing

CONTENTS

1.1. What Is Testing?	2
-----------------------------	---

1.1. WHAT IS TESTING?

Testing is a concept known to us for quite some time now. It has been in use since a long time. Since the old age, testing has been in use in different fields such as science, medicine, military, education, research, retail, etc. Different types of tests in different fields have evolved over the last century.

A test is defined as a method or a process by which the presence, quality, or genuineness of anything is determined (Test, 2020). It is a measure or a channel that helps perform a trial of the validity/quality of a particular entity. A test can, in simpler words be described as the trial of the quality of something.

The implementation and use of tests can be traced back to the early 1900s. For instance, in the United States, in 1942, a formal physical fitness test was introduced to decide who was fit to fight a war (World War II Fitness Test, 2020). This test served as a basis to check if a potential recruit was physically fit to be admitted to the army troops. The test served as a practical way to screen likely recruits as millions of able-bodied men were being summoned to fight in the Second World War. This test was a pragmatic way to discern if a candidate was prepared to handle hardships of combat. This test assessed and selected candidates based on certain parameters such as age, height, weight, eyesight, verbal abilities, etc. Furthermore, a program for the physical development of new recruits was implemented to prepare the men for battle. This training program included a wide variety of physical tasks such as push-ups, squats, pull-ups, sit-ups, and a run of about 300 yards. The intent of this training program was to prepare the soldiers and help them develop the physical endurance that is needed in a real battle.

Similarly, tests have been in practice in the education sector and they can be traced back to the early 1900s. The development of the very first IQ or intelligence quotient for humans is credited to the French psychologists Alfred Binet and Theodore Simon who published the Binet-Simon test in 1905. This focus of this test was primarily the testing of the verbal abilities of people and was initially created with the intention of identifying intellectually disabled school students. This test aimed at identifying ‘slow’ or ‘challenged’ students in order to be able to place them in special educational courses instead of labeling them as ‘sick/unwell’ and sending them to an asylum (Binet and Simon, 1916). Later, in the year 1916, a psychologist at Stanford University named Lewis Terman revised the scale developed by

Binet which led to the conception of the Stanford-Binet Intelligence scales or the Stanford Binet Test which is a standardized intelligence test that is employed in the United States of America. This test is currently in its fifth edition which was published in the year 2003 (Terman and Merrill, 1960).

Soon afterwards, in 1935, Alan Turing came up with a theory of software and software was introduced to the world (Turing, 1935). This was followed by several notable discoveries and inventions such as the Turing machine by Alan Turing in 1938, the ADA programming language by Ada Lovelace in 1843, the assembly language by Kathleen Booth in 1950, the coining of the term ‘compiler’ by Grace Hopper in 1951, the launch of FORTRAN in 1954 and so on (Wirth, 2008). It kept on growing in magnitude at a fast pace. Soon enough, software became mainstream in nature and it garnered a lot of attention worldwide. The rise of software led to the development of heterogeneous operating systems, a plethora of programming languages and its own place in the educational field. As more and more software languages were introduced, more software applications were being developed to perform a range of tasks or actions that helped mankind improve their quality of living. Consequently, as software complexity arose, there arose a need for ensuring the quality of the software application. Promptly, the concept of testing made its way into the software field. We shall delve into the history of software testing in the next chapter.

History of Software Testing

As we are aware, since the inception of software, tests have been present in some form or another. In the early ages of software, since the software developed was more at the system or compiler level, the only way to check if a particular piece of code was working correctly was to debug it. A clear distinction between testing and debugging was not established at that point of time. In the later stages of software evolution, we start to see a clear distinctive line form between debugging and testing. This line of separation kept on growing darker until both were thought of as a separate entity of the software development life cycle as it is today.

The timeline for the growth in testing is classified into different periods as shown in Table 2.1 (Gelperin and Hetzel, 1988).

Table 2.1. Stages of Evolution of Testing

Year	Name
Until 1956	The debugging oriented period
1957–1978	The demonstration oriented period
1979–1982	The destruction oriented period
1983–1987	The evaluation oriented period
1988 onwards	The prevention oriented period

Source: Gelperin and Hetzel (1988).

We shall now look at these different time periods in brief:

1. The Debugging Oriented Period: When software had just come into existence, most of software primarily depended on hardware. Since digital computers were just introduced, there existed more hardware related issue rather than software issues. Although software issues were present, they were overshadowed by the need for reliable hardware. The testing of the reliability of hardware was given more importance than the testing of software reliability. The main target of testing was to ensure that the hardware components were functioning in a reliable and stable way. The initial articles that discuss the phrase or term ‘computer testing’ mostly cover testing of the hardware components rather than its software counterparts.

The original thought process behind testing during this time period was that when you wrote or created a program; you needed to ‘check it out.’ At that time, this was interpreted in different ways. It was a combination of checking if it works, debugging, and testing. These concepts were synonymous to one another. Some described debugging as an event that helped remove ‘bugs’ or ‘issues’ from the system. To others, the process of removing these ‘bugs’ was considered to be ‘testing.’ Some even used both of these terms in an interchangeable fashion. There was no clear distinction for most. But, there did exist a subset of people that believed the two to be different. Yet, many were unable to describe the difference between these terms. Alan Turing, also known as the father of computer science, published an article in 1949 which was one of the earliest works that mention the concept of ‘checking’ a program or routine (Morris and Jones, 1984). This article was presented by Alan Turing at a conference held at Cambridge University in June 1949. His article discussed the use of something called ‘assertions’ in order to check the correctness of a program. This paper stated that a programmer must make the use of several assertions that are definitive and that could be checked in an individual manner in order to verify the correctness of the program as a whole. The paper implies that instead of checking a routine or a program as a whole at a given time, all its parts can be tested one by one to check the whole routine which is more effective than the former approach.

Later, in 1950, Turing went on to publish another article that examines the concept of ‘testing.’ In this article, Turing poses the question ‘How would we know that a program exhibits intelligence?’ He further states that if the demand of the program is intelligence, then how one goes about checking if the requirement of the request is met. He deliberates how one

would know that a program has met all of its requirements. Turing came up with an operational test that could be used to test or check intelligent behavior of a program (Hodges, 2009). He proceeds to define a test for the intelligence of a program in which the behavior of a program and a reference system (a human being) cannot be distinguished by a potential interrogator or tester. The job of the interrogator is to ask questions to the program and the reference system and obtain the same results which suggest that the program satisfies its initial requirements.

2. The Demonstration Oriented Program: In this period, the definition of ‘checking’ was initially used as a means of measuring the correctness of a program. Up until the year 1956, during the debugging oriented period, testing was considered to be synonymous to debugging and no clear distinguishing lines separated the two concepts. In the early phases of this period both debugging and testing were terms that were used to describe the following activities: detection of faults, identification of errors, locating errors in the program and correcting them. Still there was no clear distinction between the two.

But, in the year 1957, Charles Baker successfully managed to separate the two concepts (Baker, 1957). He conducted a review of a book written by Dan McCracken’s called Digital Computer Programming and concluded that debugging and testing are different concepts. He identified two goals for the theory of ‘Program checkout’ namely debugging and testing.

He defined debugging as follows (Baker, 1957):

‘Make sure the program runs.’

He defined testing as follows:

‘Make sure the program solves the problem’

He revolutionized the field of computer programming by providing clear definitions for the two traditionally polemic terms. He clearly defined debugging as a means of ensuring that the program executes properly and testing as a way of ensuring that a correctly executing program correctly solves the given problem that was asked. The phrase described by Baker for testing was later translated to a goal in the life cycle of a program. It became a phase that presented or demonstrated that a particular software program has successfully satisfied its requirements. Finally, there was a clear distinction between the two terms. The separation between the two terms came to be measured in terms of the success of the operation; meaning success in finding faults versus success in identifying faults. This definition of separation eventually led to the emergence of the destruction model

and the destruction oriented period where detection of errors or faults was classified as testing and fault location and fault correction was categorized into debugging.

This is shown Table 2.2, differentiating the terms ‘testing’ and ‘debugging’:

Table 2.2. Different between Testing and Debugging

Destruction Model	Demonstration Model		Aim
	Debug	Test	
Test	Detect	Detect	Detect faults
Debug	Locate, identify, correct	Locate, identify, correct	Fix faults
Aim	Make sure it runs	Make sure it solves the problem	

Source: Gelperin and Hetzel (1988).

This period was identifiable by a rise in need and requirement for software and systems that are effective and ensure reliability in terms of their intended purpose or use. Due to this increased complexity of systems, there was more capital that was being invested into the development of software applications and software systems. Evidently, as the financial input rose, so did the factor of risk associated with the investment. Investors started to expect more in terms of ‘proof of correctness’ in the software that was being delivered to them. They wished to safeguard their assets by counting on a way to determine, isolate, and fix any faults in the system. If errors were detected beforehand, it would potentially save them from losing their revenue if the system fails to operate smoothly. During this period, the failure of a software system usually meant loss of revenue and loss of confidence in the vendor. There did not exist a way to fix errors without significant loss in terms of effort, finance, and time as software systems used a combination of both software and hardware.

It was amidst this era that many organizations that developed software began realizing the importance of testing in software. They were made aware of the benefits of tests following the development of software. Many employers then started to request ‘Testing Skills’ in their job descriptions for a software engineer. They began recruiting and training candidates to perform tests on software systems and applications. Soon software test engineering became a field of study at universities and colleges. It became

an alternate career trajectory for most software engineering graduates. A new career path was created during this time. The field gained popularity and the earliest conference on software testing was held in June 1972 at the University of North Carolina (Gelperin and Hetzel, 1988).

3. The Destruction Oriented Period: The next time period falling between the years of 1979 until 1982 was classified as the destruction-oriented period. The main objective of this period was to locate errors and issues in the software. The primary focus during these years in the field of software was to come up with different ways to detect errors in software applications. The premier or first destruction ‘testing’ model was introduced by Glenford J. Myers. He is credited with the discovery of a model that clearly separated debugging and testing. He defined the process of testing as “*The process of executing a program with the intent of finding errors*” (Meyers, Sandler, and Badgett, 2011). This model proposed by Meyers primarily concentrated on finding faults in the system. His intent was to create test data and test the program against it with the goal of finding faults in the system. His concern was that in the pursuit of a proving that a program has no faults, a programmer or developer may select a data set that would have an extremely low probability of causing errors or failures. But he stated that if the goal of the model is to find faults, the data set would be more heterogeneous and therefore a higher probability of identifying faults which would lead to the development of a better program in the end. This would result in better testing capabilities as well. His model paid a lot of attention to testing the limit or the breaking point of the system. This meant that a test case was supposed to find a bug that would not have been discovered under normal circumstances. His publication stated that “*A successful test case is one that detects an as-yet undiscovered error*” (Meyers, Sandler, and Badgett, 2011).

His work led the focus of the software engineering world to shift from demonstration to detection via destruction models. His iconic contribution to the field of software eventually led to the development of several fault detection models. Software came to be discussed in different terms. Different concepts such as software analysis, software review in addition to testing were introduced to the field of software engineering. Additionally, his work inspired other researchers to come up with a combined fault or error detection models. One such example is the researcher Micheal Deutsch (1961) and William Howden (1982). The clear definition of separation of debugging from testing in practical terms or in terms of a model helped the software community immensely. Several methods and models of testing were being proposed during this time period which clearly illustrated the aspiration

of the software engineering world to separate basic activities involved in software development such as debugging, verification, analysis, and testing.

4. The Evaluation Oriented Period: The period starting from the year 1983 until 1987 was categorized as the evaluation-oriented period. The key aim of this period was the quantification of the quality of the product under consideration. The thought process was that in the life cycle of any software, the product must be evaluated by measuring the quality of the software product in some manner. The focus was to formalize and define the standards for measurement of the quality of a software application or a software system.

In the year 1983, a Guideline for Lifecycle Validation, Verification, and Testing of Computer Software was published by the Institute for Computer Sciences and Technology of the National Bureau of Standards (Adrion, Branstad, and Cherniavsky, 1982). This guideline marks the formalization and introduction of testing into the software engineering literature. This work formally states that testing is a methodology in software which encompasses activities such as analysis, review, and testing. This particular guideline focused on the federal information processing systems. It described a methodology that encompasses analysis, review, and test activities that furnish the product evaluation during the life cycle of a software system or application (Adrion, Branstad, and Cherniavsky, 1982).

Three sets or methods of evaluation are defined in this guideline namely the basic, the comprehensive and the critical method. Each method is more detailed than its predecessor. This means that each method contains the previous method in its implementation. For instance, one cannot apply the comprehensive method without also implementing the basic method by default. The philosophy behind these guidelines was that a single technique couldn't possibly guarantee the creation of correct software without errors. But the thinking was that a careful selection of a set or subset of techniques for a particular project would surely aid the development and maintenance of the software project's quality. It was the belief of the authors that testing strategies for each software development project had to be unique to the project taking into consideration factors particular to the project under test (Adrion, Branstad, and Cherniavsky, 1982).

Additionally, in this period, two standards for testing were published. One standard was released in 1983 on test documentation namely the ANSI/IEEE STD 829 and another standard describing unit testing namely the ANSI /IEEE 1008 was released in 1987 (Gupta, 1987). The first standard

defines the organization of the phase of testing in terms of design and documentation. The standard defines ways to identify test cases. The second standard that was published in 1987 describes the different phases, activities, tasks, and the documents that are a part of a comprehensive unit testing task. Two documents are required as per this particular standard namely a test design specification and a test summary report.

5. The Prevention Oriented Period: From the year 1988 onwards, we move into the prevention-oriented period. It was named or seen as prevention oriented because tests were now being used to show that a software application successfully satisfies its requirements, to identify errors and to avert faults. The focus was to prevent the software system from failing due to unidentified faults in the system. The thinking behind this approach was that prevention of faults would be a better and effective way rather than the reverse.

In this time period, the unit testing standard was further generalized by IEEE in 1985. The IEEE updated this process to include all levels of testing and came up with an integrated version called the systematic test and evaluation process or STEP. STEP is considered to be a comprehensive testing methodology to attain efficiency in testing. Varied testing related activities that are to be performed in parallel to the software development tasks were introduced by The STEP methodology (Gelperin and Hetzel, 1988). It can be described as a system comprised of a range of testing tasks, certain products and roles that help attain the test targets in a way that is cost effective and efficient.

This specific procedure is founded on a prevention life cycle model that looks at testing in the same way as development with the same tasks that are part of any development life cycle. Like the phases of development such as planning, design, and analysis, testing has them as well.

The activities in terms of testing would be as follows:

- i. **Analysis:** Defining or setting the target or objective of the tests.
- ii. **Design:**
 - Defining or constructing the architecture of the test plan.
 - Development of details tests cases and procedures to be followed in terms of testing.
- iii. **Implementation:**
 - Establishing or construction of test data;
 - Development of test procedures;

- Acquiring software needed for testing;
- Execution of the test plans (running and re-running of test cases);
- Calculating the test results.

iv. Maintenance:

- Addition of new tests as the software evolves;
- Saving and backing up of test cases;
- Updating test cases as the software evolves.

This procedure or methodology changed the world of testing as the process of testing now started to include activities usually involved in development such as planning, analysis, design, implementation, execution, and maintenance. During the phase of the implementation of STEP, it was observed that planning of tests, analysis of the software beforehand to prepare the test plan and early design of tests had a positive impact on the outcome of the software project. A significant impact on the quality of the software specifications was noticed. The STEP procedure helped figure out inconsistencies, ambiguities, and incorrectness in the software requirements in the early stages of the life cycle of the software. As the creation of test cases needed a deeper understanding of the requirements of the project, any potential problems in the specifications were highlighted due to inclusion of tests at the analysis phase. This was because a tester's point of view tends to be completely different than that of a developer. They infer the specifications, the requirements in a contrasting manner when compared to other participants. They have to focus a lot more on logic and the functional aspects of the software which enables them to spot flaws in the specifications or the software designs that are provided to them. They can thus find gaps or incomplete data or vagueness in the specifications thereby helping developers build a better product. Additionally, as models were being built to assess the potential behavior of the software, it became simpler to see how the software would behave in certain circumstances. These models were being built based on the specifications and these models helped show how the software would work. This helped elimination of possible errors in the specifications.

For instance, test models consisted of several test cases that consist of input-response type scenarios that help understand, model, and predict the behavior of the software system. By using such sets of test cases, it becomes easy to discern the nature of the software. By using such sets of test cases, it becomes easy to discern the nature of the software as the test cases have

a defined input and an expected response to the input provided. As the test cases have a defined input and an expected response to the input provided. Hence, if the responses are not as expected or if the predicted response is different than what is expected as per the specifications, a flag is raised and the requirements are re-evaluated. Any discrepancy that is found helps uncover issues in any of the parts of the project such as the test plan, the test data, the design, or the specifications. Test cases along with the test data and the set of samples demonstrating the behavior of the software clearly show the potential customers, analysts, architects, developers the consequences or results of the software requirements and the software design choices. When such models started, being developed it helped the development process immensely. The timely construction of test models helped detect the need for changes in the software design and software requirements.

As an added bonus, detection of these issues in the early stages was cost effective as well. Issues that arose in the initial phase of the life cycle were cheaper to correct in the beginning itself rather than after the completion of the development phase. Moreover, a lot of time and effort that would have been spent by the development team is reduced if design problems are detected in the prior stages thereby adding to the financial gain in terms of manpower and time. The uses of a test case set or model to provide or generate a functional definition of the software which is also accessible to the users, clients, and developers helped in improving the quality of the software product.

STEP methodology consists of the following testing activities (Gelperin and Hetzel, 1988):

1. Planning;
2. Acquisition;
3. Measurement.

Each of the above activities consists of the following sub-activities in order (Figure 2.1).

Planning	<ol style="list-style-type: none"> 1. Development of a master test plan; 2. Determination of features to be tested; 3. Development of detailed test plans
Acquisition	<ol style="list-style-type: none"> 1. Designing the test cases; 2. Implementation of the test cases

Measurement	<ol style="list-style-type: none">1. Execution of the tests;2. Checking of termination.3. Evaluation of the results
--------------------	---

Figure 2.1. Testing activities in STEP.

Source: Gelperin and Hetzel (1988).

Further evolution in the field of testing eventually led to conception of test driven development (TDD) where the focus shifted from development/programming to attain a goal to developing/programming by focusing on tests. The introduction of TDD further separated a developer or programmer from a tester. The concept of writing test cases in the analysis phase first and then sharing them with the developers/programmers was soon popularized. This approach was found to produce code that consists of a low defect rate. Today, testing is a discipline of its own and is being taught across universities and different methodologies of testing and testing software are being developed daily. We shall now take a look at software testing and its types in the next chapter.

Software and Systems Testing

CONTENTS

3.1. What Is Software Testing?	16
3.2. Basic Terminologies In Software Testing.....	17
3.3. The Importance of Testing Software	18

3.1. WHAT IS SOFTWARE TESTING?

Software Testing is an activity or action of evaluating a software system or its components in order to check if the software system meets its required specifications.

It is the process of examining software with the intention of verifying whether the requirements have been met or not (Beizer, 2003).

In elementary terms, it can be described as an activity or procedure of running/executing a software application or system with the purpose of pinpointing any errors, bugs, gaps, or missing/incomplete requirements that were not present in the initial requirements.

The definition of testing according to the ANSI/IEEE 1059 standard is (Singh and Singh, 2012):

“A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item.”

The exercise of testing is meant to point out differences between the expected results and the actual results that could help understand or figure out shortcomings or inconsistencies in the requirements to begin with. In software, testing, in broader terms, is a manual or automated way of validating the requirement specifications of the system under consideration.

Testing can be called a process of affirming that the software product that was manufactured by an entity is a product of high quality and also to ensure that the final product is working correctly as per the specifications and satisfying all of the customer's demands or needs.

In short, the purpose of testing can be classified or categorized into the following points:

- Verification and validation of the requirements and the quality of the software;
- Error, bug, defect detection in order to find issues in the system with the intention of finding solutions to the detected problems.

Assessment of differences between the given input and expected output is done in this process.

The features of a software system are analyzed and examined and the quality of the software is determined by means of different testing methodologies.

3.2. BASIC TERMINOLOGIES IN SOFTWARE TESTING

1. **Error:** It is usually a mistake or a wrongdoing on the part of the engineer or the developer which in most cases is a misunderstanding of a particular requirement or of the design specifications.
2. **Fault:** It is generally referred to as the display/detection of an ‘error’ in the code of the software. It is often called or labeled as a “bug.”
3. **Failure:** It is defined as incorrect behavior or incorrect output that is the result of execution of a fault in the system. When a code containing a fault is executed, it results in a failure of some kind. The severity of a failure can range from a simple non-functioning of a particular part of the application to a complete system failure such as a crash. Any type of failure can occur at different times during the execution of the software application; some occur instantly and some occur quite later in the process of execution of the software.
4. **Testing:** It can be thought of a way to manifest or reveal ‘failures’ in the software application or systems.
5. **Debugging:** It is a process that aims to find an association between ‘failures’ and ‘faults.’ The aim of this process is to eliminate or remove the faults that cause failures from the software system or software application.
6. **Verification:** It is a process of ensuring that the software system meets the specified requirements imposed at the start of the development phase. It helps check whether the software is being built or developed properly (Kubde and Sable, 2014). It serves to answer the question-“Are you building it right?” It serves to confirm the functionalities of the software system or application. The software engineering standard recognized as IEEE-STD-610 provides the following definition of verification (View, 1990):

“A test of a system to prove that it meets all its specified requirements at a particular stage of its development.”

This part is usually done at the development/implementation level by the development team. The key component in the definition is that the verification is done at a particular stage of development. This means that the activity of verification is conducted during the implementation of the software product to make sure that the software adheres to the requirement specifications.

7. **Validation:** It is the process or activity of confirming that the software system does what it was supposed to in terms of functionality and behavior. It is the step of validation wherein the end result/final product is executed and tested against the expected output. This procedure serves to answer the question-“Are you building the right thing?”

As per the IEEE-STD-610, the definition of validation is (View, 1990):

“An activity that ensures that an end product stakeholder’s true needs and expectations are met.”

It helps verify if the software meets the expectations of the customer/client. Validation is a process that primarily concentrates on safeguarding the stakeholder’s interests. It is way to ensure that the customers or stakeholders get the product that they needed. The validation task does not focus on how the development was done and simply focuses on the end result and tries to establish that the needs of the client have been successfully met. This task is mostly done by a team of testers and it is performed on the actual product.

3.3. THE IMPORTANCE OF TESTING SOFTWARE

In this section of the book, we shall take a look at the use and the value of testing in software. We shall answer questions such as-Why is testing needed? Why is it important? What is the value added by testing in software?

As we have seen in chapter 2, as the complexity of software rose, so did the need for effective testing. Several strategies and methods of testing have been proposed since. The need for testing originally stemmed from a need to prove that a system works. The need to prove the validity of software led to development of a test case which was followed by a test suite. The initial implementation of testing showed the software community the real benefits of testing the software systems due to which it gained a lot of following. Now, testing has morphed into a discipline of its own.

There exist a plethora of reasons that describe the need of testing in the field of software. We shall now take a look at few arguments in favor of software testing:

1. **Better Software Control Process:** Testing serves as a method of effectively managing and tracking the understanding of the software requirements during the lifecycle of the software. The process or the cycle of software development should always be monitored and at each phase, one should check if the software

matches with the previously established requirements or not. Software testing helps find errors and faults that were introduced in the phase of development. It helps find errors early on so that the fewer issues are found later on after the software product is complete.

For instance, there is a chance that a programmer or developer makes an error in code during implementation for one of the following reasons:

- Lack of experience;
- Lack of programming knowledge or syntax;
- Lack of understanding of the software requirement;
- Insufficient subject matter experience;
- Incorrect implementation of the algorithm;
- Incorrect implementation of the algorithm due to complex logic;
- Simple human error.

Software testing helps identify simple errors or omissions or grammatical errors in the code. Once identified, bugs can be fixed at an early stage. Once fixed, the bugs can be verified again. This iterative process helps in effectively managing the software development and helps understand potential areas of complexity in the requirements. Also, it helps maintain a record of the issues faced, estimate risks and help save developers from additional tasks that arise when code isn't tested beforehand.

2. **Cost Effectiveness:** Testing is an excellent method of saving money of the investors or stakeholders of the software. It is a cost effective approach in the long term. It has been proven time and again that software projects having testing as an important phase in their cycle are more successful; than projects that do not have it. This is because the process of development of software development is a long process and is made up of several stages and if errors or bugs are captured in the initial stages, the cost of fixing them is significantly low. Hence, testing must be done at the earliest possible moment. Testing is an investment that will benefit the project in terms of its budget and quality.

For instance, consider a banking application that allows you to send money at better rates and fast. Let us imagine that a user wishes to open a new account with the bank and he clicks on the 'Register' button. Let us assume that there is an issue in the implementation of this functionality and when a user clicks on the button, there is an error message that pops up for

no reason. The error message is possibly a result of incorrect implementation of the code behind the ‘register’ button. But, due to the problem, the user is unable to register to the site. He is unable to make use of the product. Also, not just one user, all new users cannot register for an account on the application. The business is at a loss due to this. They already lost money as no new transactions are being made due to the error that is present. This issue is faced at the onset or the beginning when a potential new user wishes to sign up. The new user faces this problem at the start and he may be deterred by the problem. Each user that does not register to the application because of this problem contributes to monetary loss for the business. Moreover, such problems may encourage users to look for similar applications that provide the same functionality. They probably would dissuade their acquaintances and friends from using the application as well. The simple problem could lead to a loss of users that will probably never return to using the application because of the bad experience they had while using it.

Now, if the feature of registration had been tested during the development phase, the issue with the button would have become apparent when someone tried to use the ‘Register’ button. Therefore, the process of testing in the long run helps save both time and money because potential issues, bugs, and faults are usually resolved before they could escalate. If testing is done correctly, the cost of maintenance is lower as well thereby saving money in the end. The cost paid to test is worth the gains achieved by the success of the product which is made possible by software testing. In simpler words, it is a cost-effective process.

3. **Customer Satisfaction:** The penultimate intention of a software product owner is to provide the best level of customer satisfaction. The customer is a crucial part of any product as he is the one that is using it and paying for the product. Most providers of software products wish to gain the customer’s favor by giving them the best experience possible. One of the reasons for testing software systems and applications is to give the user a good experience when using the product. The users of the product end up deciding the value of the product in the market and hence testing helps deliver a good user experience. Also, if the delivered product is good, its value in the market rises and helps gain long term clients. If the clients trust the quality of the product, the product gains loyal customers. To gain confidence of the client, the product’s value must be good, which helps achieve long term

client satisfaction. This is only possible with the implementation of testing during the development of the product. Based on his/her experience, a customer will surely inform his/her friends and acquaintances about the product thereby marketing it indirectly. This in turn will help the sales of the product in the market.

But, if the product does not function properly more than 40% of the time, it becomes difficult to gain the confidence of the customer. If the customer is given a product that glitches, his trust is not easy to obtain. Bad experiences with the product are a surefire way to make sure that the customer never uses the application, or worse, it ensures that the customer deletes the application altogether. A bad experience with the software product can result in a customer telling others not to use the product. It is of paramount importance that the user's initial impression of the software is a good one, else the user is highly likely to go look for another similar product elsewhere. The market contains a sea of products that are available to the customers and hence establishing and maintaining trust is necessary and vital to the life of the software system or application. If the client thinks that the software product is unreliable, it is easy for him to switch vendors and choose a competitor's product. Hence, it is essential to ensure that the customer finds the organization or vendor reliable and that their level of satisfaction in the software system or application is always maintained (Oza et al., 2006). Trust can be established if sufficient investment in testing is made during the product development phase as adding value to the product helps the product stand out to the customers.

Additionally, monetary losses can be incurred if the customer's needs are not met. Certain projects are subject to strict quality and timeline contracts. If the product lacks in quality or if the deadlines are not respected, the vendor may have to pay heavy fines due to non-compliance. But, if proper testing had been conducted, there is a high chance that the problems would have been detected early on and probably no delays would have been encountered. In such types of projects, software testing could prevent financial losses for the vendors and help them produce better results.

4. **Product Quality:** So that the vision of the software product comes to life, the software should be able to work and execute as planned. Hence, the product requirements must be properly followed so as to obtain the desired results for the product which in turn provide the desired results for the customer. A software product must do what it is supposed to. A software product in the end is generally provided as a service to the users and therefore

it is imperative that the product brings forth the value that was promised to the customer. The product must work properly and efficiently in order to ensure the best customer experience. A principal precursor to instilling customer confidence is the quality of the software product provided. As discussed in the previous point, the customer's experience directly dictates the success of the software product and a good customer experience can only be achieved if the product is of a certain quality. Hence, it is considerably essential that the quality of the product is assured. If a good quality product is delivered to the customer, the customer's trust is gained in the vendor and his product. The customer, if convinced by the product will trust the vendor in the future as well. Hence, testing must be performed so that the requirements can be validated and the functioning of the product can be tested. Repeated tests performed on the software product aids in providing a higher quality product.

For instance, let's consider the development of a simple banking software application. The testing process would include checking and testing every part of the application. The graphics would be checked, the functionality would be checked, the alignment of the web pages would be checked, the menus would be checked, standard messages and errors would be checked and so on. Once the test team discovers issues in their first round, they would do another round to check that the issues have been fixed properly and that no new errors occur. It is a good thing that issues are found before the product has been released to the market as the quality can then be ensured. The product can be verified and validated before it is marketed and this can only be done with the help of testing.

Furthermore, in order to provide a software product of higher quality, rigorous testing must be done. This is because in depth testing will not only validates the software requirements, it will also ensure that unidentifiable or tricky issues are identified and fixed. Additionally, if a high quality product is delivered, the results of use of this software product are better and more reliable. Also, such products that have been well tested produce significantly accurate results. The better the quality of the product, the easier its maintenance is post deployment. This is because a good quality software product produces fewer errors which incur lower maintenance costs thereby reducing monetary spending. Fewer defects are observed in products that are of good quality which in turn also reduces the effort required in terms of maintenance and manpower which in turn reduces costs.

5. **Better Maintenance of the Product:** Once a product has been delivered in the market, it is handed over to maintenance. Any issues or defects that arise after the product is in the market are handled by the maintenance team. If the product has been well tested and is of good quality, the number of defects is less and the maintenance team's job is reduced.

Generally, post deployment teams are not too large due to budgetary constraints. Hence, if a lot of defects are found post deployment, there is a chance of the maintenance team becoming saturated with defects. This could mean that they would not be able to solve all the defects coming their way in time which could lead to loss of customer satisfaction. Also, if the product has a large amount of bugs in production, there could arise a need for more resources to help resolve the defects which could lead to addition financial spending which would be detrimental to the overall gain of the production. Hence, it is of utmost importance that a software product be tested thoroughly in order to ensure that fewer defects are uncovered in the post deployment period. A product that ensures its quality is more likely to have a higher success rate and create only minor issues after it has been released in the market.

6. **Better Product Performance:** As seen in the previous point, only a product that has been well tested produces good results. In order for a product to perform well on the market, it needs to be of good quality hence making testing a necessity. Testing is recommended in order to produce a software product that is efficient in nature. This is because one of the simplest and most straightforward ways to create a software system that performs well is to test it properly. A product will perform well if it functions correctly and produces fewer defects. Therefore, it is of paramount importance that the software application should not result into any big issues as it can be costly in the future. Any major defects found after deployment of the software product or in the final stages of the cycle of development can result in a lot of financial loss, loss of time and manpower loss.

Competent testing is an excellent way of making sure that defects, bugs, faults, and errors are detected in the early stages of the development life cycle of the software product or application (Devi, 2012). The reason for starting the testing from the beginning is also to avoid errors that could cost a lot in terms of development effort. When a software system or application is tested from the early stages and if there are defects or bugs that expose problems in the software requirements, they are easy to fix.

However, if defects related to the software requirements or the software design are identified later in the software life cycle, there is a high probability that those bugs are expensive to fix as it may need re-designing, re-implementation, and re-testing of the entire application either from scratch or from an earlier phase in the life cycle. If this happens, a lot of time is wasted notwithstanding the human effort that would be required to redo all the work. Furthermore, as more work needs to be done to finish the development, the vendor may need to postpone the launch of the application or maybe hire more staff that could help in the second attempt causing a lot of loss in terms of capital, time, and effort. A product that is well tested eliminates issues of this kind in the early stages itself and executes it the correct fashion as well. As the product goes through repeated tests and checks, it gets validated throughout its lifecycle thereby increasing its chances of producing accurate results. The more the product is tested, the more it is analyzed and understood. It can be said that a product that is well-tested performs well. So, it is imperative that the product's performance is good so as to ensure the overall well being of the product, the development team, the customers and the organization or vendor. It is necessary to deliver a product that achieves its desired result and performs well which can only be achieved with the help of testing.

7. **Business:** The market is a crucial place where all software systems and applications are introduced to the public. The organization offering the product needs the software product to do well in order to stay afloat. One surefire way to ensure that the product does well is to test it correctly. Testing would ensure the quality of the software product which in turn would build the customer's trust which eventually would make sure that the product sells.

Customers or users are prone to avoiding software systems or applications that has bugs in it. They do not look too favorably on applications that glitch a lot. As discussed previously, customers are wary of adopting a software system or application if they are unhappy or unsatisfied with the performance and the stability of the application. Therefore, if the product is disliked by the customer, there is high chance of failure of the product. This could result in significant losses in terms of finance for the company selling the product. They may end up incurring debt and may need to shut down their operations in the worst case scenario. This may not hold true for all the companies but it could happen if the organization is smaller.

For instance, in case of a startup organization that develops a single product, failure may not be an option. Their entire business would be dependent on the success/failure of that software product. If such an organization produces just one product and if this product is of poor quality which is rejected by the customers, it may not do well in the market. People or customers may be unwilling to adopt the software product and lead to the product failing miserably. In this case, the startup would definitely lose time, money, and the effort they invested in the rollout of the product. Financial loss would be the worst outcome for this organization and their business would probably not recover from this failure.

Hence, to avoid facing such a crisis, the only way of ensuring that the product performs well is to ensure its quality, its validity, and establishing the customer's trust. This can only be done with the help of testing. Testing would help the organization ensure the production of a product that is solid, of good quality and valid. It would help build trust and retain the customers which would help the business stay afloat. Testing would help increase the quality of the product significantly. In brief, testing is directly or indirectly required for the organization to stay in business.

Now that we have seen why testing is needed in software development, we will proceed to study its importance via a few brief case studies.

Importance of Testing: Case Studies

CONTENTS

4.1. Case 1: Ariane 5 Flight 501 Rocket Launch Failure	28
4.2. Case 2: Knight Capital Group Error	31
4.3. Case 3: Nasa's Mars Climate Orbiter.....	46
4.4. Case 4: Nissan Vehicle Recall	53
4.5. Case 5: Multidata Systems Disaster	58
4.6. Case 6: Therac-25 Medical Accelerator (1985).....	74
4.7. Case 7: Heathrow Terminal 5 Opening	107
4.8. Case 8: Cairns Hospital Software Glitch.....	108
4.9. Case 9: Uber Was Sued Due To A Software Bug.....	113

4.1. CASE 1: ARIANE 5 FLIGHT 501 ROCKET LAUNCH FAILURE

In the month of June in the year 1996, the novel rocket named Ariane 5 was launched on its inaugural flight. This rocket was carrying a payload of scientific satellites. This rocket was highly significant in terms of profitability for the European Space Agency because it was capable of carrying a payload that was much heavier than that of its predecessor series Ariane 4. Within thirty seven seconds of its planned launch sequence, unfortunately, the rocket veered off course, disintegrated, and exploded. All of this happened at an altitude of about 3700 m (Lions et al., 1996).

What happened was that the software present within its inertial navigation system shut down. This caused incorrect signals to be sent to the engines. This led to enormous amounts of stress to be placed upon the rocket and it started to veer off and break apart. The ground controllers, then, as per their programming launched the self-destructed protocol and the rocket including its payload was demolished. The end result was that 36.7 seconds into its first launch, the self-destruct safety plan was forcibly activated due to the computer failures, and the rocket shattered into a dazzling fireball.

The post-mortem analysis revealed that the European Space Agency's latest unmanned rocket that launched satellites, namely Ariane 5, reused functioning software from its ancestor, the Ariane 4 rocket. Dismally, the Ariane 5's engines that were quite fast, managed to exploit a bug that was unidentified in all of its previous models. In essence, what the guidance system was trying to do is insert a 64 bit number into a 16 bit memory space. The issue was induced by an error more known now called 'operand error' in the inertial reference system (abbreviated as 'SRI' as in French it is called 'Systèmes de Référence Inertielle') (Lions et al., 1996). This error occurred during the conversion of data in a subroutine in the software system from a 64-bit floating point integer to a 16-bit signed integer. One value that was generated by this routine was too large and couldn't be converted, thereby creating the Operand Error which was not handled by the program. The floating point number that had been converted had a value greater than what could be possibly represented by a 16-bit signed integer. So as per the specifications, the SRI halted and the second backup SRI also halted because of the same operand problem. Since at this point inertial guidance was not available and as the control system depended on it, incorrect signals were sent out that eventually lead to the disaster. This error in conversion of integers occurred in a sub-routine that had been 'reused' from Ariane 4,

whose trajectory varied from that of Ariane 5. The variable that contained the calculation of the horizontal bias (BH) which is a quantity linked to the horizontal velocity of the rocket, went out of 'planned' or 'acceptable' bounds. The difference in the case of Ariane 5 was that this value was planned for the Ariane 4 but catastrophic for Ariane 5. The overflow conditions caused as a result of this error crashed the primary and backup SRI systems as they were both executing the same code causing the Ariane 5 to veer off, move 90° in the incorrect path and eventually lead it to self-destruct mid-launch (Dowson, 1997).

The feature that caused the rocket to fail was a feature particular to the Ariane 4 specifications but was never removed in the design phase of Ariane 5. The error that was witnessed originated from a part of the software that exclusively did alignment of the inertial platform. This module of the software system was designed to calculate meaningful and valuable results only before take-off of the rocket. The moment the rocket is launched; this function is practically of no use. This function of alignment of the inertial platform was operational for 50 seconds after the start of the SRI's flight mode which occurred at 3 seconds for Ariane 5. Therefore, after take-off, this alignment continued for approximately 40 seconds more after flight mode was started. This sequence of events that led to the module being active for approximately 50 seconds was established as a requirement for the predecessor of Ariane 5 which is Ariane 4. It was not a requirement that was specific to the Ariane 5 rocket (Nuseibeh, 1997).

This feature (alignment feature) was not needed for Ariane 5 and a decision to keep the facility/feature despite the fact that it was not needed was made. Decisions were made to not remove the existing feature perhaps due to the risk of potential new bugs. Furthermore, overflow exceptions were not tested in order to avoid overloading the processor which was already heavily burdened. The Ariane 4 was a smaller vehicle and hence the initial acceleration followed by the build-up of the horizontal velocity was low. The value of this velocity could never reach a value that caused an overflow error. As the alignment feature was not required for Ariane 5, there were no specifications linked to this particular feature.

Consequently, as there were no requirements, no test cases or specific test suites were directed at that part of the software system. Furthermore, during the course of system testing, the IRS systems computer simulators were used and they did not generate any errors as there was no requirement to begin with.

The synopsis of the official report of the post-incident evaluation (Downson, 1997): *“This loss of information was due to specification and design errors in the software of the inertial reference system. The extensive reviews and tests carried out during the Ariane 5 development programme did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure.”*

The decision of incomplete analysis of the IRS and the decision to keep the component despite being unnecessary ended being a significantly costly decision. The expenditure that was incurred for development of Ariane 5 cost nearly 8 billion dollars. Furthermore, it was bearing a satellite payload whose value is estimated to be 500 million dollars at the time of the explosion. The Ariane 5 bug is notoriously considered to be one of the expensive defects in history.

A lesson that can be learnt through the study of this incident is that all the loss that was faced was probably easily preventable. The situation clearly is a reminder that certain assumptions made in previous code can potentially lead to disastrous consequences when practiced under different conditions. Any assumptions made when code is reused must be re-evaluated when the conditions change. Another crucial lesson that we learn is that the design and code of software systems must be reviewed during the development cycle of the software. In this case, there is a possibility that either the IRS software was not reviewed, or an assumption was made that since it worked before it would continue to work, or testing was not done on this component, or the system testing didn't reveal or adequately test the consequences of a system failure (Nuseibeh, 1997). An extra lesson that we glean from studying this event is that exceptions must be handled correctly. A default exception handling mechanism that results in complete shutdown of the system is hardly ideal. Another blunder from which one can learn from is that the system was designed in such a way that a single component failure caused the entire system to fail. Further, we gained insight into the importance of correctly testing a system for errors. There is a possibility that if proper tests would have been conducted on the system, if testing was given the importance that it is today, the redundancy of the alignment system could have come to light, and this incident perhaps was avoidable.

4.2. CASE 2: KNIGHT CAPITAL GROUP ERROR

Knight Capital Group was an American firm specializing in worldwide financial services that provided and engaged in several services such as electronic trading, market making for stocks, trading, and institutional sales (Davidson, 2012). In the year 2012, this organization was one of the largest traders in the United States equities and had a share in the New York Stock Exchange (NYSE) of around 17%. Knight also owned a considerable share on the Nasdaq Stock Market. On an average, their electronic trading group (ETG) handled over 3.3 billion of trades on a daily basis. They were responsible for trading over 21 billion dollars on a daily basis. The Knight Group was in business in the market for well over 17 years but a single software error nearly lead them to bankruptcy in less than an hour. Knight Group struggled to stay afloat after a software defect triggered a loss of around 460 million dollars in around 45 minutes.

On the first of August 2012, Knight Capital group successful deployed an update of their software to their working and in use production servers. Early morning at about 08:01 AM, their staff received exactly 97 notifications via email. These notifications stated that '*Power Peg*,' an internal system that was defunct and which hadn't been used since 2003 was not configured correctly (Saltapidas and Maghsood, 2018). In hindsight, this was the first sign that should have sounded the alarm bells in warning. These messages were sent through a channel that was not a channel where alerts of high priority were managed and the messages weren't checked out in real time (Buchanan, 2015). But, it was a proverbial smoking gun that could have saved the company had it been reviewed before the stock market opened.

The turns of events that followed next is nothing short of a catastrophic nightmare. At sharp 9:30 AM in the morning, the Knight Capital Group experienced a significant failure in the running of its system that routes orders for US equities. They received 212 orders from several retail clients and unknowingly or unwittingly performed thousands of orders every second into the stock exchange over a period of 45 minutes. The software executed a quantity of more than 4 million trades in around 154 stocks which totaled to a quantity of shares of 397 million. The malfunction caused the system to send thousands of orders per moment that would buy stocks at a high price and then sell them at a low price. They assumed a long position (net long) of around 3.5 billion dollars in 80 stocks and a short position (net short) of around 3.15 billion dollars in 74 stocks. The market got flooded with millions of trades that were unintended. The company lost more than 460

million dollars as a result of these incorrect and unwanted positions. Owing to these losses due to the faulty software system, the following day their stock price dropped by 75% as the CEO, its employees, their clients, and customers were trying to anticipate their next move (Buchanan, 2015).

Let us now take a look at the history of this group and the events that eventually led to that fateful day when the company ended up losing over 460 million dollars.

In 2011, the company was doing exceptionally well financially and its net worth was over 1.5 billion dollars and they earned a net income of over 115 billion dollars.

The company was growing and it had around 1400 people in its employ. The Knight Capital Group employed more than 100 software developers and had several offices in the United States, UK, China, Singapore, and Switzerland. By this point, the company made markets in US and European equities, it traded currencies, and it traded fixed income for its accounts that were proprietary.

Daily, it managed over 3.3 billion trades in US equities and handled trades that were valued at over 21 billion dollars.

The Knight Group's main customers were online brokerage firms and discount brokers such as TD Ameritrade, Scottrade, Vanguard, and E*Trade. Their competition included other large businesses that provided financial services such as Citigroup, Citadel, and UBS. But, the competitors of the Knight Capital Group had a distinct advantage as they had the capability of internalizing large amounts of trades in secret or in clandestine within their own limited markets or private markets that are shared among several stakeholders, also known as dark pools.

These so called dark pools have seen a steady rise in the markets gaining popularity over time. Starting from the year 2008, the quantity of stock trades in the United States that took place in 'private markets' or dark pools saw a rise from 15% to over 40%. This is shown in Figure 4.1 (Tabbaa, 2019):

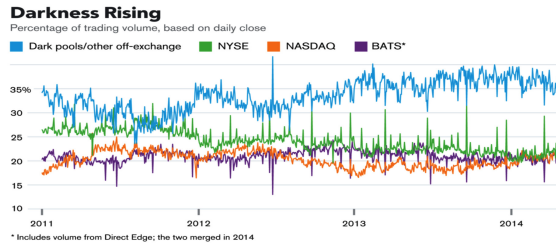


Figure 4.1. Rise in dark pools.

Source: Tabbaa (2019).

In the month of October, in 2011, the NYSE created and arranged a dark pool or a private pool of its own, called the retail liquidity program (RLP). The RLP created a private market of traders inside the NYSE that could perform transactions of trades anonymously for a cost of a fraction of pennies that was either more or less than the actual displayed bid and its offer prices, respectively (Bolger, 2012). This program was extremely controversial even inside the parent company of NYSE named NYSE Euronext. The CEO, of NYSE Euronext, Duncan Niederauer, had penned a public letter in the paper Financial Times that criticized such dark pools and blamed them for shifting “more and more information... outside the public view and excluded from the price discovery process” (Tabbaa, 2019).

The decision that was to be made by the SEC largely benefited large financial investors who would have the capacity of buying or selling large volumes of trades or shares with a certain level of anonymity and also without changing or moving the existing public markets. The downside to this decision was that it would flourish at the expense of several market making firms. But the SEC ruled in favor of this pool. They approved the RLP system in the early days of June 2012. As soon as the NYSE received approval from the SEC, they made an announcement where they confirmed that the program would go live on the 1st of August 2012. This meant that market making companies had less than 2 months to prepare for this significant market altering change and market makers were now faced with the dilemma of either joining the RLP or facing losses. The CEO of Knight Capital Group Thomas Joyce firmly asserted his desire to participate in the RLP market or dark pool in a bid to avoid financial losses. The reasoning made was that giving up the flow of orders in this pool would cause a

reduction in the profits and adapting to the RLP would be best possible option for saving the business.

The Knight Group was expanding its business and building new technology. As a part of this new effort, they came up with several strategies for improvement and renewal. They decided to deactivate or retire the older software systems and build newer technologies for trading. They built the smart market access routing system (SMARS) trading technology which had the potential to execute over thousands of trades per second and was also capable of comparing stock prices coming from a wide variety of different sites in mere fractions of a second. This system worked well, and the group was reaping the profits made by this system. But, the SEC's decision to approve the new dark pool proposed by the NYSE changed things drastically. As the SEC approved the new pool, things took a turn as the CEO of the group decided to participate in the RLP pool.

Now, there was barely a month left between the announcement of the SEC's approval and its scheduled go-live date, which meant that Knight had to update all of its software with the changes that would kick in on 1st August 2012. Their development team worked frantically to update their trading systems including the SMARS system. They scrambled to make the required modifications to this system so as to include the new RLP system. The SMARS system had an interesting feature where it received orders from upstream components in their trading platform and later as per their needs, based on factors such as price and liquidity, it send more 'child' or representative orders to their downstream components or external sites or venues for execution of the orders (Furrer, 2019).

As the development team made changes to the SMARS component, they came across some unused code. The code pertaining to RLP ended up replacing this unused code in a part that was functional and relevant to the order router. The old code that was replaced was once used for an old algorithm named 'Power Peg.' This algorithm hadn't been in use since 2003 by Knight. This algorithm named Power Peg was a simple test program developed early on that was programmed to buy stocks at a high price and sell them at a low price. This was a test program that was particularly designed in such a way so that it would move the prices of the stock from a higher value to a lower value so as to verify the reactions of its other trading algorithms in a controlled test environment. This 'Power Peg' algorithm was technically not supposed to be in production (Davidson, 2012). It was never meant to be used in a working production environment.

To begin with, there were several issues with the 'Power Peg' algorithm. Firstly, the code for this algorithm was present and remained in the code during the RLP deployment although it was not used. Secondly, in the rush to finish the updated modifications to the SMARS system, the RLP code that was newly added reused a flag that was earlier used in order to activate the code or functioning of 'Power Peg.' Basically, this flag would indicate or intimate the system to the fact that the component 'Power Peg' was activate. This flag meant that when it was set to 'yes' or 'true' it would mean that this component was activated and that the system should buy stocks at a higher price and sell them at a lower price. The intent behind this flag in the new updated version of RLP was that when this particular flag was set to 'true' or 'yes' the RLP unit would be activated and not Power Peg. The developers made the mistake of reusing the existing flag for the RLP unit. Now, what happened is that the reusing of this flag caused confusion in the code which ended being detrimental to the firm. This oversight or mistake meant that now the SMARS system thought that it was operating under test conditions and so the system now behaved the way it would in a test environment. The system carried out trades as fast as it could without thought about the spread values. Thirdly, the code had been refactored over the years without appropriate regression testing. After the RLP changes, the workflow of SMARS was disconnected from Power Peg. In the year 2005, an alteration or change was enacted to the code of the Power Peg system or unit which recklessly disabled the safety checks that would have easily prevented this catastrophic scenario. Yet, this particular update was nonetheless deployed in production system at that point of time. This was done notwithstanding the fact that no testing and effort was made to check if the Power Peg algorithm was still working or not. So, when the RLP changes were done, it wasn't clear whether the functionality of Power Peg was active or not. Lastly, there was an issue at the time of deployment of the new RLP code (Davidson, 2012). What happened was that one week before the planned go-live on 1st August, an engineer employed by Knight performed a manual deployment of the new updated RLP code to the SMARS servers. There were a total of 8 SMARS servers. But, the engineer committed an error and did not copy the updated RLP code to one of the 8 servers. He possibly forgot about it or unknowingly neglected to deploy or copy the modified code to the 8th server. This was an important deployment and it was not given the level of importance it deemed. The deployment team at Knight failed to have another engineer review the deployment which could have identified the error and avoided the catastrophe (Buchanan, 2015). Furthermore, the system did not

have any automated monitoring system that could have possibly raised an alert regarding the discrepancy in the code present on the servers. All these factors contributed to the malfunctioning of the system on the day of the go-live on 1st August 2012.

So now, on the day of the go-live, they received some emails at 8:01 AM saying that Power Peg was disabled which did not raise any flags and weren't propagated to the right people or checked in real time. This was an opportunity or a 'code smell' which could have helped identify the issue before the market opened. Then, when the market opened, Knight started getting or receiving stock orders at 9:30 AM from the dealers and their brokers as they normally would. As per the functioning of the system, these orders were distributed among the many servers the system had. Now, out of the 8 servers, 7 of the servers which contained the updated RLP code processed the requested order in a correct manner. But the 8th server with the old code did not work correctly. The orders received by this 8th server containing the Power Peg flag which had been reused and repurposed triggered a fault. Now, for this server, the Power Peg code was active because of the reused flag. This flag prompted the system to send orders continuously from each parent order. So, for each incoming 'parent' order, the system started sending multiple child orders non-stop without considering the number of stock market executions that Knight already had accepted from other trading sites or venues. The result of these executions was nothing short of catastrophic for the company. The defunct server processed 212 incoming parent orders. For each of these orders, several thousand child orders were sent to the NYSE's pool. Each of these orders followed the principle 'Buy high, sell low' as was designed by the 'Power Peg' algorithm. The situation devolved swiftly and rapidly from there. The trading algorithms began to push erratic trades into the market and moved over 150 different stocks and sent the market into spasms.

So, for the 212 trade orders that were received by the 8th server with the defective or flawed code belonging to the 'Power Peg' algorithm, the SMARS system sent several thousand orders that would buy stocks at a higher price and sell them at a lower price which resulted in over 4 million trade executions that were carried out by the SMARS system. Over 4 million executions were performed in 154 stocks for a quantity higher than 397 million shares in roughly 45 minutes. Out of these 154 stocks, 75 of them shook the prices in the market over 5% and these stocks comprised of over 20% of the volume of trades during those 45 minutes. For another 37 of these 154 stocks, the market prices jostled to higher than 10% and the Knight Group's trade

executions formed or was responsible for more than 50% of the current volume of trades (Bolger, 2012).

Now, the SEC had certain rules put in place that allowed the firms to cancel such trades, but there were several catches. These rules were established following the Flash Crash incident of 6th May 2010 where the DJIA ended up losing over 1000 points in the market in a matter of minutes. As a result, the SEC introduced regulatory rules that helped police and manage trading activities. The first rule was that circuit breakers must be put in place so as to stop the trading if it witnessed 'significant price fluctuation' of over 10% in a period of 5 minutes. Secondly, for the cancellation of trades, if the trading events involved between 5–20 stocks, the trades could only be cancelled if they were priced at a value that was at least 10% higher or different from the 'reference price.' The 'reference price' was the last sale price before the disruption in price. Further, for trading events concerning more than 20 stocks, the trades could be cancelled if and only if the deviation in the price was more than 30%. Thirdly, the Securities Exchange Act Rule C.F.R 240.15c3–5 rule went into immediate effect, which meant that the exchanges and broker-dealers were required to implement risk management controls to ensure integrity of their systems as well as executive review and certification of the controls (Bolger, 2012).

Now, in this case, when the price fluctuated, the second condition or rule could have been met. But, sadly, since the rule catered to price fluctuations and not volumes of trades, the rule was not activated. This was because the deviation caused by the Knight's servers maxed out at the 10% price threshold and hence the trading was not stopped. Only a few of the stocks managed to cross this threshold value of 10% meaning that the second rule was not activated. So, unfortunately the trades were not cancelled on their own. Soon, at 9:34 am a computer analyst working at the NYSE luckily noticed the trading volumes. He noticed the spike in the volume of trades which was almost double the regular level at that given time and managed to trace the raised levels back to the Knight Capital Group. Duncan Niederauer, the CEO of NYSE Euronext on learning the issue, tried to call Thomas Joyce, the CEO of Knight but could not reach him as Joyce was recovering from a knee surgery at home. But, the NYSE managed to get a hold of Knight's CIO (Chief Information Officer) who quickly assembled the best IT staff in the firm and asked them to investigate the issue. Knight did not have a documented procedure for such an incident, and they did not know how to respond to this situation. Generally, most firms that competed in trades would contain a kill switch of some sort that was present in their systems, but

this wasn't the case. No documentation was available to guide them either. The IT engineers continued to scramble in the dark for another 20 minutes or so and concluded that the newly deployed code was the root cause of the erratic behavior. The team reasoned that as the 'previous' code (before RLP changes) worked, they should revert the systems to that working version of the code. This was another mistake. They unwittingly reverted back to the version which was running on the malfunctioning 8th server and ended up with the bad version on all of its 8 servers. This deteriorated the situation as now all the 8 SMARS servers had the flawed code of the Power Peg in live action on production. The Power Peg code was now activated by the reused and repurposed flag and began executing the ordered uncontrollably. Soon, the engineers realized that this decision was more detrimental than the previous situation. Finally, at 9:58 AM the engineers eventually figured out the root cause of the errors and finally shut down SMARS on all of its 8 servers. But, it was a little too late and the damage had already been done by then. They had involuntarily executed 4 million trades in 154 stocks which totaled to over 397 million shares. They assumed a net long place valued at 3.5 billion dollars in 80 stocks and a net short place valued at 3.15 billion in 74 stocks. Their stock took a sharp dive by 33% the same day and the loss suffered by its traders amounted to more than 460 million dollars (Davidson, 2012).

This is shown below (Tabbaa, 2019):



The aftermath of this fateful event was equally chaotic. Knight Capital Group had to raise a lot of money really fast. Their investors such as TD Ameritrade and other clients publicly announced that they would continue to work with Knight. But, Knight had lost over 460 million dollars and they were trying their best to raise money as they didn't have enough cash to settle their liabilities. The weekend of 5th August, Knight managed to

raise over 400 million dollars from several investors which was handled by Jeffries Investment bank. The new investors got 70% control of the company and Knight Capital Group had to accept three new board members namely Martin Brand from Blackstone, Matthew Nimetz of General Atlantic, and Fred Tomczyk of TD Ameritrade (Davidson, 2012). This deal hampered the shareholders of Knight Group but the alternative was bankruptcy and they wanted to avoid it at all costs. Eventually, the board members met at the end of the year to finalize the takeover offers received and was acquired by one of its rivals Getco LLC for the cost of 3.7 dollars per share which was an ample premium to the initial amounts paid by the investors. Eventually, the merger with Getco was completed in 2013 and it was renamed to KCG Holdings and Thomas Joyce resigned as the CEO.

In addition to the merger, the company was found guilty of violation of the SEC rules and policies and fined an additional 12 million dollars in 2012. As per the SEC guidelines, Knight flouted several of its risk mitigation and management controls and policies. The third rule of the SEC that was mentioned earlier stated that broker dealers were required to *“appropriately control the risks associated with market access, so as not to jeopardize their own financial condition, that of other market participants, the integrity of trading on the securities markets, and the stability of the financial system”* (Davidson, 2012).

The investigation into the events of 1st August conducted by the SEC led them to conclude that the Knight group violated the followed subsections of the SEC Rule 15c3–5 as follows: (Davidson, 2012):

- i. Subsection b which stated that such firms must *“establish, document, and maintain a system of risk management controls and supervisory procedures reasonably designed to manage the financial, regulatory, and other risks”* of having market access (Davidson, 2012).
- ii. Subsection c which required the dealers or brokers to have standardized controls of risk management combined with supervisory procedures that would be designed in a reasonable manner so as to prevent the passage of erroneous or flawed orders that exceed the previously set credit threshold and capital threshold values in the aggregated vales for each customer and the broker/dealer.
- iii. Subsection e required that the CEO of firms to review and certify

that the controls and procedures comply with subsections b and c of the stated rule.

As noticed by the SEC, the Knight Group was in violation of all these rules. There were no pre-defined controls that prevented the erroneous entries from being entered into the market. There were no pre-defined and set thresholds for the capital in the aggregate was defined. Further, there were no controls and supervisory techniques or processes for the deployment process. Also, there was a lack of formalized or even well-written response procedures in case of an incident. Furthermore, there was a less than sufficient written or formalized characterization of its risk management level controls at an enterprise level. The company and its board failed to manage its enterprise risks.

The CEO's certification was a farce as it was incapable of handling or providing proper handling of incidents. The firm had certain "processes" that were set up to comply with this SEC rule but in reality, the implemented 'processes' such as the controls and the procedures did not actually comply with the SEC rule. Moreover, the Knight group did not have any circuit breakers that were installed which are a quite common practice for most firms of this type. The violation that caused them to be fined 12 million dollars is when the firm performed millions of orders of the type 'short sale,' they failed to mark these orders as a short sale and also failed to borrow the underlying security required for such a sale. This action was in direct violation of the SEC Rules 200(g) and 203 (b) which is a part of the Regulation SHO. Hence, the firm was fined 12 million dollars for these violations that were payable to the US Treasury (Davidson, 2012).

The following vital lessons can be gleaned from this case study:

1. **Proper Deployment Guidelines must be put in Place:** One of the main reasons that Knight Group suffered such a huge amount of loss was that their deployment was not done efficiently. First of all, manual deployment was done. Even though manual deployment was still carried out at that time, it is usually supervised, and strict guidelines are put in place to ensure that it is done correctly (Carzaniga et al., 1998). The deployment process that was set up by Knight was highly inadequate and the engineers were not fully prepared for the risk that the software was exposed to. Additionally, the manual deployment was not supervised by anyone and no verification of the deployment was done beforehand. So, until the day of the go-live, no flags were raised because the deployment was never validated by anyone.

The error made by the engineer was left uncaught and resulted in the company almost going bankrupt.

Additionally, no step by step guide was provided to the deployment engineer. This could be a reason that led him to forget to deploy the code on the 8th server. The chances of someone making such an error with a proper checklist printed out and kept right in front of them are extremely rare. Perhaps if the engineer had some sort of written instructions along with a checklist, this error could have been completely avoided.

2. Risk Management should be done: Risk management is an essential and vital component of any organization and it has special importance in the financial industry. The SEC defines good risk management practices as practices that include the following (Chapman, 2011):

- Quality assurance;
- Continuous improvement;
- Controlled UAT;
- Regular reviews for compliance with the rules and regulations;
- Independent audit processes;
- Process measuring, management, and control;
- Technology governance capable of preventing software malfunctions, system errors and failures and service outages;
- Prompt, effective response that leads to effective risk mitigation.

Although Knight had certain order controls in place in other systems, it did not have sufficient controls in the SMARS system. It did not do a comparison of the orders entering SMARS and the orders exiting the system. Their principal risk management or monitoring tool was called PMON and it was a post-execution monitoring system that monitored the positions. The day the market opened; some personnel observed that a large volume of positions was present in a special type of account called 33. This account was used to temporarily hold multiple types of positions that were a result of Knight's order executions. It also stored positions that were received back from the markets that Knight's systems could not match to a possibly unfilled quantity of a particular parent order. This account had a 2 million dollar gross limit, but unfortunately, this limit check was not synced or programmed to any automated controls that managed the financial exposure of the company. This monitoring system was entirely dependent on human monitoring and it was not programmed to generate any automated alerts of

any sort. It was not capable of highlighting the account exposure based on the 2 million dollar limit. It was incapable of raising a flag when this limit had been exceeded. If Knight had added some automation to their monitoring system, they might have caught the bug earlier. Even if these values were being monitored in real time and evaluated, there is slight chance that the failure could have been avoided. In addition to an inadequate monitoring system, the risk mitigation processes were not well defined or to be precise, they were 'absent.' No proper guidelines were defined or documented in case of a failure. No documents or procedures were available on the risk management subject. Generally, in case of incidents, there is an incident response guideline that must be present. Usually, such a document in its bare minimum capacity at least asks the team to shut down the system when a critical incident occurs. However, the reaction of the incident response team was different as no such document was made available to them. They did not know what to do when an incident occurs so they ended up performing a 'rollback' which was a blunder. Moreover, Knight failed to implement circuit breakers in their system. This was the first rule outlined by the SEC and it is a standard followed by most organizations providing financial services (Davidson, 2012). So, when the system started failing, they were unable to automatically stop the trades by using them.

3. Detailed Testing of the Project must be done: Needless to say, testing must be carried out whenever there are changes in the code. Different types of testing (unit, integration, regression, system) must be done in order to verify that the system is still intact and that it does what it is supposed to. These tests will be covered in the next chapter.

Knight refactored their code several times without testing their functionality. They did not perform sufficient regression testing to check whether the 'Power Peg' code was still working or not. The algorithm was never tested and hence the fatal bug was never exposed. Furthermore, had they performed proper system testing, unit testing or integration testing, the inconsistencies would have most definitely surfaced and led them to the discovery of the dead code still present in their new version. If they would have implemented a test plan or would have automated testing, there was a higher probability of the bug being discovered during the development phase. Sadly, sufficient testing wasn't done and they ended up paying a hefty price for their negligence.

4. Proper Version Control and Documentation must be maintained: One of the downfalls of the Knight group was the dead code that was still a part of their systems. A mistake that was made by the developers was that they let the dead code stay in

the system despite knowing that it was not being used. Although the 'Power Peg' algorithm hadn't been used since the year 2003, it was still kept in production which was a mistake (Davidson, 2012). Code that is not being used must be pruned. It is a bad software practice. Versioning of the code must be done if old versions need to be saved. One of the lessons we take from this incident is that dead code must never be deployed into production. Another lesson we learned was that code must not be repurposed without properly testing and analyzing the impacts of repurposing the code. In Knight's case, the flag that was reused from 'Power Peg' ended up running the power peg algorithm which was still present in the system in production despite it being 'dead code.' This oversight amplified the losses faced by the company.

5. **Proper Time and Project Management must be done:** A reason for the failure of the Knight Group was their dismal project management practices. They failed to manage their time and their project leadership was inadequate and inefficient. Due to the lack of efficient time and project management practices, they failed to deliver the RLP solution to its customers. An error on the part of Knight was that they tried to do too much too soon. Due to the one month deadline that loomed before them, they implemented aggressive deployment schedules that led to their failure. They scrambled to include the RLP related changes in their code in a scarce amount of time—a month, which was an unwise decision. They forced their developers to implement critical high impact changes to their trading algorithms in less than 30 days which was a catastrophic decision. They pushed for an extremely aggressive delivery schedule which led to the development team making several mistakes in the code. The tight schedule did not give the development team enough time to perform proper testing. This led them to delivering a code of sub-standard quality which was not properly tested and vetted. The reason for pushing such extreme deadlines was fear of financial loss due to the new 'dark pool' called RLP. In a bid to save money, the managers, the CEO and the CIO pushed the IT team to deliver a major change that handled millions of dollars in the stock market in just a month and this decision backfired spectacularly. Even for projects that are simple, thirty days is insufficient to plan, implement, test, and deploy major changes to an algorithm that impacts the entire

system. One can imagine the stress experienced by the developers during that one month period where they were responsible for updating an algorithm that was used to make markets and perform trades that were worth over billions of dollars. The decision to aggressively push for the strict deadline was a naïve, impulsive, and reckless to say the least. Knight's managers, CIO, and CEO should probably have delivered the RLP changes in phases which would have given them ample time to ensure the quality of their product. They could have proposed an alternative strategy to their investors until the new RLP modifications were done. Instead of trying to deliver everything at once, they could have rolled it out which probably would have saved them from the huge financial loss they ended up incurring. What we learn from this is that any software project must have proper time and project management in place so as to ensure the quality of the product. Aggressive deadlines and changes must be reviewed and alternatives must be provided in case these deadlines cannot be met. Especially in case of critical applications, a cost benefit analysis must be done before embarking upon such risky time sensitive projects. Management should react in a way that prizes quality over quantity.

6. **Code Reviews must be conducted:** Another prime reason for the failure of Knight's RLP deployment was its code. The code included the Power Peg algorithm although it was defunct and not in use. Ideally, the power peg code should have been removed from the system's codebase completely. But still, it remained in the code. The practice of keeping such code which is considered to be 'dead' is a bad software development practice. The practice of keeping 'dead code' is bad but is generally common in software systems that are massive and that have been maintained for years. As the SMARS was an old system, the dead code remained alive and executable when the RLP algorithm was deployed even though it was not in use. Secondly, the flag which came from the 'Power Peg' module and enabled it and which was reused for the activation of the RLP component was a bad software choice (Davidson, 2012). It was a grave error that did not have any benefit and added no value to the quality of the code. Such type of repurposing of variables or configuration flags more often creates chaos and confusion. In this case, it ended up being the

downfall of the Knight Group which caused them to lose 460 million dollars.

If there had been a code review, the review would have easily exposed the issues in the code. Code review is a methodical examination of the source code of a software system. The aim is to find mistakes in the code that were overlooked during the software development thereby improving the quality of the software. It is also called peer review (Robillard et al., 2004). Generally, a review of the code and the artifacts that are supposed to be deployed should sniff out faults and raise queries that end up preventing and resolving bugs, faults in the code before they are deployed into a production environment. If a proper review is done, most bad coding choices are exposed and faults are prevented well in advance.

Had there been a proper code review, the repurposed flag would have been exposed and lead to the discovery of the dead code. The review may have shown the presence of the flag and its use in another part of the code which may have prevented the flag from being delivered in production. There is a fair chance that the dead code would have been noticed and removed before it was delivered had a proper peer review been conducted by the Knight's IT team. Usually, such reviews are conducted in different forms such as formal inspections, pair programming, walkthroughs, etc.

7. **Deployments should be automated if Possible:** Another argument in favor of the failure of Knight's RLP deployment on the 1st of August 2012 was their deployment process. Firstly, they deployed their code a week before the go-live. Secondly, it was a manual deployment. As we have seen, there was an error in the deployment of the RLP update that was developed by their IT team. Further, there was a lack of sufficient documentation pertaining to the deployment given to the engineers. Also, the deployment that was done manually was not reviewed by another member of their team. Had it been reviewed; the fateful event could have been avoided.

As software is becoming more complex day by day, so is its process of deployment. Due to the complex nature of software systems, they tend to consist of several components and deployment artifacts which make it difficult and tedious to deploy manually. The minute the deployment process starts relying on human beings, there is an inherent risk involved. When deployment involves reading, analyzing, and following written or dictated instructions, there is always a small chance of error. Nobody is

perfect and human beings are capable of errors and mistakes. There could be a misunderstanding of the instructions, the interpretation may be false, or the execution of the instructions may be incorrect. Especially if just one person is given the job, there is no supervision and he can make mistakes unknowingly. Hence, deployment in software has shifted to become more automated.

Generally, deployment is a repetitive process and can be easily automated. In order to avoid human error and to increase efficiency, deployments are now being automated. If the Knight Group had implemented an automated system of deployment, the new code would have been delivered on all the 8 servers instead of just 7 servers. The defunct code would never have executed and caused losses to the tune of 460 million dollars. An automated system would have correctly deployed the RLP code on all the SMARS servers and it would have also checked the configurations. If the deployment was automated, a simulation could have been performed to check if the deployment succeeded. A test run could have been performed which would have exposed any faults within the deployment artifacts. This could have saved them from their terrible fate.

Had Knight managed to write clean code, review their code, ensured proper project planning and management, implemented effective risk mitigation practices, implemented automation in their tests, implemented automation in their deployments, they could have potentially avoided the entire nightmarish situation.

4.3. CASE 3: NASA'S MARS CLIMATE ORBITER

The Mars Climate Orbiter was a spacecraft weighing 338-kilogram that was launched by NASA on the 11 of December, 1998. It was a robotic spacecraft that was supposed to study the Martian climate, the atmosphere on Mars and changes in the surface of Mars. It was phase 2 of NASA's program for exploration of MARS. The previous phase or the first phase had already launched 2 spacecrafts namely the MARS Global Surveyor and the Mars Pathfinder in the year 1996. These spaceships were launched to assess MARS and take pictures of the planet and to look for water on MARS. The MARS missions were designed and crafted to aid scientists and researchers comprehend the planet and understand the history of the planet and investigate the possibility of life in the past history of the planet. In addition, its function was to act as the communications relay in the Mars Surveyor '98 program for the Mars Polar Lander (Llyod and Writer, 1999).

But, on 23rd September 1999, this spacecraft, on its mission to Mars was lost in space forever. NASA lost all signals from the Space Orbiter. The spacecraft successfully completed 9 months on its mission, but as it was moving into an orbit around Mars, the communications stopped completely. What had transpired was that the Climate Orbiter's trajectory was wrong. It was supposed to simply lightly enter into the orbit of Mars, but in reality, it was orbiting at an altitude much lower than what it supposed to be orbiting. It was at 170 kms below the altitude that it was supposed to be. The ground software was responsible for miscalculating the required or anticipated trajectory. As the satellite orbited too close to the surface of Mars, it was deduced that the heat and the drag from the atmospheric pressure mostly likely destroyed the Orbiter. The miscalculation in the trajectory sent the Orbiter too close to the surface of Mars which was not the intention of the Mars Climate Orbiter. It was supposed to be orbiting at a safe distance from the surface of Mars and serve as a relay, not land on it.

After the signal was lost, the team began analyzing and investigating the failure. To understand the cause of the failure, one must know that most spacecrafts make use of thrusters to control the trajectory of the spacecraft. Reaction wheels help control the altitude and orientation of the spacecraft. The reaction wheels generate excess momentum, a control called angular momentum desaturation (AMD) is used to de-spin the wheels and counter the change in momentum with a thruster burn (NASA Safety Center, 2009). Each time an AMD operation takes place, the software on board the space probe sends data to the ground control software which will then calculate the next position of the Orbiter. Now, in the initial few months of the mission, the ground software compelled the team handling the navigation to manage only with the help of emails. They were dependent on emails from the contractor to be able to track the progress of the space probe. The issue was that the navigation system was designed by a sub-contractor and hence communication was limited and any anomalies weren't well communicated or overlooked. The team who managed the ground control software had limited access to accurately gauge the progress of the probe (NASA Safety Center, 2009).

The Orbiter did not behave the way it was supposed to and the failure was not expected by NASA engineers. The failure of the probe initially baffled the engineers of the investigative team for a little while. But soon, after a closer analysis, it was discovered that sub-contractor who had designed the navigation system of the Climate Orbiter had made use of imperial units of measurement instead of the metric system units which was the NASA

standard. It became clear that this sub-contractor had simply failed to perform a simple metric conversion from English units to metric units. The investigation revealed that the ground software system was calculating values using the English Units whereas the onboard system was using metric based units. So, based on this fatal error, the ground software was calculating the 'new' trajectory values in 'pounds force' instead of 'Newtons.' The values were incorrectly being calculated in English units. So, each new calculated value was now higher by a factor of 4.45. Now each time an AMD event occurred; the force of the thrusters was increased by this difference. This kept happening for each successive AMD event. Every consecutive such event was introducing an even larger value for the 'new' position of the space probe. Each event of this type kept adding a larger offset to the originally scheduled trajectory as the ground navigation software kept using the new values. Each event of the type AMD ended moving the Orbiter further away from its target location. The basic flaw in the system was that the navigation software on ground was using English Units whereas the on-board and all other calculations that were being performed were in metric units (NASA Safety Center, 2009). As communication between the teams was not extremely detailed, differences or discrepancies in the values calculated were not noticed and slipped through the cracks. Every piece of software involved in this mission was under the assumption that metric units of measurement were being used.

The end result was that the space probe costing over 125 million dollars orbited too close to Mars and its atmosphere and couldn't withstand the atmospheric conditions caused by the heat and drag and ended being annihilated in space by the Martian atmosphere. The lapse in judgment cost NASA 125 million dollars as the Orbiter flew very close to the surface of Mars after its failed attempt at stabilizing its orbit at an extremely low altitude. At this low of an altitude, the probe did not stand a chance of survival. The controllers of the mission believe that the orbiter went into the Mars' atmosphere where the stress caused by the heat and drag of the Martian atmosphere severely crippled the probe and its communication systems leaving the probe hurtling through space. It is believed that the probe is lost in space in an orbit around the sun (NASA Safety Center, 2009).

The investigation revealed that proper verification and validation of the system did not occur. The discrepancy in the units system was not identified during the testing or the validation phase of the satellite. It is evident that the main reason for the mishap was due to contractor's oversight as it was the contractor that provided the navigation software. The contractor failed

to deliver the system that he had promised to supply. The requirements of the navigation system had clearly documented and requested the need for metric units. The foremost mistake that was made was by the contractor Lockheed Martin Astronautics located in Colorado which used English measurements. The contractor, Lockheed Martin Astronautics, as per its agreement, was ideally supposed to convert all of its measurements to the metrics system (Llyod and Writer, 1999). But, the contractor did not follow the specifications. It could be a mistake or misunderstanding on the part of the contractor. The error or discrepancy managed to introduce a bias in the calculations of the trajectory of the orbiter which ended up sending the Climate Orbiter too close to the planet than intended.

The inspection of failure of the probe also identified inadequate verification and validation of the navigation system as another reason for the failure. Although the incorrect units were implemented by the contractor, it was not the only cause of failure of the orbiter. The investigative team found several issues that led to the failure of the Climate Orbiter. The Orbiter ended up destroyed because the navigation system was not completely verified by the team. The procedure of verification and validation of the system failed to confirm that the navigation system met the requirements prescribed by NASA. The development team responsible for the on-board software system failed to ensure that the specifications were correct. The testing team neglected to use the requirements specification document correctly to ensure that the on-board system was compatible with the navigation system. There is a possibility that each component was tested individually and not together as a whole. The team that carried out the investigation did not find ample evidence of testing from one end to the other. They could not find proof of any end to end system testing that was conducted by the test team. Additionally, it was difficult to determine if independent or separate verification and validation of the software had been done or not. The process of verification of the interface was not carried out efficiently. The entire process of verification of the navigation system and the validation of the requirements was not sufficiently stringent. It failed to sniff out the discrepancy between the requirements and its implementation. If this process would have been carried out properly, the flaw in the metric system would have been exposed long before the mission started. The testing team failed to check for compliance of the software requirements which contributed to the disintegration of the space probe.

Another key point of failure that was identified by the investigative team was the lack of communication between teams during the mission of

the orbiter. During the course of the mission of the orbiter, several chances presented themselves that could have helped catch the bug. There were several opportunities when the development team could have identified the bug and taken steps to correct it. When the orbiter was in action during the initial 10 month period, a slew of anomalies had been noticed. The project team had the golden chance of ensnaring the error and repairing it, but they missed it. The key problem was ineffective communication between the different operation teams of the orbiter. When it was still working, there were some inconsistencies and anomalies that would have raised suspicions with the operations team. The issue was that the team responsible for the navigation did witness these anomalies, but they did not share their concerns with the ground control and other teams. They discussed their growing concerns regarding the trajectory of the Orbiter amongst themselves but failed to communicate their apprehension to the operations team or the managers of the project. They failed to communicate their issues or concerns and their potential impact to the respective authorities such as the operations team for instance. There was insufficient communication between several teams across the project which was extremely unfortunate. All the teams were at different levels of understanding which contributed to further lack of communication between them. Moreover, there were no 'formal' methods of communication. Most communication that was taking place between the teams was informal. Informal and insufficient channels of communication were used instead of standard means of reporting concerns of this kind. The operations team that dealt with the navigation system was particularly isolated from the other operations teams which further prevented the informal concerns to reach the appropriate parties.

The next clause that contributed to the mishap was incomplete understanding of the design of the probe. The Mars program of NASA employed the same operations team for all of its MARS missions. This approach was thought to be a cost saving approach that would help the team learn quickly. But the downside to this approach was the members of the operations team did not gain full understanding of the orbiter. They were unable to gain in-depth knowledge of the design and working of the Climate Orbiter's system. They could not master the functioning of the Orbiter which led them to make incorrect decisions. The same team was responsible for the three Mars mission at the same time which is a stressful situation. The operations team was deficient in staff and was not capable of managing the three missions all at once. They were given the bare minimum training and did not have sufficient knowledge of the design of the orbiter.

They were oversubscribed and the team mainly relied on their knowledge of the Surveyor that was developed before the Orbiter. The operations team depended on their knowledge of the Surveyor to operate and handle the Orbiter which was not right. This led them to making assumptions about the working of the orbiter. Based on their prior information and knowledge about the Surveyor they made assumptions regarding the orbiter which greatly contributed to the failure of the mission.

There was a contingency plan put in place that could have saved the Climate Orbiter. It was called the trajectory change maneuver (TCM-5). This plan would have swayed the space probe away from the surface of Mars and put it at an altitude where it would survive. But, due to a lack of knowledge of this plan, it was not executed. The team did not understand that TCM-5 was essential in order to save the Orbiter. Had the staff been trained, they could have at least tried to save the orbiter (Mars Climate Orbiter Failure Board Releases Report, 1999).

Lastly, even if the team would have been aware of the maneuver, they lacked preparation for launching the maneuver. Adequate analysis was not done for this maneuver and testing was incomplete as well. The procedures and documentation pertaining to the TCM-5 maneuver weren't completely developed and the team lacked a deeper training. So, the team decided not to use this maneuver and allowed the orbiter to continue its original timeline without considering the risks.

The 8-point cause of the failure detailed by the Mars Climate Orbiter Mission Failure Investigation Board is as follows (Mars Climate Orbiter Failure Board Releases Report, 1999):

- *Errors went undetected within ground-based computer models of how small thruster firings on the spacecraft were predicted and then carried out on the spacecraft during its interplanetary trip to Mars.*
- *The operational navigation team was not fully informed on the details of the way that Mars Climate Orbiter was pointed in space, as compared to the earlier Mars Global Surveyor mission.*
- *A final, optional engine firing to raise the spacecraft's path relative to Mars before its arrival was considered but not performed for several interdependent reasons.*
- *The systems engineering function within the project that is supposed to track and double-check all interconnected aspects of the mission was not robust enough, exacerbated by the first-*

time handover of a Mars-bound spacecraft from a group that constructed it and launched it to a new, multi-mission operations team.

- *Some communications channels among project engineering groups were too informal.*
- *The small mission navigation team was oversubscribed and its work did not receive peer review by independent experts.*
- *Personnel were not trained sufficiently in areas such as the relationship between the operation of the mission and its detailed navigational characteristics, and the process of filing formal anomaly reports.*
- *The process to verify and validate certain engineering requirements and technical interfaces between some project groups, and between the project and its prime mission contractor, was inadequate.*

All the operations teams, the project team and the testing team failed to notice issues in the system throughout the cycle of the mission which ultimately led to the failure of the Mars Climate Orbiter's mission. Incorrect verification and validation, incorrect implementation of the requirements, insufficient communication between the teams and a lack of understanding of the design of the orbiter were all the factors that led to the preventable failure of the mission. All these factors coupled with a gross lack of adequate communication prevented the operations teams from identifying the error and mitigating the situation in advance causing the mission to fail.

We can learn many crucial lessons from this incident. One item that is of utmost importance is the process of verification and validation of the requirements of the software system. It is crucial that correct implementation of the requirements is done and later verified by the teams. This helps in identifying any inconsistencies between the request and its implementation early in the life cycle of the software and helps reduce unnecessary costs. Had this been done in an efficient manner, the mission might have been saved.

One more lesson that can be taken from this study is that communication is key in any software project. Proper communication between different teams involved in any project is primordial to its success. It helps maintain the software and ensures smooth functioning of the product once it has been developed. Additionally, it should be made sure that different teams

that monitor the system once it is in production have proper lines of communication open. Any anomalies, concerns, discrepancies must be reported and communicated between the teams so as to ensure that errors and faults do not go undetected for long periods of time. Every project must establish formal lines or channels meant for communication on a regular basis. Informal means and methods of contact should be avoided at all costs. In the case of the Orbiter, it was the lack of official and formal communication that was instrumental to its failure. Proper channels of communication, if present, would have prevented the loss of the orbiter in space.

Lastly, the importance of system testing cannot be stressed enough. A complete end to end test that explores the system is essential. It helps verify smooth integration of different components of the system. Even if individual components of the system function, there is a chance that these components do not function well together. Complete testing of the system can help a great deal in sniffing out major issues in the interoperability of heterogeneous and complex systems. In the case of Mars Climate Orbiter, there was no proper end to end test that was conducted which could have potentially identified the discrepancy and probably prevented the mishap from happening. If the testing team was sufficiently trained and had they conducted a test where the navigation system interacted with the other software systems, they would have noticed the discrepancy in the trajectory calculations and identified the error. Testing in detail can only benefit the project in the long run.

4.4. CASE 4: NISSAN VEHICLE RECALL

Nissan is one of the biggest automobile manufacturing companies in the world (Co. and Ltd., 2001). This Japanese company founded in 1983 is responsible for mass producing automotive vehicles on a large scale. It is a multinational company that sells its cars under many different brand names namely Nissan, Infiniti, and Datsun (Co. and Ltd., 2001). With time, more, and more automobiles started the implementation of software controls in their automotive systems. Software is now at the center of many eco-systems and it promises to do a lot for this field. Software began to make its way into cars as software helped improve their efficiency and increase the ease of use for the customers. Complete automation in the automotive industry is the next target of the automobile world and software is the only way of achieving this goal. As IT worked its way into the automotive industry at a slow but steady pace, the automotive vehicle industry saw a rise in the number of IT-related issues as well.

Nissan was forced to recall over a million vehicles starting from the year 2014. The reason for this massive recall was a software glitch. The software glitch turned out to be a serious one. The software system installed in some of the vehicles manufactured by Nissan was unable to correctly identify if someone was seated in the passenger seat of a car. The system was programmed in a way that would identify if someone was seated in the passenger side of a vehicle and if a crash occurs, based on its identification capabilities, the airbags would be deployed. The issue was that the system of occupant classification was supposed to discern whether someone was seated in the passenger side or not. If the system did not compute that a person was seated, it would not deploy the airbags in a collision situation. The issue was that the system was unable to ascertain whether the seat was occupied or unoccupied. This determination would help the system further in its decision to activate the passenger side airbag. If the occupant identification system did not detect an adult holder/occupant in the seat, the passenger side airbag would be deactivated. This gave rise to a crushing possibility where the airbags would probably not deploy even if someone were seated in the passenger seat. This was a big safety risk and hence the company was ordered to recall their cars.

This issue was found on cars that were from the 2013–2014 models released by Nissan. The car ranges or models that were impacted by this defect were (Teli et al., 2014):

- 2013–2014 Nissan Altima;
- 2013–2014 Nissan Pathfinder;
- 2013–2014 Nissan LEAF;
- 2014 Infiniti Q50 and QX60;
- 2013–2014 Nissan Sentra;
- 2013 Infiniti JX35;
- 2013 Nissan NV200.

Over 544,000 sedans of the model Altima were recalled initially. Later this number rose to the tune of over 1 million as more experts investigated the issue and customers reported issues with several models as well. The list of recalls includes: 544,000 Altima sedans; 29,000 Leaf electric vehicles; 124,000 Pathfinder SUVs; 183,000 Sentra compact cars; 6700 NV200 taxis; and 104,000 Infiniti JX35, Q50, and QX60 vehicles (Isidore, 2014). This recall was done in the United States and Canada. The failure of an airbag of a car in the uncertain event of a crash is a perilous and high risk situation

and should never happen. Apparently, if the weight of the passenger side occupant was not enough, or the position of sitting was ‘unconventional’ the system could not tell that a person was seated in that seat which led the system to deactivate the airbags. So if the passenger-seat airbags would fail and not deploy it greatly increases the likelihood of injury and death.

For instance, if we imagine a small child to be seated in the car, it is safe to assume that a tiny child whose weight might not be a lot, may possibly not be thought of as an ‘occupant’ by the system. The implications of this assumption or calculation are disastrous. One can imagine what would happen in the worst case scenario which would be a car crash in this case. So now, during the crash, if this child is seated in the passenger side, the system would have been unable to discern the presence of this minor and hence would have deactivated the airbag on the passenger side. So, when the crash occurred, due to the bug, the software system would make an error and would be unable to determine whether the child is sitting in the front passenger seat. The system would determine that the seat is vacant and in the crash, it would not deploy the airbag. This could result in the child suffering from severe bodily injuries and in the worst scenario, death.

As per Nissan, the airbag system was faulty and the automaker stated that the software that made these decisions was calibrated in such a way that rendered it to be highly sensitive to other factors as the engine vibration. The automaker, Nissan, informed the National Highway Traffic Safety Administration that a software issue in the occupant classification system was possibly unable to detect an occupant in the passenger side seat (New York Times, 2014).

An article published in the *New York Times* states that, “The automaker blamed the sensitivity of the software calibration, particularly when ‘a combination of factors such as high engine vibration at idles when the seat is initially empty and then becomes occupied’ or an ‘unusual’ seating posture are factors” (New York Times, 2014).

The posture of the passenger seated in the passenger seat was also one of the reasons that led to the air bags being deactivated by the system. This means that if a person sat in the passenger seat in less than ideal posture, the airbag would be deactivated because of the software glitch. This was thought to be a serious issue as one can assume that young children do not sit still. So, if a child chose to sit in an ‘unusual’ position such as the edge of the seat or to the extremely far left or right edges, there was a chance that the software system would not be able to determine that this child is sitting

on the passenger side seat. It would incorrectly calculate the absence of an ‘occupant’ in the passenger seat and deactivate the respective airbag. So, if the car were to be involved in a collision, the airbag would fail to deploy and the child would surely be injured in some way which is just devastating.

Additionally, their response also states that a factor was the higher vibration of the engine at idle times when the passenger seat would be initially empty and would be then occupied. This potentially means that if the seat was initially vacant but was filled later when the engine was vibrating at higher rates, the system would not be able to determine that the seat was now filled. It would mean that if the car was idle and waiting, and if there was no passenger in the passenger seat in the beginning, the system would not notice or calculate the presence of a passenger later as the vibration was potentially higher. This is also a potentially dangerous situation for the passenger side occupant as he would not be counted in the event of a car crash.

Moreover, the company was aware of three separate instances of crashes related to this particular defect. In these three incidents, it was found that although there was a passenger in the passenger side seat, the system said that the seat was unoccupied. In the case of these crashes, in all three instances, the airbag on the passenger side seat did not deploy during the crash. The automakers stated that they were not aware of any fatalities that occurred that were related to the particular software error. The experts that analyzed and investigated the problem said that they were unaware of the problem causing any wounds to the passenger side occupants. They stated that they did not have any concrete information that could link crash wounds directly to this particular software deformity. Any details pertaining to injuries caused as a result of this software glitch were not released by Nissan. As per the company, they had resolved the particular issue after many customers had registered complaints with them (New York Times, 2014).

The system was initially deployed with the thinking that airbags are more dangerous than an accident itself. Several studies indicated that the risk of injury, casualties or death were greater due to the activation of airbags. Many vehicles have been recalled over the years due to faulty airbags that would deploy incorrectly and even send off shards of metal during a crash. It was thought that the risk is higher when an airbag is deployed rather than when it is not. Another instance when cars were recalled by Nissan in April 2013 was due to potentially lethal airbags. Nissan had to recall around 480,000 models manufactured/modeled from years 2001 to 2003 due to

defective airbags. Some of the models were Nissan Maxima, Pathfinder, Infiniti I35, Sentra, and Infiniti QX4 series vehicles. They expressed that there is a possibility that harmful shards of metal could be launched from the airbags in these cars in the event of a crash. This was a dangerous issue which meant that when a customer or driver of these cars was in a crash, they would more likely die from the airbag than the actual crash. They may survive the crash but the airbag that would be deployed would release flying projectile shrapnel in their bodies leading to injuries or even death. They blamed the source of this issue on another Japanese company named Takata that made the explosive particles or wafers that was being used in the airbags of these cars (Isidore, 2014).

So, in an attempt to reduce risk, systems that detected a presence in the passenger seat came into the picture. The potential risk of physical trauma to a child was higher from an airbag than an accident which led to the development of such occupancy detection systems in cars. The need to make airbags safer was imminent. As unnecessary deployment was one of the main reasons, it stands to reason that the airbags must deploy only in case someone was actually sitting in the passenger seat. So it was thought a car should not deploy the airbags and sense if there isn't enough weight in the front passenger seats. But, when Nissan tried to implement this system, they failed to check its validity in all cases. The issue with their cars was a completely new issue. In a bid to develop safer airbags, they went a little too far. They developed sensors that shut off the airbag in a passenger seat even if someone (adult or child) was present in it. So, in an attempt to 'fix' the previous error due to the potentially dangerous airbags, the situation was made worse. This was a grave error which led to Nissan having to recall millions of cars in a 2 year period. This would have cost them a lot in terms of capital and of course, their reputation must have been significantly hit due to these consecutive issues in their cars.

This incident helps us understand the importance of testing and testing suites in software. If the automotive manufacturer in this case had conducted crash tests that included children or possibly thought of 'unusual' seating postures, they would have been able to identify the defect during the testing phase. If there had been a series of tests to assess the proper functioning of the airbags in the event of a crash, there is a possibility of the software bug being detected early on (Teli et al., 2014). It would have prevented them from having to recall more than 1 millions of their vehicles. If someone had thought of out of the box situations and come up with a test plan accordingly, it would have helped them determine bugs of this nature. A

lot of time, effort, and manpower would have been saved if the issue could have been discovered at an earlier stage. Additionally, if their software design would have accounted for different impacting factors such the engine idle time, initial absence followed by presence in the passenger seat and ‘unconventional posture,’ the occupant detection system probably would not have failed. It makes us see the importance of design of software. We learn that each phase of the development life cycle of the software is important and must be executed with the same level of attention. This incident tells us to imbibe principles of effective software design in order to build, develop, and maintain good quality software systems.

4.5. CASE 5: MULTIDATA SYSTEMS DISASTER

Software has made an enormous impact in all walks of life. Almost all fields are impacted by software in one way or another. One field in particular that has seen a tremendous shift in their working is the medical field. Healthcare systems, over the last decade have gradually moved from the paper and pen format to a more electronic or digital format. Medical systems including hospitals, emergency units, pharmacies, etc., have started to migrate towards software systems for storing of data related to the patients, medicines, treatments, and so on. Software is now being used to replace the traditional paper records that were in use since a long time ago. Hospitals all over the world have been phasing out their old paper records and are gradually moving towards digitizing all of their records and converting them to an electronic or machine readable format. With the increasing demand for software to be implemented in this industry, most hospitals have been forced to move their businesses to a more electronic stage. Hospitals have started to keep digital records of their patients, their respective treatments, their histories, their inventories, their staff and so on.

Digitization of records was probably the first step towards the implementation of software in this domain. Software has moved from being a small player to the most valued player in the medical industry. Over the years, imperceptibly, subtly it made its way into the domain. Software that helped the medical industry came to be developed. Vital instruments such as fit bits, software that measures heart rates, pacemakers, and many more items that depended on software made their way into our lives. Soon machines using software such as insulin pumps, imaging machines, full body scanners for example, came into existence. Several critical and lifesaving machines now depend on software for their functioning. People started to depend heavily

on software run medical machinery and appliances in their daily lives. Now it is a fitted part of the medical system. With the increased volume of use of software in medicine, there arose several dangerous issues with it. One such instance that proved fatal was the incidence of overexposure of radiation on cancer patients at the National Cancer Institute, located in Panama City. This incident is known to be one of the most notorious software disasters of all time.

A software bug in the software that was being used to inject cancer patients with radiation had a serious flaw which caused the patients to be exposed to dangerous amounts of radiation. The National Cancer Institute regularly treated patients suffering from different types of cancer such as bowel, cervical, brain, lung, prostate, and other types. Treatments for cancer include mainly chemotherapy, surgery, and radiation therapy. The Republic of Panama has a single institute that performs radiation therapy for cancer patients. It is the National Oncology Institute (Instituto Oncológico Nacional) and it is the largest and the only public radiation therapy institution there. The only other option publicly was to visit private radiation clinics that not all people could afford. For most people, this institute was the only choice if radiation therapy was the treatment option for a patient's cancer diagnosis.

Starting from the end of the year 2000, the cancer patients from this institute were being overexposed to radiation. The doses of radiation they were receiving were severely higher than the prescribed amounts. This fact was not known to ION until the month of March in the coming year 2001. At this point, the director of the National Oncology Institute was alerted that cancer patients that were being treated with radiation therapy were reacting severely to their radiation treatments. This was not the expected outcome to their planned radiation therapy for cancer treatment. There were around 478 patients that were treated between the month of August in 2000 and March 2001 (Borras, 2006). Out of these patients, three had died during this time. It was speculated that these deaths had occurred due to radiation poisoning. But at that point, no precise conclusions could be drawn. In an effort to determine the cause of the overexposure, the government of the Republic of Panama asked experts to start an investigation into the case. Medical physicists were invited from the Pan American Health Organization (PAHO) to investigate as well. The Ministry of Health of the country of Panama asked PAHO for their assistance in the investigation in April 2001 (Coeytaux et al., 2015).

The medical physicists from PAHO conducted a study and analysis of the patients and the recent deaths. They did an analysis of the patients

and determined that patients that were being treated by a computerized system were impacted. They calculated that around 56 patients were impacted. They were coming in for treatment of cancers such as uterine cervix, rectum, prostate, or endometrium. These areas were treated by means of a computer treatment system as the areas under consideration were sensitive and required to be partially blocked so as to ensure healthy tissue in these areas wasn't targeted by the radiation. Their analysis revealed that the radiation dose received by the patient who first died was more than twice the prescribed values set by the government and the health system in Panama. The investigative team found that the patients impacted were those that were being treated by a computer system that calculated the radiation dose for each patient. The team from PAHO then calculated the doses of radiation that were administered to the 56 patients under consideration. They ascertained that for just 11 of these 56 patients, the acceptable error of $\pm 5\%$ was found. But, their analysis revealed that for 28 out of these 56 patients the errors were significantly higher than the norm of $\pm 5\%$ which was unexpected and incorrect. There were drastic errors in the range from $+10\%$ to $+105\%$ for the 28 patients which wasn't the acceptable limit. They received doses that were in excess of as much as a 100% which means that some of these 28 people were exposed to more than double the quantity of radiation required to treat their cancer. The team came to the conclusion that these 28 patients were overexposed to radiation therapy and that they were treated by a computerized system. The ION along with PAHO concluded that these patients were overexposed. Soon other experts confirmed their analysis. Medical experts working at the M.D. Anderson Cancer Center corroborated the findings of ION/PAHO regarding the overexposure. They determined that there was an issue in the computerized treatment system. Soon, they determined that the fault lay with the system developed by Multidata Systems Incorporation Limited, a United States based firm. They were responsible for the development of the radiation treatment system that was used to treat cancer patients with cobalt radiation therapy (Borras, Rudder, and Amer, 2001).

The National Cancer Institute of Panama, after they discovered the issue, reported the problem to the Panamanian regulatory authority that was responsible for managing radiation safety. Due to the sensitive nature of the case, soon the International Atomic Energy Agency (IAEA) got involved. They were requested the Panamanian regulatory authority to send their experts to analyze the situation. The IAEA complied and sent their experts to investigate these overdose cases and the deaths linked to it. One of the lead

physicists from PAHO's joined the team send by the IAEA. This organization is responsible for ensuring that safety protocols in radiation therapy systems are met. They verify and validate the calibration that controls the dosage of the beams of radiation used in radiotherapy in medical institutes, hospitals. The IAEA makes use of thermoluminescent dosimetry (TLD) to determine the doses used in therapy and ensures that they are within standards. The lead physicist from PAHO reviewed the TLD measurements from the cobalt based radiation therapy unit at the ION (Leveson, 1993). The doses received by the 56 patients that were a part of the initial group that were possibly impacted by the treatment, were analyzed. The medical charts belonging to these patients were checked by the team. Several data related to their treatment was collected from their charts such as:

- Treatment times for the patient;
- Size of the field/area that was being treated;
- Factors such as if beam modification was used (if the area was partially blocked or not);
- Depth of the treatment beams from the skin of the patient;
- Radiation output of the cobalt 60 therapy unit.

All these details from the charts of the patients were studied, reviewed, and analyzed to determine which patients were given higher doses and why. For patients that were highly impacted by radiation, the doses they received were also calculated manually. The factor that was different in the affected patients was the treatment time. It was determined that some patients were treated for longer times than they should have been treated. By the end of May 2001, they found out that the distributions of the doses for 28 patients was slightly different in some cases. Two sets that differed were identified by the analysts. They unearthed the main reason for the overexposure. The issue was that when blocking coordinates or 'blocks' were being used in the treatment, there were significant discrepancies in the amount of radiation used. When these 'blocks' or shunts that were supposed to protect the healthy tissue from the cancerous tissue were entered/placed in an individual manner, they worked in the anticipated way. But, if they were entered in, a continuous manner such as a 'loop' there was an issue.

What was discovered was that the system was calculating higher treatment times per minute for the 'loop' combinations of therapy. For the block, data (shield) that was entered in a loop, the time of the treatment for each dose of radiation that was given to the patient was double which was incorrect. It was determined that the system that calculated the time of treatment for each

dose was unable to calculate the correct dosage of radiation for each patient. The patients undergoing treatment via radiation therapy who needed shields were being exposed for longer periods of time based on the calculations of the investigative team.

Most of the patient records that were being studied suffered from cancer in the pelvic region. The pelvic region is highly sensitive to radiation as the kidneys and intestines can be impacted severely by overexposure to radiation. These organs are highly sensitive to radiation and hence care is taken so as to limit unnecessary exposure to radiation during radiation therapy. Usually, when a cancer patient is diagnosed and he decides to opt for radiation therapy, a plan of treatment is decided by the doctor. This plan is made to decide the amount of radiation that can be applied to the tumor or cancerous region. The doctor decides how much radiation would be required to treat the tumor in a safe way. Different aspects are considered by the doctor such the position of the tumor, the depth of the tumor in the body, the possibility of spread of the cancer and the sensitivity of the organs surrounding the tumor to come up with the most effective form of treatment. The physician takes all these factors into consideration and then chooses the best angles for attacking the cancerous areas. The treatment plan usually includes the use of shields that are placed above and below the region of the cancerous tumor. These items are also called placement shields or 'blocks.' So, the doctor tries to determine where exactly these blocks would be placed in the area around the tumor that has healthy tissue (Borras, 2006). These placement shields or blocks are generally made from lead or a similar metal alloy namely cerrobend. These blocks aim to protect normal non-cancerous tissue from the cobalt gamma rays. Once the plan is finalized by the doctor, it is given to a physicist who then enters this information in the software system that later calculates the duration of the treatment. The software systems, in most cases, generate a 3-D depiction of that presents the distribution of the radiation. The 3-D drawing depicts how the different beams of radiation originating from different angles will intersect within the patient's tissue. The dosage is decided by the doctor and after he prescribes an amount, the time of treatment is calculated by the software system.

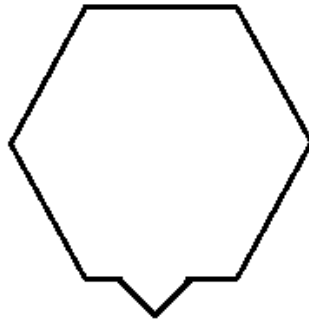
The computer system that was the source of calculating the duration of treatment allowed for the placement of blocks of metal (also called shields) to protect the patient's healthy tissue from being exposed to radiation. These shields were supposed to be entered manually in the system by a radiation therapist. The software provided by Multidata permitted the clinicians to

draw on a screen and decide where the shields would be placed. This was put in place to help the patients from being overexposed to radiation during their radiation therapy treatments. The radiation specialist's job was to manually enter the location of these metals shields or blocks so as to protect the correct zones from the radiation. The precise location of these shielding blocks was decided by the radiation therapists who could only do it by manual inserting the locations. The software, in this case, permitted a radiation therapist to draw on a screen the exact location where they would be placing the metal blocks or shields on the patient during their radiation therapy treatment. This is where there arose a critical issue in the understanding of the system.

The issue was that the software provided by Multidata only allowed the placement of 4 or less shields. The software reportedly worked well within the margin of allowed error when there were 4 or less protective shields placed. In case of the cancer patients that were being treated, most of them were getting radiation therapy in sensitive areas such as the pelvic region. Hence, the doctors had to be careful while prescribing the doses and also had to place shields so as to protect the other organs situated in close proximity to the pelvis. Now, it was determined that the physicists in the ION were adding a fifth block while they drew the shields on the computer system, in a bid to add more shields. The doctor reportedly requested the radiation therapists to be more safe and protective. So, the physicists wished to add another shield to protect the patients. They found a way to add a fifth shield or block in the computer system and did so thinking they were helping the patients and saving them from unneeded exposure to radiation. The physicists added a fifth shield in the software thinking it was going to add another level or protection. The extra shield, this extra fifth block was supposed to help these at-risk patients against overexposure to radiation. These patients were sensitive to radiation and were at risk as their tissues were sensitive due to earlier radiation treatments and past invasive surgeries. But ironically, it did the exact opposite leading to disastrous consequences for the patients involved.

The base issue was the software provided by Multidata was designed to only handle 4 or fewer shields. It was supposed to calculate the treatment times only when there were a maximum of 4 shields present. But, the clinicians working were instructed to be safer and more protective. One of the technicians attempted to trick the system. They read the software manual provided by Multidata systems and came to the realization that the software could be manipulated and it could be made to add an extra shield or block. They failed to realize that it was a wrong move. The radiation specialists

wanted another shield as they thought it would be better and so they came up with a way to fool the software system. They figured out a method that permitted them to place 5 shields instead of just 4 shields. What they did was that they drew a large block instead of 4 separate distinct blocks with a hole in the middle. The software allowed the technicians to enter 5 blocks if a single composite shape was drawn and a hole was left in the center. This was supposed to inform the software to target only the exposed part (Borras, 2006). For example, a rectangular shape with one triangular 'block' in each corner and a fifth block protruding from the top or the bottom was allowed by the software. The block looked something as shown below:



As we can see, the block overlaps itself and is drawn in a loop, which, unknown to the radiation therapists, was not actually beneficial to the patients in the long run. The blocks determined the amount of time the patient would be exposed to radiation during his or her treatment. What the technicians did was that they entered the coordinates of the unusual rectangular block, first the inner and then the outer perimeter of the shape. What the shape did was that it left a single hole in the center that was supposed to convey to the software the area to be protected. What the doctors and technicians did not realize is that the software had a serious bug. When the data for this weird shape was entered, if it was done, say in a clockwise manner for the inner perimeter followed by an anticlockwise direction for the outer perimeters, it worked correctly. But, if the direction of drawing was the same for the outer and inner perimeters, the calculations were incorrect and significantly higher than the normal dose. The bug in the software was that the direction of drawing confused the system. If the direction of the outline was a specific way, it recommended twice the mandated dose. The error in the system was

when the looping occurred. When the loops that defined the two perimeters, the inner, and the outer perimeter overlapped or crossed one another, the software was unable to determine or recognize the shape that was drawn. This bug resulted in the software to get confused and behave differently. The software behaved differently and depended on how the hole was drawn. At no point in the usage of the system was a red flag raised saying that such shapes were not permitted or that looping wasn't allowed. The system locked up and started miscalculating the required dosage times. So, for each treatment that was received by each of the 28 impacted patients, they received an overdose of harmful gamma rays due to this bug. They were given significantly higher amounts of radiation ranging from an overdose of 20% to 100% which is a serious issue. The staff was digitizing the lines and crossing them in an attempt to fool the software into thinking that there are 5 shields instead of 4. This turned out to be a calamitous decision. The moment the loops of the inner and outer perimeters crossed or overlapped, the treatment times were automatically increased by the software (Borras, 2006).

Further, by law, the radiation therapists were supposed to re-check the dosage calculations that were generated by the computerized system, which they failed to do so. They introduced changes in the existing software system and neglected to check whether the dosage or the treatment times were correct or not. It was the responsibility of the user of the system to double check measurements that were generated in terms of treatment times which the physicists did not do. Another issue that was found by the IAEA during the course of their investigation was that any verification that was done by the hospital was only related to the hardware and not the software. The quality assurance measures were not put in place for the software system. There was no system that required the physicists to check or verify or confirm the results generated by the machine or computer system. Moreover, the staff was not required to disclose the fact that they had modified the way of entering data into the system or that it was now being drawn in a loop. The staff failed to check the dosages which were a grave mistake on their part. If the hospital staff had verified the software system's calculations by manually computing them by hand or by other means such as testing of the doses in water before the patients were exposed to radiation, the overexposure and the incorrect calculations would have been identified before anyone was impacted.

By the month of June 2001, the FDA was now involved in the investigation. The IAEA team published their report following detailed analysis and investigation conducted by their experts. The FDA started

investigating Multidata systems and their role in the overexposure. But, the software was still being used during this investigation even though it had a serious bug. The outcome following this incident is etched in the memories of some people forever. This was an unfortunate and ill-fated incident and the fallout from this incident is astounding.

This incident resulted in death of several of the patients. Several of the patients were overexposed and suffered from severe health issues due to radiation poisoning. Initially the death toll was limited to around 9 patients. But, all the patients suffered as a result of a fault in the system that was exploited by the staff. By the end of September 2005, 23 out of the 28 patients had died, mostly from the impacts of radiation poisoning. At least 18 were confirmed to have died due to complications resulting from their overexposure to radiation. One can say safely that the fallout was enormous. The lives of hundreds of people were put at risk due to a combination of negligence and software error (Borras, 2006).

In May 2003, the three physicists who incorrectly entered data into the software system and neglected to check the doses were indicted for murder in the second degree. The trial proceedings started this year. The FDA issued an injunction against Multidata Systems and prohibited them from manufacturing and issuing radiation therapy machinery in the United States of America. By November, the criminal proceedings were complete; one of the three physicists was acquitted while the other two physicists were found guilty and sentenced to four years in prison. The situation was grave and the worst part is that the software was still in use during the years from 2001 to 2003 when the investigation and the trial were ongoing (Borras, 2006).

Further, in their report, the IAEA reported that the manual provided by Multidata was not sufficient for its intended audience. Their report concluded that the manual provided by Multidata systems was not easy to understand and did not describe the digitization process of the shielding in adequate details. They revealed that there was a lack of illustrations demonstrating the use of the shields and specific warnings. Further, different ways of data entry were not mentioned in the manual. This means that there was no warning against the possible manipulation of data entry coordinates. No deterrent was issued regarding the possible incorrect entry of data into the system. The possibility of any error in case of false entry of data was not considered or described in the manual. It was assumed that no possible entry could result in any miscalculations by the system. Also, no preventive measures were provided in the manual. This means that, no information

that suggested bad calculations was given in the manual. No preventive information was given pertaining to possible data entries that could result in bad calculations. It failed to mention if any particular sequences of entry of data would result in incorrect computations. The manual did not mention that entry sequences that overlapped or crossed or self-intersected would result in bad calculations which would have stopped the physicists from manipulating the system. Due to the overall lack of information about the software, the physicists were able to enter not only incorrect shapes but also incorrect looping/self-intersecting entry sequences that killed people. It was an abject incident that could have been easily avoided if the people involved would have been more alert and also aware of their seemingly correct decisions. The company, Multidata, suffered a lot of backlash for the way they handled the situation and was taken to court by several of the affected patients. But the damage was already done. They could not bring back the dead and they refused to admit fault in their system. They did not respond well to the letters or warning sent by the FDA and further harmed millions of other people using their software system (Borras, 2006). They are being tried in Panamanian court now. The incident is a stark reminder of how software can impact lives and given in the wrong hands, it could be disastrous for those involved.

This incident teaches us a lot in terms of software quality assurance. Software that could potentially harm patients should always go through stringent measures of quality assurance. This software literally impacts millions of people and decides the fate of millions of people and hence should be treated with some amount of importance and prudence. This incident tells us that quality assurance is a pre-requisite to life-critical software systems. The quality of software that impacts so many people should be high in the first place to begin with and secondly it should be regularly maintained. Had the software system been thoroughly tested, the bug could have been sniffed out long before it became a lethal instrument of death.

Software systems that are critical in nature must be regulated on a highly rigid and inflexible level. The people working on developing such life altering systems must be well trained, certified, and informed of the implications of their code. The developers are responsible for creating the code that runs these vital lifesaving machines and they must follow strict rules when it comes to software development. Stringent software development practices must be followed when such important software is being developed. Ideally, developers working on such software must be certified and must be well trained to code on such systems that have a significant risk. Proper coding

conventions should be adhered to by the developers and they should strive to develop a product that is of an excellent quality. They help maintain the code and make it easier to read, comprehend, and analyzed by people other than the developers.

The bug that killed so many patients would have been discovered if someone would have tried to test the boundaries of the system. If values that were usually not supposed to be entered into the system were entered, the unusual behavior would have been exposed. The development team would have identified that the software was reacting in a different way and possibly uncovered the bug. If the testing team would have entered incorrect data entry points or incorrect shapes and tried to run the system, and then verified the calculations, it would have showed that the calculations are way too high than the standard. This would have set off the proverbial alarms and possibly someone would have identified the bug or even looked into the root cause of the issue. In depth testing, followed by verification and validation of the outputs would have probably saved the souls of the people who are now dead due to impacts of exploitation of the software bug.

A crucial lesson that we learn is that just testing is not enough. Validation along with verification of the expected output must be done. Compliance of the requirements should be done and double checked in case of critical and sensitive software systems. The results of exposed errors that could be potentially manipulated by the users of the software system can be disastrous and must be avoided at all costs. Life-critical software systems must be heavily regulated. Hence, testing the software system with proper validation and verification processes in place is a must. It can help eliminate serious issues in the design and development of the software.

One of the key reasons for limited testing of software is money. The issue is, in order to ensure detailed and invasive testing of a software application, a lot of time, money, and effort is needed as an investment. Many software providers are aware of several testing tools that are available, and which can help them perform almost exhaustive testing of their code. But these tools are never considered as they usually need to be purchased and project budgets usually do not deem testing tools as a 'necessary' expenditure. Many organizations choose not to invest in expensive, commercial testing tools in order to avoid additional costs. The main goal of any project is to inflate and expand the profit while limiting the financial/monetary investment in the project. Hence, many software testing tools are overlooked and never even considered. But, testing can be performed manually as well. Even though

manual testing may not be accurate and may not be highly exhaustive, it can help in making the software application devoid/absent of major bugs. But unfortunately, in some instances, testing is not done properly. The issue is that setting up a testing team cuts into the project budget and hence not a lot of personnel is assigned to this task when budget considerations are made. These people could be working on other items deemed more important than testing and hence testing is not given the importance it deserves. Also, most organizations that provide software have a tendency to test the system only when the development is complete and not during the course of development. If bugs are discovered, they are fixed on a quick last-minute basis. Testing is considered as just a formality at times. Testing done at the end is usually done with the intent of quick fixing and not looking for major issues in the code. This means that sometimes deadlines must be met and that testing is the last thing that is done. In most cases, the testing process is rushed in a bid to meet the deadlines set by the management. This should not be done. Compensating quality for time and money can prove damaging and harmful if critical systems contain major bugs or faults. Testing should be treated as an important part of the software development life cycle and given equal importance as it is essential for the well-functioning of the software.

The next lesson to incorporate from this incident is that no assumptions must be made when dealing with critical software applications and if assumptions are made, they should be well documented and the users must be made aware of them. An incorrect assumption was made during the development of the radiation therapy software was that users of the system would not enter incorrect patterns or sequences for addition of shields. Another wrong assumption was to think that since the machine was designed to work with only a maximum of four shields, no one would add more than the specified limit. The assumption that no one will try to test the limits of the system is ludicrous. If the system was built to handle a maximum of four shields or blocks, it stands to reason that any more than four blocks is not allowed. Going by the assumption that the maximum limit is four, one can easily deduce that any number above this value is incorrect or not acceptable to the system. Therefore, by this logic, any value over four should have been automatically rejected by the system. Data entry that is not supposed to be used should have been tested to monitor the behavior of the system. There should have been proper checks put in place to check if the values being inserted are permitted or not. For instance, whenever incorrect data is entered, the system's reaction should have been observed. Ideally, if the data was incorrect, the system should not allow it and warn the user that the values entered are incorrect. The possibility of entering incorrect data entry points and

shapes without any warnings or messages was highly inappropriate. A radiation therapy system is a critical life altering system and hence, should be treated and tested as one. It should be accurate in its functioning and warn the users if the data is incorrect and block the entry of such bad data. Such checks would have helped dissuade anyone trying to enter values that are not allowed. In this case, the manual did not provide any information regarding values, shapes, and drawing patterns that were not allowed by the system or even that overlapping of sequences was incorrect. The physicists were able to enter incorrect coordinates, draw incorrect shapes, and even place five blocks without the system being able to tell that the values were wrong.

Another point to remember is that it should just not be assumed that the user of the system will not make a mistake and enter false data. It should not be assumed that the user will use the system in the way it is supposed to be, as described to the user. One must always assume that mistakes will be made. The software did not account for or even consider an unpredictable user of the software. The creators of this software simply assumed that the people using it would perform a fixed set of pre-determined actions that could never cause any harm. They did not account for unpredictable use of their software which was a mistake. Whenever a user interacts with the software system in an unnatural way, things can go sideways soon. The software can react in several ways, sometimes poorly, as we saw in this case. Unpredictable behavior cannot always be anticipated at all times. Software developers cannot foresee all the unusual ways in which someone would use the software. But, they can choose how their software would react under unnatural circumstances. The only thing a software developer can control is the reaction of his/her product when a mistake is committed. So, software that has huge implications on human life and also relies on humans for its execution must be based on solid assumptions that cannot be misunderstood in any way whatsoever. The users of the radiation therapy machine did make a mistake, they did not check the treatment time and the doses which was a grave mistake on their part, but their oversight shouldn't have led to the death of so many people. If a system has the potential to kill someone, it should be built robustly, and its users should be endowed with enough preventive measures to avoid the worst case scenario.

This incident also guides us in terms of software design and its gravity. This event that took place impacted millions of lives and it impressed upon us the importance of development of good quality software. The software system developed by Mulitdata systems was flawed in its design. It was unable to calculate correct dosages for treatment times when a loop or

self-intersecting drawing was submitted to the software. The doctors and staff made a grave error and did not realize that the Multidata software was calculating higher doses based on the direction of drawing the sequences. If one drew in one direction, it worked correctly and if one drew in another direction, the system recommended twice the standard dose. Sadly, the doctors or physicists and any users of that system should never ever have been given that amount of freedom of input with the software when so many lives were concerned. This reveals a major design flaw in the system. No system of this critical a nature should permit the leeway that was allowed in this case. The physicists should never have been permitted to design a shape that crossed the perimeters or enter a shape that had five shields. The software, in ideal cases, should have raised a red flag when such an event occurred. The calculation error was an issue linked to the algorithms in the Multidata software that was unable to recognize the shapes entered and hence ended up calculating incorrect values for the dosage.

The blocks were supposed to protect the patient's healthy tissue from unnecessary radiation and ended up hurting them. The limitation of the software was of four blocks and it should have been strictly enforced. The physicists were able to get past this limitation which was a failure in the design of the software system. The radiation therapists were able to draw and submit five blocks to the system as a single continuous block with a single hole in the middle which should not have been permitted as per the design of the system. They were easily able to trick the software into accepting five shields instead of four which is a flaw and it should have been considered during the designing of the software system. There should not have been any way to trick the software in the first place. The software, unfortunately, due to the gross oversight, gave different outputs depending on the way the hole was drawn which should never have happened to begin with. Now, the technicians responsible are in prison as a result of their misuse of the software. Even if the users entered or drew shapes that weren't correct, the system should have handled it properly and have certain checks in place that informed the user of the error in his or her input rather than simply accepting any inputs and processing them. The design made assumptions based on the thought that a user will not enter an obscure unusual pattern or false data and hence all input was processed without any validation. This data, which was false, and which killed so many people, was so easily accepted by the system which is abhorrent to say the least. This was a major aspect of software development principles that was overlooked and then ended up having to pay for their mistakes. The designers of the software should have accounted

for it. This occurrence is a lesson in how technology can be misunderstood and misused if not designed properly. It is also a cautionary fable for users of critical systems to be more careful when using such systems. It serves as a staunch reminder to creators of software to ensure quality of their product as some software can kill.

Also, another issue in terms of software design that is highlighted by the overexposure occurred is the inherent complexity of the design of the radiation therapy system. The analysis revealed that the software was built in a manner that was complex. The design of the software was highly complicated and the bug was highlighted only when people drew in a certain way. This means that the design was somewhat ambiguous in nature as no clear outcome was visible when data was entered incorrectly. This nature of the software ended up being exploited and led to fatalities which could have been otherwise avoided. We learn to take into account the complexity of any requirement and simplify it before developing the software system. A bad design must be avoided at all costs in all software applications, no matter their criticality. Bad or poor design can introduce bugs into the system which can later be possibly exploited or misused by someone and can lead to serious consequences for those involved. Complex systems that do not smoothly integrate all of its components can cause unnecessary side effects when using the system after it has been released. Hence, inherently complicated software design should not be implemented as it can lead to possible issues later on in the life of the software system.

Another valuable input to be gained from this study is the maintenance of software. In the investigation conducted by the IAEA, ION, and PAHO, it became evident that the software was not properly maintained. The hospital never acquired a maintenance contract for the software and preventive maintenance was never done on the software. In this case, the machine in question was being used for around 3780 hours per year which was almost two times the recommended amount for the machine as per the maintenance procedure. The staff was using a software system that was not maintained at all and the system was being overused. Although the maintenance was not the issue, it is possible that other defects that were discovered during the life of the software system were not patched as the software was not properly maintained by the people using it. Since the software was being used to treat several people on a daily basis, the least the hospital could have done is to maintain it. In the fall of the year 2001, Multidata systems created a software patch intended to fix the issue that was identified at the National Cancer Institute. Their patch managed to check the calculations that were

given out by the system and also able to reject any shapes that were not identified as valid by the software system. But, any hospital that did not have a maintenance contract with Multidata systems, would probably not receive the software update. This impresses upon us the importance of proper maintenance of software. When a software system or application in use is not maintained, it can lead to major bugs being left untreated due to a lack of funding or resources. In software that is vital to the well-being of millions of people, it should be maintained due to its potential impact on its users.

The last teaching that can be assimilated from this fateful tragedy is the role of proper documentation. The incident reiterates the importance of documentation. It demonstrates the need and high importance of documentation in software systems. The radiation therapy system came with a manual that was not helpful to its users. It was not detailed enough for the people involved to use it correctly. It did not contain sufficient information pertaining to use of blocks or shields during the treatment therapy. It was a critical system and it touched the lives of so many people, hence, its users or intended audience should have been given the best possible tools to help them operate the critical system in the safest way possible. Not only did the manual provided by Multidata systems fail to do so, but it also succeeded in proving that incorrect data that could potentially kill people could be easily entered into the system without any warnings or alerts.

The technicians were able to enter unidentifiable shapes that confused the system as its manual did not provide adequate information that detailed the allowed shapes, sequences, and permitted values of the system (Borras, 2006). The manual was incomplete in its description of the data entry points and it specifically lacked information that informed the user that approaches/sequences or shapes other than what were present in the manual were not allowed. Had there been even a sentence in the user manual that said ‘entries or sequences that overlap or cross-loop or self-intersect would lead in wrong calculations of the dosage,’ the technicians would have been warned and would never have tried to add the fifth block or shield. They would not have been indicted for murder if they would have had some information that deterred them from adding a fifth block.

We learn that documentation is so crucial in such cases. Every software system must have a detailed documentation that explains its functioning to its users. Critical software that poses a significant risk to life should especially have clear, simple, and detailed documentation that explains how the system should be used. If the documentation provided by the software

firm had been accurate or even detailed, the technicians would have been informed of the risks involved and it would have deterred them from entering the unidentifiable shapes into the system. A possibly small detail which could have spared the lives of the 23 people who died should have been present in the manual. The issue that is highlighted by this case study is that not enough thought was given to the manual that potentially impacted millions of lives. A user manual that gives instructions to the people using it, to people who use it on a regular basis to administer harmful radiation was not given enough priority when it should have been given utmost priority. A manual is a document that explains the software system and in most cases, provides instructions describing ways in which it can be used. Simple step by step procedures are provided to explain how the system under consideration can be used. Guidelines that shed light on proper practices to be implemented while using the software system are generally provided (Shand, 1994). Such a document is often extremely helpful to its audience and helps prevent mishaps from occurring. The fact is, such documents do not require a lot of time to finalize and deliver. The information is already known to the development team, all they have to do is to set aside some time to explain the working of the system and describe the correct ways to use it. This useful activity should not take up a lot of time if the team is aware of how the system works and what is allowed or not allowed. Multidata systems did not take the time to invest in a clear, concise document and it resulted in the loss of lives which is heartbreaking. It serves as a lesson to all software creators to take into consideration the impact of the software and its documentation on others. It serves to instill the significance of proper, detailed, and informative documentation of software systems.

4.6. CASE 6: THERAC-25 MEDICAL ACCELERATOR (1985)

As we have seen, software is now present in every domain and has a huge impact on all of us. Each field is touched by software in one way or another. The medical field is a few of the most impacted domains as it has witnessed a radical software transformation. Hospitals, pharmacies, government healthcare systems have shifted from paper to electronic or digital formats. They have started using software as a means of performing several important actions. Software is now being used to replace the traditional manual methods of working in the medical fields. All over the world, hospitals have begun to replace paper information and replace it digital computer

systems that run complex software. With the rising demand for the medical industry to implement software in its framework, most hospitals have been reluctantly forced out of their traditional paper records and hardware based functioning to adapting a more software inclusive approach. More and more hospitals around the globe have started to now maintain digital records of their patients, their treatment plans, their medical histories, their inventories, details of their staff, etc.

The digitization of paper records was probably one of initial steps towards the introduction of software systems in this medical or health field. But it has slowly but swiftly accelerated its presence in the medical field. It has moved from being a tiny part of the healthcare industry to one of the most irreplaceable parts in the field. It has now become the linchpin that holds the health industry together. Over the years, imperceptibly, and subtly, software started to creep into the healthcare system. Soon enough, software systems that helped medical professionals and performed medical tasks started being developed and implemented in the medical domains such as hospitals, wellness centers, etc. Critical medical machines and instruments such as MRI machines, CAT scan machines, ventilators, pacemakers, automated insulin delivery pumps, morphine pumps, software that measures heart rates, and many more items that depended on software made their way into our lives. Soon machines using software such as insulin pumps, imaging machines, full body scanners were developed and were in use. Hospitals, people, medical professionals soon become dependent on software run medical appliances. Everyone impacted by the medical field knowingly or unknowingly began relying on software run medical machinery and appliances on a regular basis. Today, the medicinal world without software in it cannot be envisaged. Software is now at the heart of this domain. Many medical devices that literally hold the lives of patients in its hands are now circulating in the world.

As software gains more and more importance, it should be noted that it brings along a few issues. But, any software comes with some amount of risk associated with it. Risk involved in case of life critical medical systems is extremely high and hence such software must be properly developed. As software kept on gaining momentum in the medical field, several key issues started coming up. Several flags have been raised over the years regarding the safety of use of software in medical systems. As the quantity of software added to medical devices kept on rising over the years, several key problematic items came up. As computer machines become ubiquitous and control increasingly significant, life-saving complex systems, human

beings are exposed to increased harm and risks. Issues in the software systems being used in the medical and health industry started to come up which showed how dangerous poorly built software systems can be. One such instance that proved catastrophic was an incident of radiation poisoning that harmed 6 people between the years 1985 and 1987 due to the Therac-25 machine, a cancer treatment radiation therapy device. This incident was one of the earliest known catastrophes in the medical field as a result of software malfunction.

The Therac-25 was a radiation therapy machine that was built jointly by two organizations namely the Atomic Energy of Canada Limited (AECL) and CGR (Companie Générale de Radiologie), a French company. The machine titled therac-25 was one of the most sophisticated machines produced in that era. It was the early computerized machinery which was sophisticated and highly advanced for its time. It performed radiation therapy on cancer patients and was programmed to do so on cancer patients. It was a significant advancement in the field of medicine in terms of software use (Thomas, 1994). In its kind of machinery, that is, radiation therapy machinery, it was one of the first machines that was built in such a way that it run completely on software. A few hardware based versions of the same machine were built prior to the Therac-25 machine. One was Therac-6 and another was Therac-20. The main difference between these machines and the Therac-25 was that it was software controlled. This machine had the capability to deliver and administer two types of radiation therapy, a low-power electron beam that is not as powerful as X-rays (also called beta particles) or X-rays or high radiation carrying beams. The Therac-25 is a 25 Million electric volt (MeV) accelerator that worked in dual mode (Leveson and Tucker, 1993). The low power mode was an electron beam that was used for treatment of surface level cancers. This mode was mainly used to treat cancers of the surface that were not located too deep into the tissue. The other mode, the X-ray mode (also called photon mode), had a preset setting of 25 (MeV) for radiation treatment. The X-ray mode was used to destroy cancer cells located deep within the body tissue. This was a one-of-a-kind design where dual mode was supported. The AECL had developed a compact, dual functionality machine that delivered X-rays (photons) at a strength of 25 MeV or electrons (beta particles) at varying levels of energy or strength (Leveson, 1993). Compared to its predecessor Therac-20, it was arguably better, compact, and versatile in nature due to the component of duality and software. All the three versions of this machine were built to work on a PDP-11 computer. This machine was described by AECL as a

standalone real time operating system (Leveson and Tucker, 1993). The Therac-25 machine was built using magnets that helped control the intensity of the beam. It made use of two magnets that folded the electron beams at 180 and 270° respectively before they reached their intended target (Leveson, 1993). The machine was equipped with a turntable that controlled which mode was being used. These magnets were placed on the turntable to control the concentration of the electron beam. When the machine is in low energy or electron mode, magnets present on the table spread the concentration of the beam to a safe level. This mode provided several energy levels ranging from 5 MeV to 25 MeV (O'Brien, 1985, p. 101). In the other X-ray or photon mode, only a single energy level was available that of 25 MeV (Porello, 2012). To ensure that concentration is not too high, in this mode, a beam flattener was built into this machine. This flattener basically helped spread out the radiation to an area of treatment that was consistent in nature. It meant to reduce or flatten the impact of the high intensity x-rays (O'Brien, 1985, p. 101). This component was crucial to the functioning of the machine. If the flattener was not in place, a dangerously high output/dosage of radiation would be emitted, and the patient would be targeted with this significantly high dose. The turning table also possessed a third mode which was the field light positioning mode. This mode served to ensure the position of the patients in a proper manner. This mode was not thought to be a dangerous mode. In this position, no mechanism of safety was employed. This means that no beam flattener was used in this mode as no potential rays of any kind were expected to be delivered in this position mode (Porello, 2012). This positioning is a dangerous situation especially if some error were to occur and if a beam was generated in this mode as no protection is offered. This is a dangerous hazard of use of this machine as this is a risky position (Leveson and Turner, 1993, p. 25). AECL manufactured 11 of these machines. Five of these Therac-25 machines were sold to the United States and 6 were installed in Canada itself starting from the year 1982.

The functioning of the machine was enabled with the help of a computer input system. Using the computer system onboard, different treatment positions on the table could be selected along with the type of the beam and also the strength/intensity of the rays. The computer sent instructions to the machine regarding the type of beam to be fired and the programming of its intensity. The decision whether the beam would be placed or not was taken by the radiation therapist who would set it. In a manual setting, a radiotherapy technician would have to set up these different parts of the machine physically. The technician would be responsible for placing

different parts of the machine such as the turntable for instance. The turntable would be placed by the radiation therapist to place one of the three other parts of the device in the path of the electron beam (Porello, 2012). When the low power beam was used, the magnets placed at different angles would be instrumental in spreading out the beam to cover a larger area that was only surface deep. The magnets controlled the intensity or strength of the beam in electron mode. In the other mode, the X-ray mode, a target, or the flattener was placed in the path of the beam administering the X-rays. The flattener would strike the target and produce X-rays which were then administered to the patient. The flattener controlled the intensity of the radiation administered. This was the second position of the table or mode of operation. Finally, the last mode was similar to a rest mode. In this mode, a mirror could be placed in the beam. In this mode, ‘technically’ the electron beam would never be activated as long as the mirror was in place (Levenson and Tucker, 1993). The goal of this mirror was to help the technician. It would reflect light helping the radiotherapy specialist to direct the machine properly and precisely. The modes are shown in Figure 4.2.

Operation Modes of Therac-25

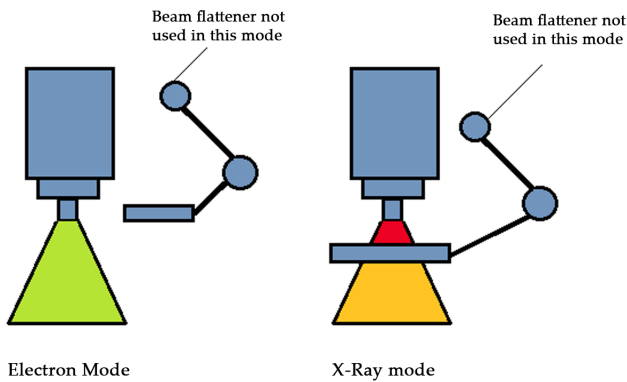


Figure 4.2. Therac-25 modes.

Source: Levenson and Tucker (1993).

The turntable assembly was as follows in Figure 4.3.

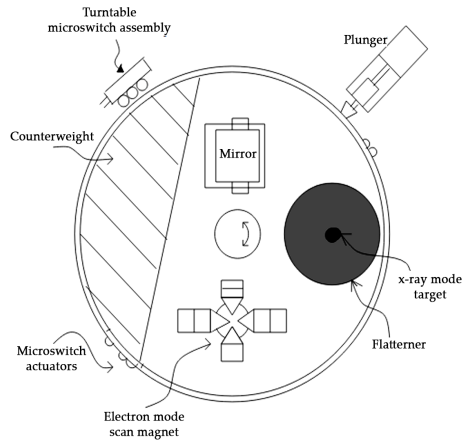


Figure 4.3. Therac-25 assembly.

Source: Leveson and Turner (1993).

In the earlier models of the accelerator, namely Therac-6 and Therac-20, manual setup needed to be done. During the design phase of the Therac-25, the company decided to build the machine to have only computer controls and no manual controls. The earlier versions had manual controls and also hardware fail-safe mechanisms in place. The hardware interlocks were a measure of safety that controlled situations that were deemed harmful or scary. Hardware interlocks were present that prevented dangerous situations from occurring. But, in the development and construction of Therac-25, the manual controls and the hardware interlocking mechanisms were removed. The machine setup in its entirety was to be managed by the computer software. The computer machine or system was given the responsibility of setup of the system, tracking of the treatment, etc. The machine was supposed to handle dangerous situations as well. If it detected potentially deadly situations, the machine was programmed to shut down when such events occurred.

The Therac-25 machine's software component was inspired from the Therac-20 software which was in its turn based on the software from Therac-6. A single programmer, over the course of several years, revised, and reused and translated the software/mechanism of Therac-6 into the new Therac-25 software system. The difference between Therac-20 and Therac-25 was in the role of software in its functioning. In the earlier version, Therac-20, software had a small role. It was a feature that was 'nice

to have.’ It did not perform all the critical actions exclusively. It was a nice addition to the hardware that maintained all the controls. However, in case of the Therac-25 system, the critical tasks are all performed by the software. The software had sole control of all the actions, tasks, and safety checks that ensured smooth operation of the machine. The safety checks that prevented from mishaps were added to the hardware of the Therac-20 machine but not the hardware of the Therac-25 machine. The software was designed to be the sole controlling, safety managing component in this machine.

The software of the Therac-25 machine was responsible for performing the following tasks (Porello, 2012):

- Machine status monitoring;
- Machine setup for treatment therapy plans;
- Taking in input for administering the radiation treatment;
- Switching on the electron/photon administering beam;
- Turning off the electron/photon beam in two cases: firstly after a successful radiation treatment or in case of a software/hardware malfunction;
- Detection of hardware malfunctioning and delivery of diagnostic messages accompanied by a pause in the treatment or a suspension of the treatment.

We can glean from the above list that the software of Therac-25 was responsible for the ensuring the safe execution of the system. It was responsible for making sure that malfunction detection and diagnosis of the machine was done in a correct and safe way. The machine relied completely on software to ensure safety of all the people receiving the treatment and we shall now see how a software system can potentially kill if it is not developed properly.

The Therac-25 machine started operating in 1983. Initially, no issue was noticed in its working. For 2 years, it worked well without any known complications. Several thousands of patients had been treated for cancer using this machine. None of the patients that were treated in this time reported side effects of a deadly nature. No problems were faced and all was good. The very first event showing error in the machine occurred on June 3, 1985. A woman was receiving radiation treatment in the Kennestone Regional Oncology Center in Marietta, Georgia. She was suffering from breast cancer and had been receiving treatment there. She was prescribed by the doctor to receive 200 Radiation Absorbed Dose (rad) by way of a 10

MeV electron beam. But this was not the case. When the machine started, she felt a huge burn. She even informed the operator of the machine by telling him “You burned me” (Leveson and Turner, 1993, p. 22).

She did not know yet, but she had been burned by the machine. She had been unknowingly exposed to a high dose of radiation somewhere in the range from 10,000 to 20,000 rad which is fatal if applied to the entire body. The patient developed reddening and swelling in the central part of the area where the radiation treatment was administered. Initially, this was not even considered to be a bug related to the computer system. The cause for the patient’s burn and subsequent health issues was not determined by the hospital. The AECL denied the possibility of the machine burning the patient. The manufacturers of the machine refused to believe that the patient had suffered a radiation burn. The patient’s redness and swelling and reactions were then attributed to a normal treatment reaction. The patient, over the course, began to have spasms in her arm and her shoulder began to freeze. Although she was hospitalized for this trauma, her doctors, not realizing that the radiation treatments caused her condition, kept on sending her for further treatments using the Therac-25 machine. The patient lived, but lost her left breast and the use of her left arm and shoulder due to the radiation (Leveson and Turner, 1993, p. 22). It was later estimated by the physicist at the Kennestone facility that this patient had received a dose of somewhere between 15,000 and 2000 rad once or twice.

The second accident related to the machine occurred in Canada at the Ontario Cancer Foundation clinic on 26th of July, 1985. A 40-year old patient had visited this cancer center to receive radiation therapy treatment for cancer of the cervix. This was her 24th treatment using the Therac-25 machine. The machine was powered on by the operator, but it shut down 5 seconds later with an ‘H-tilt’ error message. The console of the machine indicated that the patient had received no dose and that the treatment was paused. The operator thinking that the patient had not received a dose hit the ‘P’ or ‘proceed’ button so as to ask the machine to deliver the dose correctly the next time the machine was started. This malfunctioning was considered normal by radiation therapists or technicians and as they were not causing the patient any harm, they were usually ignored. Pushing on the ‘P’ button was the standard procedure of operation in such cases. In this case, the machine kept shutting down. The technician kept pressing the ‘P’ button to direct the machine to dose the patient. This cycle continued for 5 times. The machine kept shutting down after reboot and each time it was restarted, the operator pushed the ‘P’ button. After the 5th time it was rebooted, it

paused and the machine went into the ‘treatment suspended’ mode. Every time the machine was started, it showed that no dose had been given to the patient. The technician of the machine, who was used to this behavior of the Therac-25 system, called the service technician, who was unable to find an issue with the system. The lack of results or issues uncovered by the technician was also a common event. After this round of treatment, the patient had issues. She complained that she had an issue in her hip. She said she experienced an “electric tingling shock” in her body in the hip. On the 29th of July she returned to the hospital for more treatment and informed the doctors that she was feeling burned, she had pain in the hip and extreme swelling in the region of impact. Eventually, it was suspected that was given a high dose of radiation. She was hospitalized for her condition as overexposure to radiation was suspected. The patient died on 3rd November 1985 due to a virulent cancer (Leveson and Turner, 1993). The autopsy performed on the patient revealed that she had died from cancer. But, the autopsy on the patient also revealed radiation overdose. Had she not died of the virulent cancer and survived the cancer, she would have needed a complete hip-replacement surgery in order to survive (Leveson and Turner, 1993, p. 23). It was later estimated that this patient had been overdosed with a high dose of radiation ranging from 13,000 to 17,000 rads.

The third incident occurred in the month of December in the year 1985, where another woman was burned by radiation from the Therac-25 machine at the Yakima Valley Memorial Hospital in Washington. She was receiving radiation therapy for cancer located in her hip. When she received her treatment, she developed excessive reddening of the skin in the treatment area. The reddening of the skin was visible in the form of two parallel stripes on her hip. The parallel striped burn marks on her skin matched the blockers on the beam of the Therac-25 machine. The grooves in the blocking trays present in the Therac-25 machine could have caused the pattern, but the trays had been discarded by the time the staff thought of it. The staff also enlisted the help of a technician via phone who was thought that the machine could not malfunction. The hospital staff was unable to find the underlying cause of the skin reaction of the patient. The response of the AECL led the staff to think that the patient’s reaction was not caused by the machine. This was because the company had installed extra hardware in September 1985 to ensure safety and claimed that the machine was now 10^5 times safer than its previous version (Leveson and Turner, 1993). The worst part was that the patient continued to receive radiation therapy treatments even after this reaction. The doctors, thinking that the machine was extremely safe, kept prescribing it for her cancer treatment.

The hospital staff never suspected that the patient had been burned by radiation. Eventually, a year later, after another similar incident took place at the same place, the doctors suspected that the first was caused due to the machine and that the machine had overdosed the patient. Radiation exposure was discovered after they reached out to the patient. They tracked down the patient and discovered that she had developed necrosis of the tissue under the skin, she had a chronic skin ulcer, and that she was in constant pain. Luckily, they were able to help her by repairing her skin by means of surgery and skin grafts. She needed the skin grafts as they helped close and repair the damage to her skin that was caused by the radiation burns. Her symptoms were relieved, and she somehow survived the overdose. It was later concluded by the hospital that her exposure was not too high as the symptoms appeared 6–8 months after she was burned. She suffered minor disability and scarring of the tissue, but it could have been worse. The placement of the beam, the targeted area of exposure of radiation also determined how it affected her and also saved her from developing severe complications as a result of the overdose.

The fourth casualty occurred in the state of Texas at the East Texas Cancer Center on 21st March 1986. A male patient was receiving radiation therapy for cancer on his upper back region. He had come in for his 9th scheduled treatment for removal of a tumor on his back. His prescribed treatment was a 22 MeV electron beam with the dosage of 180 rads on his upper back and a little left of his spine. He was directed to the treatment room and was asked to lie face down to receive his standard dose. The operator closed the room and sat at the control terminal. The experienced operator was capable of entering the prescribed dosage and able to edit it comfortably using the ‘user-friendly’ editing features provided by the Therac-25 software. She realized that she had entered the dosage incorrectly; the dosage was for X-rays instead of electron rays. She had typed ‘x’ indicating X-rays instead of ‘e’ for electron mode. She used the feature provided by the software which is known as the “cursor up” which is a key that helps edit and change the mode entry. She quickly pressed the ‘enter’ key on the keyboard and started the treatment (Porello, 2012).

Now, as happened in the previous cases, the machine shut down and indicated a treatment pause. The machine also displayed a “malfunction 54” message of error on the screen. The technician verified the sheet provided on the side of the machine and it said that this specific error message meant that it was a ‘dose 2’ error. No information was given in the manuals. The operator saw that the screen showed an underdose and hence she was prompt

to hit the ‘proceed’ or ‘P’ key so as to continue the treatment process. The same sequence of events repeated itself. It gave another “malfunction 54” error and the display screen still said that an underdose was given to the patient. It was much later clarified by an AECL technician that it means that either a dose too high or a dose too low had been administered to the patient. No other information was provided in the instruction manual or other documentation provided by the manufacturer AECL. This message was not supposed to be used in production. It was meant to be used only internally by the company during development.

So, after the first pause, the technician repeated what she did before and pressed the ‘P’ button. The operator of the machine had no direct contact with the patient due to the usual and necessary audio-video monitors being disabled as they were not working correctly. The technician had no direct lines of communication with the patient. When she first started the machine, the first attempt to administer the treatment, the patient felt like he had received an “electric shock.” He said that he felt like “someone had poured hot coffee” on his back (Leveson and Turner, 1993). As he was a patient that received regular treatments of radiation, he knew this feeling was not right. He knew that that the treatment should not cause him to feel burnt. He started to get up from the table but it was at this moment when he was targeted with another beam. This was the second attempt of the operator to deliver the radiation treatment. The beam hit the patient’s arm and he felt a tremendous shock in his arm, and felt that “his hand was leaving his body.” He managed to reach the door of the treatment room and pounded on it to get the technician’s attention. He was shaken and upset and was examined by a physician from the hospital (Leveson and Turner, 1993).

The doctor or physicist, who examined him noticed reddening of the area of treatment but attributed it to nothing serious and concluded that it was probably just an electric shock. The shaken patient was then discharged and asked to contact the hospital if his condition worsened in any way. Over the weeks that followed, the patient continued to suffer painfully. He had pain in his neck, shoulder, and had bouts of nausea and vomiting. He lost functioning of his left arm. He was hospitalized for radiation induced complications to his cervical cord which caused him to be paralyzed in his left arm, both his legs, vocal cords and left diaphragm. He also suffered from neurogenic bladder and bowel and had developed lesions on his left lung along with skin infections. He suffered a lot, lost use of almost all of his limbs, couldn’t speak, and had several other serious issues. Eventually, five months after the incident, he died (Leveson and Turner, 1993, pp. 27, 28).

After this incident, the Therac-25 machine was shut down for testing. Fritz Hager, a physicist notified the AECL of the issue and possible overdose. A local engineer from AECL and another engineer from Canada came to investigate and began testing the machine. They were unable to reproduce the “malfunction 54” error and suggested that the error was caused due to an electrical problem in the machine. An independent firm checked the machine for electrical issue. Their report stated that no electrical issues were found in the system and that the machine was not capable of giving an electric shock to the patient (Turner, 1993). After this report, the physicist at the ETCC put the machine back in service on 7th April 1986 as he was convinced that it was functioning correctly. Fortunately, he was proven false.

On 11th April 1986, three weeks after the previous incident, 4 days after the machine was put back into service, another male patient came in for his radiation treatment at the East Texas Cancer Center. We have to remember that the patient from the previous incident at the same center was still alive at this point. Doctors never suspected radiation exposure at this point. The male patient at this time was receiving treatment for skin cancer located on the side of his face. Fortunately, the same technician who delivered the previous patient who had complications was the one handling the machine. She began to prepare this patient for treatment. She proceeded as she did for the previous patient. She entered the prescription data and noticed that the ‘X’ for X-ray mode was entered instead of ‘e.’ She used the cursor up key to correct the ‘x’ to ‘e’ and saw the ‘BEAM ready’ message on the screen. She turned on the beam. Again, this time, within a few seconds of turning on the beam, the machine shut down and displayed the same “malfunction 54” machine on the screen (Leveson and Turner, 1993). When the shutdown occurred, a loud noise was heard through the intercom as the audio-video functionality was now working in the room. The technician then rushed into the room and found that the patient was moaning in pain. The patient started removing the tape that was secured to his head to keep it in position and told the operator that something was clearly not right. The patient told the technician that he felt like his face was on fire. The technician immediately notified the physician that something went wrong during the treatment. The patient felt ‘fire’ on the side of his face that was hit by the beam. When the physicist ask him to describe what he felt, the patient said that his face was hit with something, that he then saw a light flash and that he heard a sizzling like sound that reminded him of eggs being fried. The patient had suffered from a serious radiation burn and he suffered from disorientation, fever, a coma, and severe neurological damage. His brain stem and right

temporal lobe were damaged due to acute radiation burns. He succumbed to his injuries due to radiation overexposure on 1st May 1986. This was the fifth such incident involving the Therac-25 machine.

The sixth and last traumatic incident occurred later in 1987 on the 17th of January at the Yakima Valley Memorial Hospital. On January 17, 1987, an operator placed a patient on the turntable in the field-light or default no beam position for the purpose of verification doses of small position. The machine, on this particular day, on activation for administering the dose, shut down with a malfunction message and went into the ‘treatment pause’ mode. The operator naturally assumed that the dose had not been given and proceeded to press the ‘proceed’ or ‘P’ button. When the technician did this, the machine halted again and went into pause mode. The display screen stated that the patient had received a dose of around 7 rads. But, the operator heard the patient say something over the intercom. The technician or the operator could not decipher what the patient was saying and went into the room to speak with him. The patient said that he felt burned. He complained that he left a sensation of burning in his chest. Four days following this event, he developed redness and a burn over the treatment area. He developed a pattern that was striped on his chest and this pattern was similar to that seen on the patient treated a year ago where the cause wasn’t determined. The investigation speculated that the beam was somehow present on the turntable in the field light mode but this condition could not be reproduced by the technicians. It was later estimated that the dose received by the patient was something in the range from 8000 to 10000 rads. The patient suffered from severe radiation exposure and died in the month of April, three months later, as a result of complications from the radiation overexposure (Leveson and Turner, 1993, p. 33).

It was the physicist at the East Texas Cancer Center, Fritz Hager, who played a vital role in discovering the cause of the radiation exposure. He was determined to find the root cause of the issue of the machine. After the second incident at the Texas Cancer Center, as he noticed that the same technician was present for both the incidents, he contacted her and asked her help in recreating the issue. As the operator described the way she had entered the data, they were able to reproduce the bug and achieve the “malfunction 54” issue. Both of them worked through the weekend to discover what the issue was. The issue started with the cursor movement. The VT-100 console that came as a part of the Therac-25 machine was used to enter the radiation dosage prescriptions and permitted the movement of the cursor using the cursor up and cursor down keys on the console. What

happened is, by, default, the X-ray mode was selected. So, when this mode was initially selected, the machine would then start the setup process for the X-ray mode to deliver high power radiation via the beam. This process of setup for each mode took around 8 seconds to finish. Once the phase of data entry phase was labeled and marked as complete by the software, the magnet setting phase began. But, during these 8 seconds, if the user of the machine changed or edited the mode, the turntable failed to move over into the correct position. The issue was that if some edits were done in the Data Entry phase in small 8 second interval or the magnet setting phase, the new setting from the data entry setting was not applied to the machine hardware. The turntable was left in an unknown predicament or condition. This is a modern day 'race' condition where the system doesn't know which state it is in and hence reacts poorly. The OS had a subtle yet deadly race condition, and due to this condition, a radiation therapist could unknowingly configure the machine in a way that would make the electron beam launch in the high power or X-ray mode and the patient would be targeted with this beam of high power radiation without any shielding to protect him. The interface of the machine incorrectly displayed the wrong mode to the operator who would proceed to administer the lethal dose of radiation to the patient. The physicist and the technician figured out that data entry speed played a major part in the overdose. If data was entered at a fast rate, the overdose occurred which was deadly to 3 out of 6 patients.

We should remember that testing of the machine had been done in a slow and careful manner. As this bug was reproduced only in a small 8 seconds window, it was an unusual bug. As the nature of this particular bug was slightly unusual and needed certain conditions to occur, regular slow deliberate testing would never have figured it out. Someone who had experience working with the machine, someone who worked with the machine on a daily basis was needed to reproduce it. It took someone familiar with the system and the data entry process to uncover the bug. Fritz Hager repeated the same sequence of events followed by the technician; he practiced and was finally able to generate the malfunction 54 error by himself. He verified that the display screen showed no error and stated that no dose was given. He discovered that a significantly high dose had been actually given to the patients. He spent his entire weekend on the phone with the AECL technicians to explain his result and findings to them.

Even though he was armed with the powerful proof, it took several calls, faxes, and detailed instructions that were required to send to the AECL technicians for them to reproduce the error. Eventually, the AECL after 2

days, after being told that the operation of changing the mode via the cursor up method had to be performed in less than 8 seconds, the overdose occurs, were able to regenerate the “malfunction 54” error. An engineer from the company later explained that frying sound heard by the patients was most likely the noise generated from the ion chambers being saturated due to the race condition.

Soon the FDA was involved. They declared the machine to be defective on 2nd May 1986. Investigations started and lawsuits were filed against AECL. The manufacturer of the machine, AECL, ultimately was subjected to a class 1 recall from the FDA. A recall of this level is rare and it is an event that happens only when the FDA believes that through continued use of the device there is a great risk of death or serious injury. They also started facing several lawsuits from the families of the victims and thus began a long road of trials and tribulations for them.

A staff physicist working for a cancer center in Chicago was able to prove that this bug, discovered by Fritz Hager, was also present in the software of the previous machine Therac-20. This physicist followed the exact procedure that was done by Fritz Hager on his machine, the older version. He saw an error that was similar to the one witnessed by the other technicians and physicists and a fuse in his machine was blown when he did this. The issue was that this fuse was a part a hardware interlock process. As we have seen earlier, the hardware interlocking mechanism had been removed in the Therac-25 machine which is why the error occurred. This indicated that the fault had always been there in the Therac-20 machine but was never discovered. This is because the hardware interlocking caused the machine fuse to blown which was just a minor inconvenience and was considered to be just a nuisance. This bug was always a part of the codebase but was never discovered as it did not have the capacity to kill. Therac-20’s software bug was protected by the hardware interlocking mechanism and luckily did not kill anyone. The detail that this dangerous bug was present in the previous model was uncovered 2 months after FDA recalled the machine. The mere presence of the safety features in the hardware means that the race condition had not been detected as the hardware would simply blow a fuse. A reboot of the machine would be done and the error would never come to light. As the error was not known at the time the Therac-25 was developed, the code was just copied. The assumption that was made was that the code was solid and had no errors as such and that it had been well tested. Hence, it was simply incorporated in the Therac-25 code without a lot of thought.

Also, the investigation revealed that the sixth casualty that occurred was due to another error in the machine's software. Another bug was present in the software which was a concurrency related issue. This was yet another race condition in the software interlocking mechanism of the system. Investigators speculated that the last overdose incident that occurred was the same as what happened in the Hamilton center as well due to the same bug. The incident at Hamilton was attributed to microswitch failure which actually did not solve the issue. This race condition occurred in the later part of the code that performed the setup. The error was caused due to an overflow in a shared class 3 variable of 1 byte.

Prior to the process of firing of the beam, the operator is required by law to position the machine directed at the patient's area of treatment. This is the one of the prerequisites. The entry parameters are further verified by the operator of the machine with the help of keystrokes. The next step is to press the button 'set.'

This button directs the hardware of the machine to the correct location for treatment. But, there was an issue in the verification process. There was a class 3 variable that decided if the configuration of the hardware was correct or faulty. This variable was a shared variable that was used by the subroutine. If the value of this variable was non-zero, it meant that a hardware failure had occurred. If the value of this variable was zero, it meant that the setup is correct and that the treatment can start.

However, the issue came with the variable type of this variable which was a byte. It can hold only 255 values starting from zero (Leveson and Turner, 1993).

It should be noted that the code or subroutine for the setup ran for over hundreds of time. As the upper limit of this variable was 255, it was reset to 0 at the 256th attempt. However, if the operator pressed the 'set' button at the exact same time as this variable, the code was led to another path in code. This path in the code would fire the high power X-ray beam, as the value of the class 3 variables was zero.

What happened was that when the "set" button was pressed, a different code was executed. This portion of code fired the beam with high powered radiation without any shield to the patient. Sadly, this event occurred. A lethal dose of extremely high powered radiation was delivered to the unsuspecting patient who later died as a result of this overdose (Leveson, 1995).

A summary of all the accidents is provided in the table below (Leveson, 1995):

Date of the Overdose	Location	Impact on Patient	Response to the Incident
3 rd June 1985	Marietta, Georgia	Breast removal and loss of use of arm	Incident not recognized as an overdose until after the incidents at Texas
26 th July 1985	Ontario, Canada	Total hip replacement was needed if patient had survived	Overdose still not suspected as cause of issue until patient returned with complains. Microswitch malfunction suspected as cause of issue.
December 1985	Yakima, Washington	Patient survived with minor disability and scarring	Incident not recognized as overdose until a year later after the second incident occurred at the same place.
21 st March 1986	Tyler, Texas	Death of the patient due to severe radiation burns	Malfunction 54 error couldn't be reproduced by AECL. Electrical surge was suspected as cause of overdose.
11 th April 1986	Tyler, Texas	Death of the patient due to severe radiation burns	Initially editing error was suspected. But later the bug was discovered by staff.
17 th January 1987	Yakima, Washington	Death of the patient due to severe radiation burns	All systems were shut down. Investigation started and rework was in progress.

The aftermath of this incident involved a lot of lawsuits for the AECL to handle. The FDA declared the machine to be defective in May 1986 and a class 1 recall was done. After several months of back and forth between the FDA and the AECL, several patches and hardware related updates on the machine were performed by AECL. Eventually, by July 1987, the Therac-25 was given the green light for operation. Several revisions of the Corrective action plan were first implemented until all the requirements were fulfilled (Fleischman, 2010). The machine was permitted to return to active service and no accidents concerning the device have been reported since.

It should be heeded that this was one of the deadliest software bugs of that time. The software bugs present in this machine resulted in 6 separate overdose incidents killing three people and causing serious health issues for others. It would have probably killed 4 if the patient had not died due to other natural causes. Even though this occurred in the late 1980's, this incident has marked the software community for decades to come. This is

because, even though software has evolved over the years, the problems faced still remain the same. They just manifest in different ways. A lot that was learned as a result of this incident is still relevant today. We shall now see some of the lessons that were learned from this series of mishaps.

One of the first lessons learnt is that software should be properly developed. The impact of such critical software must be discussed and impressed upon the developers that are designing it. Any software that has the potential to kill should not be taken lightly. Everything must be checked and double checked so as to ensure that no casualties occur. A complex system that would impact millions of people should have code that is written by at least more than one person. It is obvious that a single person alone cannot be responsible for developing a life critical system capable of killing. It is the minimum that can be expected of an essentially complicated system that could be lethal if not built in the proper manner. In the case of the Therac-25 one of the issues was that the operating system of the machine was a custom built operating system that was built by a single programmer. A life critical device that was lethal if used or operated in the wrong way was developed and built by just one developer who was not formally trained. His credentials are not known to this date. The lone software programmer responsible for building the software of a lifesaving machine took code from the Therac-6 and Therac-20 machines and wrote the software for the Therac-25 system.

Experts that were given access to the code of the Therac-25 were shocked by what they discovered. First, they discovered that a single programmer had written the code. Secondly, code reuse was done without any thought of the impact; thirdly, the software seemed to be written by someone who was not used to developing real time software. The programmer that worked on the development of the Therac-25 software left the organization in 1986 (Borras, 2006). The manufacturer, AECL never released any information about the developer and his respective credentials in the software field.

Furthermore, there was no evidence found that any timing analysis was performed and comments in the code were rare. Further, there was no proof of sufficient software testing and system testing that was performed on the machine. The claim made by AECL regarding the safety of the machine was unfounded. Sufficient evidence of testing was not found by the investigative team. Moreover, the claim of AECL when they added machine fixes that justified a safety of a 10^5 times the previous version was unfounded and false. This brings us to the next issue which was their blind faith in the design of the machine they built (Fleischman, 2010).

From the onset of the incidents, the manufacturer, AECL refused to believe that there could be an issue in the machine. Even though they had a lawsuit and someone claiming that they were burned, they refused to investigate the issues reported to them. In addition, they constantly refused to believe that their machine could malfunction and burn someone which was sufficiently evident had they bothered to look into the very first incident. Despite finding several issues and fixing them over the years as the accidents occurred, they denied that the machine they provided could burn someone and overdose them. Most of their claims were unfounded and groundless. Despite having reports and lawsuits that were filed as early as 1985, they never looked into the malfunction. They were convinced that their machine was perfect and could not possibly misbehave. And, as long as they were blindly faithful to a clearly defective device, they were never going to go look for faults in their own system. They refused to register and accept the hard truths they were confronted with. Overwhelming evidence was given to them and yet their engineers claimed their machine was 100,000 times safer than before (Leveson, 1995).

The company showed a high level of confidence in their software. The manufacturer simply thought that the software is foolproof and that it could never fail. They sadly believed that their software was highly safe and could not possibly fail which probably made them complacent. Their relied heavily on their software system and its functions and refused to entertain the thought that their system was harming people in some cases. Additionally, they were so overconfident in their software system, when problems crept up; their first thought was to look for issues in the hardware and not the software which is appalling. The worst part is that the safety analysis and review process did not include software, it was only hardware based which is inefficient considering the fact that the machine depended on software for all of its safety features. The hardware safety mechanisms had been removed which meant that the software controlled the safety. Despite having this information, the engineers at AECL never suspected the software which is a sad state. Even though problems came up, the investigative team only looked at the machine's hardware as a possible cause of error and never even entertained the possibility that there was a problem in their software.

Even though a physicist discovered the bug all by himself, he had to spend an entire weekend communicating with AECL engineers who were unable to reproduce the error in the system. At different times throughout the incident, improper claims were made by them and each time the diagnosis made by their engineers was incorrect. It took someone objective and determined; a third party person to find the lethal bug in their system

with no involvement in its development process. This scenario is something that should never happen. Never should a provider of any hardware or software refuse to accept that their software is not perfect. As we know now, no software is without issues and it is impossible to develop a code that is perfect. This should always be understood by all parties involved, especially by the people who develop the software. Sadly this wasn't the case and as a result six people (with more suspected) were unsuspectingly hit with lethal or almost lethal doses of harmful high powered X-ray radiation. The patients were overdosed with over 100 times the normal dose and it proved to be lethal in 3 cases and caused severe bodily harm to the others.

Improper and insufficient testing must not be done is definitely the next lesson to be taken from this unfortunate series of accidents involving the Therac-25. As noted by the investigative team, there was no proof found that adequate testing had been performed on the machine's software. The team that was responsible for developing the software related to this machine had no tester in place. Everything was done and developed by a single developer over the years. No formal analysis on the system was done. The code wasn't reviewed and extensive testing of the system wasn't done. The machine was not thoroughly tested despite the fact that it was a critical machine that could target patients with high powered radiation.

In one of the depositions of the supposed quality assurance manager at AECL, he stated the machine's software and hardware had been tested over the years. He said that a small part of the software was tested on a simulator and that the system was tested as a whole. This means that only a tiny part of the software code was run using a simulator, not the whole thing. Further, only system testing of the software was done. No unit testing of the functioning of each feature had been conducted. Most of the effort involving unit testing was minimal and more time was devoted to the integrated system testing. The machine was only tested for 2700 hours which meant that it had been tested in use for that time period only. The safety claims made by the company could not have been met with just 2700 hours of running of the machine. The investigative team was unable to find a definitive and concrete test plan for the extensive testing of the software of the machine. No evidence of any regression testing activities done on the system was found by the investigators (Leveson, 1995). The developers and project team responsible for the system failed to test the system in a detailed manner that would have helped make the machine safe. They did not give importance to detailed testing of the system which led to several overdoses. The investigation into the accidents concluded that the testing of the Therac-25's software was not properly done.

The investigation revealed that the testing methods used were highly inadequate. It was reported that the software of the machine was ‘run’ for 2700 hours, some of it on a simulator and this testing had removed all of the bugs in the software system of Therac-25 (Leveson, 1995). This statement was not fully correct, if we look into it. During the trial and depositions, an operator testified that shutdowns and error messages were frequent when the machine was in use. This means that no analysis of the design of the system was done as it allowed a flaw in its requirements to be developed into the code. This flaw was allowed to go into operational mode as no review had been done. The flaw was a great harmful one as it was a fault that resulted in the operator being misinformed about a possible overdose which is highly risky. As mentioned earlier, no definitive testing plan was uncovered in the investigation. The lack and shortage of any formal standardized testing plan confirms that no detailed testing was done. The modules weren’t tested properly and the error was incorporated into the system. If the flaw in the testing approach would have been identified earlier, there is a chance that something could have been done to address the issues. If it had been discovered that there was not enough testing done, there is a good likelihood of the critical faults in the software being uncovered before the machine was deployed before it harmed anyone.

If the testing had been done properly, those fatal bugs that killed people would probably have been uncovered well before the machine was commissioned and put into service. Additionally, timing analysis was never performed on the machine. This is why the engineers at AECL were never able to reproduce the bug to begin with. The error that led to the overdose was reproduced under certain time sensitive conditions that could only have been detected if proper testing with time sensitive operations was conducted. The fatal bug that was responsible for the massive and fatal overdoses mistook the type of radiation to be used and would occur only under specific conditions. The operator of the machine had to have the wrong entry in the first place and then he should have been able to fix the entry within 8 seconds to reproduce the bug. This means that someone who was familiar with the system could find the bug. The testing team should have been responsible for such timed testing which was never done.

If AECL had conducted and written proper test cases, they would have probably identified the error. If they had implemented a proper test plan that included analysis of timings and responses, there is a high probability that the bug would have been identified early on. But as is the case with most software projects, when real time projects are implemented, sufficient testing

is not done to ensure that timing constraints are met. Even though at that time in the evolution of software development, testing was not an intrinsic part of the cycle, it should have been given high importance simply due to the nature of the device. The device was a critical device which would go on to impact millions of lives and hence it should have been tested thoroughly considering all possible permutations and combinations.

Extensive testing should have been done to ensure that the safety is maintained. As extensive and detailed testing including a test plan wasn't implemented, the bug encountered by the technicians was difficult to reproduce by the engineers at AECL. Poor testing practices coupled with overconfidence in the capabilities of the machine led them to arrive at the blind sighted conclusion that nothing was wrong with the Therac-25. They could not accept the fact that something could go wrong with the machine and this logic was inherently flawed. Six innocent people were harmed as a result of poor management of the project as the incidents were never investigated in depth.

The next and obvious less to learn from this catastrophic series of unfortunate accidents is that poor programming must not be implemented in software projects. Especially in case of life altering software systems, the people developing the code must be properly and formally trained. Programming certification should probably be made mandatory for developers who are supposed to work on such crucial software. The incidents were caused as a result of shoddy and subpar programming work that was done in the software of the machine. The bug, the race conditions in the system was a result of nothing more than poor programming done by someone who did not have effective programming skills. Much of the software standards were not implemented in the code which led to subtle issues in the code that turned deadly. Anyone with proper knowledge of developing real time systems would have ensured that the race condition would not occur. Experienced and well-trained programmers would be able to assure that safety has been met and that the real time events would be independent of one another.

In the case of the machine's software, first, and foremost, only one developer worked on the development of the code which was not a correct management decision. Secondly, investigation revealed that the person who worked on the code did not have formal training and also possessed little experience in the development of real time systems. Also, the project lacked proper specifications which could have hindered the programmer

from writing effective code. In many software projects of a non-critical nature, poor programming skills go unnoticed and generally do not cause harm to people. This is one of the reasons programmers continue to write shoddy code as they were never required and asked to write proper code that followed the industry coding standards. This should not happen when lives are at stake. In this case, a single person wrote the code and there is a high chance that no senior and experienced programmer sat down with him and reviewed the code he wrote. Poor management also played a part.

Sometimes, due to poor management of the project, developers are pushed into developing code at a faster rate. This could probably lead them to develop unstable code. Sadly, project management plays a significant part in the outcome of any software project. In the case of the Therac-25 system, the management did not give proper value to the software component of the machine which was wrong. They most likely assumed that since the code was not the main feature of the system, since it was not a huge part of the entire machine, it could be implemented without the same level of seriousness as the hardware. As previous versions of the code from its predecessors were used, and as no issues were faced until then, it was probably assumed that the code is safe and that it would be easy to incorporate. Programming skills and the need to implement good quality code was probably never even given a second thought. The real reason behind the decisions made by the management still is not known but it would be safe to assume that they were somewhat responsible for the poor programming witnessed in the source code by the investigative team. The takeaway is that proper programming should never be considered as an afterthought. It should be incorporated in the code from the start so as to avoid bad software behavior that is observed as a result of such programming in the code.

Yet another issue was that the single programmer performed all the tests on the software which is not a good practice. The person who developed the code, ideally, should never be responsible for performing anything other than unit tests. If a single person is developing the code, it must be tested by another person who can look at the system differently. The testing done by another person is always a better choice as the person who developed the code is less likely to be motivated to find issues in the code he wrote. This is one of the oldest practices that have been part of the software engineering principles since a long time ago. The management did not invest in a proper test plan and a testing team. The vision and point of view of a single programmer is insufficient when it comes to testing a critical system such as the Therac-25. The project leaders should have realized that testing and development must have been done by different parties. This probably could have saved the people who were overdosed.

The next lesson which is important and significant that is to be imbibed from these incidents is that software design is crucial to the well-being of the software system. For complicated and critical systems, proper thought should be given to the design of the software. Design of any system that is critical and affects human life in dangerous ways should be well thought of and must follow a simple structure. Design should be such that the software could be tested. No ambiguity should be allowed in such code that is designed to work in real time. The design must consider the factors involved and the software must be built in such a way that makes it simple to code as well.

In this case, several questions arise regarding the design of the system. The very choice of the implementation must be questioned. Why was assembly language chosen? Was it a prudent choice to write the entire software of the complex machine in assembly language? Also, the decision to write a custom operating from scratch should be questioned. Wouldn't it have been easier to use a known operating system? Or couldn't have it been purchased from a third party and separate vendor? Also, the design choice of eliminating safety controls from the hardware must be reconsidered as well. Was it a good choice to rely completely on software for ensuring the safety of the machine? Was it a good choice to write code implementing real time behavior from scratch rather than seeking an outside vendor?

The magnitude, vastness, and depth of the impact of several of these design choices on the outcome and results generated by the machine can only be guessed at this point. The correctness of these choices cannot be easily decided but probability theory suggests that these choices would definitely have led to significant critical errors. These choices that were made regarding the design were risky to say the least. Possibly, at that time, software had not evolved enough and hence quality compilers were maybe not available or were too expensive. This could be a possible reason for the selection of assembly language for the operating system. It could have been the only logical choice for the developer at that time. The Therac-25 machine's software lacked proper self-check mechanism. There were no error detection or handling mechanisms in the design of the software. The presence of such detection features could have helped in detection of problems in the system before it even went into use. The design of the software was flawed as independent checks of smooth operation of the machine were not built into the code. Such checks cannot be built without having some measures of error detection to begin with. Hence, if such checks were implemented, error detection would have been added as well. Such design flaws in the design of the machine made it a risk to operate the machine.

But sadly, proper risk assessment wasn't done as well. No formal analysis was conducted, pros, and cons of the design approach were probably not considered and eventually it led to a machine being built that killed people by accident.

The next thing to pick up from this case study is that risk assessment is important when it comes to critical machines like a linear accelerator. The Therac-25 machine had a risk assessment done that was probabilistic which led to dangerous incidents taking place. The probability of error that was calculated was low and this translated into the manufacturers being complacent. The issue with probabilistic assessment is that such measurements often exclude different aspects of software that are hard to measure or quantify (incorrect software requirements, for instance). Software functionalities may have a higher impact on the safety and reliability of the system as compared to their hardware equivalents which are quantified (Turner, 1993).

This issue was that the initial or first risk and analysis of hazard completely ignored software. This was a reprehensible decision. The likelihood of software error was treated lightly in a superficial manner and it was not given the same level of importance as probabilistic assessment of the hardware. The probabilistic risk assessment of the hardware created a false confidence in the entire machine as a whole (Turner, 1993). As this value was probably a good value, it was considered that the machine is entirely safe, including its software. This false sense of confidence in the measurement of risk possessed by the system was unjustified.

This overconfidence caused the people at AECL to think no investigation was needed for the first incident at the Yakima center. They chose not to conduct any investigation of their own into the incident as they believed their machine was infallible. The risk assessment done suggested that the safety had increased five times or five orders in terms of magnitude which was sufficient for the engineers to ignore the patient who was clearly burned. The overconfidence was due to the recent updates in the microswitch functionality which was hardware related. No error in software was suspected as no proper assessment was done for the software. This was one of the causes of failure of the machine to ensure safety.

The issue was that no means of determining probabilities of risk included software. As we have seen, no software is perfect and hence the determination of risk probabilities that are accurate is not possible in most

cases where software is involved. The problem was that in this case, as software risk could not be determined, it was assigned the same risk as the hardware which was incorrect. The manufacturers chose to ignore the risk assessment of software which was not a good decision in hindsight. The solution for proper risk assessment lies in proper design combined with proper testing of the system and values that can be quantified. Instead of thinking that nothing could go wrong, the worst case should be assumed. If the worst case scenario is incorporated in the software design itself, the software will be better equipped to deal with problems. The software should be developed in such a way that it can make self-checks and decisions that provide supportable and quantifiable risk assessment measurements. Unfortunately, this was not done in the case of the Therac-25 and it led patients to be overdosed with high powered radiation.

A further issue that was determined was the implementation and reuse of code from the previous software. Code from the Therac-20 was taken and reused in the Therac-25. The developer was confident in the quality of the software recovered from the code of the previous machine as no known accidents had taken place until then. The developer was not aware of problems in the code as safety mechanisms in the hardware masked any underlying errors in the code. So, the developer trusted the code and based his design for the software of the Therac-25 entirely on it. Much of the previous code was reused which was not a good idea. The assumption made by the engineer was that reusing of the software was safe because it was in commercial use and hence was used extensively. Therefore, as it was used often, as no errors had been reported, it was assumed that there were no errors in the software. This was not the right assumption as we now know.

Sadly, false confidence in the activity of code reuse is no new occurrence. This behavior of code reuse is still practiced in the industry today. This practice of reusing code has been in place for decades and is hard to eradicate. Generally, anybody who decides to reuse code has a valid reason to do so. One reason is probably because the code has been in production, and hence, is correct and tested. Another reason is that an assumption regarding the safety of the code is made. An incorrect assumption is made that the code is safe. It is wrongly considered to be safe as it has been in use by users who probably have not faced any issues during the use of the system. This information instills false confidence in the validity and correctness of the code which is why some developers tend to reuse the code.

The main issue with such assumptions is that they do not hold true in all cases. Software is deemed safe or unsafe only under certain conditions of use and when certain parameters hold true. Just because a code is running safely in production, it does not mean that it would not cause problems in another situation. The code alone does not account for reliability and safety of the software. Other factors must be taken into account as well. Just by looking at code, its reliability cannot be determined. The safety of any code is not deducible just by looking at its use. Also, safety in software dependent machines is a combination of other factors in addition to software. A crucial lesson is that just because a code is safe in one software system, does not mean that it will be safe in other systems. This is clearly seen by the code reuse that led to issues in the launch of the Ariane 5 rocket. Another analogy is that just because matches are safe in the hands of an adult does not mean that they are safe in the hands of a 5-years old child. Safety of any system must be thought of as a quality of the system where the software is implemented. It should not be thought of as the quality of the underlying software. This was the mistake committed by the engineer who developed the code for the Therac-25. He linked the safety of the code directly to the safety of entire machine which was wrong. The belief that reused code is safe in all cases is grossly unjustified and cannot be inherently proven to be true.

Sometimes, the opposite scenario turns out to be true. As seen in the case of the Ariane-5 rocket launch failure, software reuse can sometimes lead to dangerous and unstable software design. The reuse of modules in software does not guarantee safety as the new system probably has different parameters that affect the safety of the system. In some cases, when lives are at stake, the better choice is to rewrite the code from scratch with new requirement specifications. Sadly, this wasn't done in the case of Therac-25's code and people were harmed as a direct result.

Another teaching of great magnitude to be gleaned from these mishaps is that in a software system, documentation is extremely important. It is of utmost importance that critical systems have adequate documentation. Not just critical systems, all systems must have proper documentation that explains its working and how to troubleshoot the system. This is much more important when it comes to highly critical software systems as documentation serves as a guideline to the people using the system. The role of proper documentation is crucial in such systems. These mishaps reiterate the need and value of well written detailed documentation. It demonstrates the high necessity of well thought out and mapped documentation in software systems.

The radiation therapy system, the linear accelerator, did not have a detailed manual. The only information provided was a sheet on the side of the machine that was extremely unhelpful to its users. It did not come with a manual that could be used by its operators who struggled to troubleshoot the problems they faced while they used the machine (Leveson, 1995). It was not detailed enough for the people or the operators of the machine to troubleshoot it correctly. It did not contain sufficient information pertaining to different error codes that were experienced during the radiation treatment therapy. The Therac-25 was a critical machine and it impacted the lives of thousands of people; hence, its users or intended audience or operators should have been given the best possible tools to help them operate this critical machine or system in the safest way possible. Not only did the so called 'manual' provided by the AECL system fail to do so, but it also succeeded in ensuring that the machine could potentially kill people effortlessly without any proper warnings or alerts due to a lack of documentation related to the system failures and errors. The technicians or operators of the machine were unable to understand the meaning of different error messages that they received when the machine shut down unexpectedly. The manual was incomplete in its description of the error messages that were shown to the operators in case of a failure. For instance, on one occasion, when an incident of overdose had occurred, the technician received a message stating that a "dose 2" error had occurred. No information was given in the manuals regarding what the meaning of this cryptic message was. Also, due to the fault in the implementation and design of the system, the screen showed that an underdose had occurred which led the operator to re-administer the harmful radiation again. It was clarified later on in the investigation by an AECL technician that this message meant that either a dose too high or a dose too low had been administered to the patient (Borras, 2006). No other information was provided in the instruction manual or other documentation provided by AECL which was not a good thing. Furthermore, this message was not supposed to be used in production. It was meant to be used only internally by the company during the development life cycle. Imagine how bad this situation is. Due to a lack of documentation, the operator did not understand the error code and it led the operator to overdose a patient more than once. This was the worst possible outcome from a lack of sufficient documentation and it is a shame that it happened.

Had there been even a sentence in the user manual that said 'dose 2 error messages means that either an extremely high dose or an extremely low dose has been given to the patient,' the technicians would have been

warned and would have probably paused and checked in with the patient. They maybe would have decided to check the position of the beam and the shields. They maybe would have decided to call in someone to check the system. They may have been motivated to get a second opinion. It maybe could have saved the lives of the people who are now dead as a direct result of the overdose. They would not have responsible for overdosing or killing the patient because they couldn't comprehend the issues they faced while using the machine.

We learn from these mishaps that documentation is so crucial in such cases. Every software system or application must possess a detailed documentation than explains its functioning to its users. Especially, critical software that poses a significant risk to the lives of people should have clear, simple, and detailed documentation that explains how the system should be used in a safe way. If the documentation provided by the company had been accurate or even detailed, the technicians would have been informed of the risks involved. It would certainly have deterred them from re-entering the dosage of the harmful radiation. Another error message that was highly cryptic was the "malfunction 54" error message which was faced by the operators of the Therac-25 machine. Nobody knew what it meant and the documentation was clearly not helpful. The operators were stumped as they tried to figure out why the patient was exhibiting signs of radiation poisoning. A clear document that defined the error codes along with their meaning was clearly lacking. If the manual contained information that described different error codes along with ways it can be fixed, the patients probably would not have overdosed. A tiny detail present in the documentation which could have spared the lives of the people who died should have been present in the manual. At times, the machine would enter pause and enter the treatment pause mode for no explicit reason. The causes of this behavior were not documented. The operators had no idea what to do when such an event occurred. No information was available to them and they responded in a way they thought was safe to the patients. The lack of information caused the operators to unknowingly overdose six patients and this was not their intention. If they had some knowledge of what the pause meant or what happened, they would have reconsidered their actions and would have shut down the machine immediately.

A further issue was that there was a lack documentation regarding the software requirements and the test plan. This is the wrong approach to software development. Documentation of the requirement specifications is primordial to ensuring that the software system is built properly. In the case

of the Therac-25, not enough detail was found by the investigators. The requirement specifications of any critical system or any software system for that matter should always be documented. Documentation helps putting thoughts and ideas to paper and helps make it easier for the developers to build the system. As a single developer was working on the system, this was probably overlooked. It maybe was not considered important as the developer was basing his code on the previous versions of the machine which were functioning without any apparent errors. Complacency regarding the machine's design aided in the system being built without following good programming practices which is abhorrent.

Additionally, no detailed test plan was presented to the investigators of the incident which was alarming (Leveson and Turner, 1993). This indirectly means that until the point the investigators got involved, there is a high chance that no formal or official documentation regarding the approach of testing was present. This should definitely not be the case, in particular, when talking about critical and life-changing systems such as linear accelerators. The team at AECL did not give importance to documentation and coming up with a test strategy. No document describing the tests carried out along with the results was provided. No document that explained how exactly the software had been tested for faults and how the faults had been fixed. This is appalling since the machine was capable of harming people and it did not have any formal proof that the software was tested properly. One quality assurance manager said that the machine had been 'run' for around 2700 hours which is inadequate for such a precarious machine. Also, 2700 hours of execution does not justify the claim that their machine was now safer by an order of magnitude of 10^5 as a result of their fixes (Leveson and Turner, 1993). The developer who built the software was charged with the responsibility of testing and this was not an ideal situation to begin with. It is safe to assume that he probably did not have enough training to do regression testing or even come up with a test plan. Sadly, the complacency and arrogance shown by the company resulted in the sad disasters involving death and the worst part is that adequate documentation could have probably saved this from happening.

The problem that is highlighted by this case study is that not enough thought was given to the manual that described a machine with the potential to impact millions of lives. A user manual that gives instructions to the people using it, to people who use it on a daily basis to administer harmful doses of high powered radiation was not given enough priority when it should have been given high priority. A manual is a document

that explains in clear terms how the software system works and in most cases provides instructions describing ways in which it can be used. It also provides sufficient information regarding how to troubleshoot the issues faced. Simple step by step procedures should be provided to explain how the system under consideration can be used. Guidelines and instructions that shed light on proper practices to be implemented while using the software system are generally given so as to ensure that safety is maintained. Such a document is often beneficial and helpful to its audience and helps prevent mishaps from occurring.

The fact is, such documents do not require a lot of time to finalize and prepare. The information is known to the development team and all they have to do is to set aside some time to explain the working of the system and describe the correct ways to use it. This useful and life saving activity should not take up a lot of time if the team is aware of how the system works and what is allowed or not allowed.

The issue was that in this case, a single person at AECL was responsible for developing the software and there is a high chance that he was stretched too thin and hence ignored the documentation process. AECL did not take the time to invest in a descriptive, clear, concise document about the working of the machine and it resulted in the loss of lives which is woeful and lamentable.

Documentation should not be an afterthought. The problems faced because of a lack of documentation are seen today as well. It should never be taken lightly how information can change the way someone interacts with a software system. It serves as a lesson to all the people who develop software to reckon with the force and impact of the software they create and its documentation on others. It serves to instill and imbibe the significance and virtue of providing proper, detailed, and informative documentation regarding the software systems one helps develop.

Important and crucial information that can be grasped from these mishaps is that incorrect software practices must not be incorporated in the development of any software system. Basic software engineering principles must be followed if software is being developed. The development team (just one developer) of the Therac-25 failed to incorporate some standard well-known principles of software engineering during the development of the software of the Therac-25 machine. Several healthy practices were violated and ignored in the development of the software component of the machine. These oversights directly led to the 6 known cases of radiation overdoses.

The following principles were not respected:

1. Software specifications were not clear and well documented.
2. Software documentation was not implemented and was not clear and easy to follow.
3. Software quality assurance was not done and rigorous practices, standards were not developed and implemented.
4. Software design was complex and no error detection mechanisms were built.
5. The software was not audited periodically and the design failed to include informative means of communication of errors.
6. Inadequate testing of the software was done. System testing was performed but no regression testing or module testing was done.
7. The design of the interface was flawed and did not contain meaningful messages to help the users of the system.

The lack of proper software practices, incorrect implementation, arrogance, and refusal to accept faults led to the mishaps and cost people a lot. Perhaps all of it could be avoided if proper programming and software development practices had been implemented for this project during or before its development even started.

The next but most obvious issue encountered is the investigative analysis done by the engineers at AECL. We learn that debugging, investigation, and reporting of issues via proper channels is a highly crucial skill to possess. Each of the times the operators reported any issue they faced, AECL failed to investigate the actual cause of failure. Every time a patient reported a burn, the AECL was notified. Each time, they were unable to find the real cause of failure. When the very first incident occurred, AECL refused to acknowledge or even investigate the machine. They denied that their machine could burn someone which was clearly untrue. After a lawsuit was filed by the patient, they did not investigate the machine. This is appalling behavior by the organization to say the least. They were overconfident in the ability of their machine and even though they had a case that stated otherwise, they refused to believe that their machine was responsible for the incident and chose not to investigate. Their decision led to further cases of overdose which could have prevented had they tried to understand what had transpired. Further, this incident was reported to the FDA only a year later in 1986 after the news of other accidents came to light which is horrific. The blatant disregard for human life can be clearly seen from the way the firm chose to react to the

incidents. They chose to continue to operate the machine, to not inform their users of any issues despite the fact that an incident had occurred. Further, after the second incident of overdose, they failed deplorably in finding the actual cause of the overdose. They ended up fixing the wrong thing. They assumed the hardware was at fault and fixed the microswitches after which they claimed their machine was now 100,000 times safer than before (five orders of magnitudes) (Leveson, 1995). They could not be more amiss from the mark. They never thought to check or investigate the software for errors and incorrectly thought they fixed the system. This second overdose case showed us how bad diagnosis can be harmful. The engineers refused to check or even think that software could be the issue. It could possibly be attributed to overconfidence in the system or a lack of knowledge of debugging of software. In all cases, in each incident, incorrect and faulty root cause analysis was done. Each time they were presented an error, they initially refused to acknowledge it. Even when they were shown proof, their team was unable to reproduce the errors which shows how inexperienced and inept their team was when it came to looking and fixing errors.

They tried to fix each and every lone problem as it came and never stopped to take a step back to find the root cause of all the issues. Each incident was attributed to different issues when they all were linked to a deeper underlying glitch or bug which was never uncovered by their team. The problems associated with the safety of the Therac-25 were not solved as each bug was ‘supposedly’ fixed. In other words, the engineers merely treated the symptoms and failed to find the disease. They ignored the deep-rooted underlying reason for the failures and ended up identifying the wrong thing as the issue which proved deadly to the patients.

The ability to correctly investigate and look for the underlying hidden issues is highly undervalued in the software field. It is an important skill to have. Sometimes, software is the component that fails rather than the hardware as we studied in the case of Therac-25. This was not considered in the case of this machine and ended up costing people their lives and their loved ones to be hurt and suffer a loss. The claim made by the engineers at AECL regarding the safety of the machine due to the microswitch patch led physicist to believe that the machine couldn’t possibly overdose the patient. The sheer lack of interest and overconfidence of the people at AECL following each accident is reprehensible and ended up hurting innocent people struggling with cancer. In the end, the manufacturers of such life critical machine are responsible for ensuring the safety of use of the equipment that they create as they are supposed to know the equipment

better than others. They are the ones who should guarantee safety which should be built into the product and not added later after issues occur.

It is known that although these accidents on medical equipment controlled by software, the same lessons learned from these unfortunate mishaps can be applied to development of software systems across the platform. Even today, many software systems fail due to these crucial issues that are brought to light by these incidents. As we cannot reverse time and fix what happened, the only logical approach is to learn from the mistakes and move ahead towards the building of safe, secure, and robust software systems.

4.7. CASE 7: HEATHROW TERMINAL 5 OPENING

Another interesting case of software failure was witnessed in 2008 at the London Heathrow airport during the opening of its fifth terminal. A bug in their software system that ran the luggage handling mechanism for British Airways caused hours of delay and cancellations.

The technical issues were not explained by the company, but the passengers were informed that there was an issue with the check-in process as there was a disruption to the baggage system due to a technical issue (BA baggage failure, 2019). Many passengers were asked to add essential items to their cabin luggage as the operators could not guarantee that their luggage would arrive on time. Several passengers missed their flights as they were not able to check-in their luggage on time. The system had slowed down due to an issue in the software of the baggage system.

The baggage system was newly implemented and was meant to carry over 12,000 pieces of luggage on a daily basis. Before the opening of Terminal 5 at the Heathrow airport, the staff and test team had tested this system. This new system for handling baggage was recently built and was programmed to carry enormous quantities of luggage that would be checked in regularly. The baggage system was tested extensively by their engineers. Before they opened the terminal to public, the system was tested to handle a significant amount of baggage. More than 12,000 pieces of luggage were used to test the system. The engineers were sure that its threshold capacity had been correctly tested. The tests were positive. The system worked seamlessly without a hitch during the testing phase. It passed all the tests performed before the opening of the terminal. But, sadly, on the day of opening of Terminal 5, the baggage system malfunctioned.

The system could not cope with the demand and it appeared to become confused. It is speculated that different real-life scenarios weren't well handled by the system. A passenger removing the bag manually from the system had led the system to become confused and had shut down its operation (BA baggage failure, 2019). For instance, when a passenger who had possibly left an important thing in his check-in luggage removed it from the belt manually, the system did not respond well. This action caused the system to slow down and eventually stop working causing chaos at the airport. Several people were affected by this incident of system failure. In the subsequent ten days, following the opening of the terminal around 42,000 pieces of luggage did not travel with their respective owners. Additionally, several flights had to be cancelled or delayed due to the technical issue. The issue was that although complete testing was done and the results were successful, they engineers did not account for different scenarios that would occur in real life. They did not test what would happen if a passenger did something unlikely when checking in their bags. The test did not account for unusual behavior. This is probably what led to the incident.

The lesson learnt from this case study is that in important software systems, detailed testing is a must. The system must be tested at its bounds to ensure that its behavior does not alter when a user uses the system in an incorrect or unusual manner. The boundaries of the system should definitely be tested. For instance, if a system accepts a user input of the values 1 and 0, it should be seen how the system reacts when a user input value of 2 is given to the system or when the user presses the 'enter' key continuously. Abnormal data and input entries must be tested so as to ensure that the system is prepared for all possible scenarios. In this incident at the airport, the test plan did not account for scenarios that would happen in real time which caused the system to react poorly. Hence, it is best to try to include extensive testing that account for user actions with the system so as to ensure smooth and error free functioning of the system.

4.8. CASE 8: CAIRNS HOSPITAL SOFTWARE GLITCH

As software has now taken over most of the necessary services used by people, it comes as no surprise that it is a large part of the medical industry as well. The medical systems all over the world have started to migrate towards software systems for patient's medical devices, storing their records, management of their supplies and so on. Software is being used to replace the age old paper records that were being used for a long time. Hospitals all over the world have been moving towards digitizing their records and

converting them to an electronic format. Now-a-days most hospitals have digitized records of their patients, their medical histories, their treatments and so on. With the digitization of patient records by means of software, there were increased instances of security breaches. One such security breach in softwares is a ransomware by the name WannaCry. An attempt to implement a software patch that protects hospitals against this ransomware in Queensland, Australia caused severe issues in five major hospitals in Queensland, Australia.

The history of electronic implementation of patient records has had several glitches in Australia. The nation has been witness to several failures in terms of digitization in the medical field over the past decade and more. The Federal Government in the country of Australia has ramped up its support for IT in the medical field over the last few years in order to improve the security of software systems in this industry. But sadly, one such endeavor resulted in the loss of hundreds of medical records of five hospitals in Queensland, Australia (Layt, 2019).

Queensland has its own patient history maintenance service that is used for electronic storage of its medical data. The integrated electronic medical record system is the digital software for all the major hospitals in Queensland. It is responsible for storing of electronic patient records by hospitals all over Queensland.

Due to a software error in one of the security patches, this system faced crashes causing their hospitals to be overwhelmed across the state of Queensland. Some of the affected hospitals ended up having to revert back to the old paper and pen system during this crash. The incident occurred in September 2019. This system that stored the digital records cost more than 1.2 billion Australian dollars impacting millions of patients and it was improperly functioning for around 2 hours. During the crash, contingency protocols had to be declared in order for the hospitals to be able to run smoothly.

The following five hospitals were impacted: Princess Alexandra Children's hospital, Lady Cilento Children's hospital, Cairns Hospital, Mackay, and Townsville hospitals. The attempt to update and patch the security flaws backfired in a significant way due to the routine software update that was supposed to go unnoticed otherwise. The medical staff was unable to login into the integrated electronic medical record (ieMR) system for a few hours and it hampered their activity considerably as all of the data they needed was online on that system (Layt, 2019). Although the issue

was resolved in 2 hours, the medical staff was inconvenienced during these hours. The hospitals experienced severe slowness in their systems and it affected patient care.

All the hospitals in Queensland that had implemented the ieMR software were found to be impacted by the crash. The department of health of the state, Queensland Health stated publicly that this crash was the result of a “routine software patch” that had been scheduled by the software’s manufacturer, a company called Cerner (Layt, 2019). Login issues combined with system slowness was faced by all the hospitals that implemented this system. Although the software patches had added a level of security to the systems, it caused an unforeseen issue in terms of login. These necessary protections that had been added by the patch led to severe logging issues in the integrated electronic medical record system. Many users faced difficulties in logging on to the system which impacted patient care.

The glitch affected the Cairns hospital a little more than they anticipated. The failure provoked an internal emergency at the Cairns hospital. They had to declare a ‘Code Yellow’ which stood for an internal emergency and was communicated to its staff via email. Luckily, the staff was able to carry out the surgeries as planned for the day. But 22 outpatient engagements were deferred due to the failure of the system. The amount of time that was estimated for recovery of the system was uncertain and the staff struggled to cope with the data loss. Whenever a hospital faces a Code Yellow, it signifies that the emergency division’s maximum threshold capacity has been attained. This means that additional staff is being called in so as to support and ensure the smooth flow of patients in and out of the hospital (Layt, 2019). Such a declaration also entails loss of essential services. In this case, it meant loss of software systems which they heavily depended on. The login issues caused delays in the process of patient admissions and patient discharge as their data was ‘lost’ or unavailable which meant that the processing times for each such event had increased. This loss of information caused delays in the patient care.

Although the affected hospitals were operating as ‘usual’ they faced a lot of issues as they waited for the health department to work with its vendor company Cerner to fix the issue. The apparent fix for this issue was delivered only a couple of weeks later until which the staff remained on ‘pen and paper.’ Patient data had to be collected via the old written paper and pen method as their online data was unavailable. Patients admitted in these wards were asked to give their data in paper format as the staff did not have access to their electronic versions anymore.

Another hospital that was affected was the Princess Alexandra Hospital which made use of the same ieMR system. At this place, more than 500 medical personnel could not login to the system and had to revert back to the paper form causing significant delays in the patient care. During the course of the downtime that impacted all the hospitals implementing the ieMR system, the medical staff of these hospitals were counseled and ordered to prescribe all essentials on paper. This included emergency life saving medications with their scripts/prescriptions required to be finished on paper as and when required. Operating theatres were impacted as well. Each new patient that was admitted to the emergency room or the operation theater had to have paper records. As paper records were being managed, delays were introduced in patient care that impacted the hospitals for around 2 weeks. The hospitals were impacted and faced huge delays in patient care. The fix for login issues took over 2 weeks to be implemented and they were eventually able to recover their medical records. During this time, regular contingency plans were activated and luckily, there was not any impact on the safety of the patients during this crash.

Cairns Hospital was one of the several hospitals that were chosen as a pilot launch site for the rollout of the ieMR system in July 2019. But in September, due to several failures that occurred in these systems, the program was presumed to be defective within a couple of weeks of its rollout. Most of the staff that interacted with this system commented that this system caused severe impacts of adverse nature on the care of patients and their safety as well. Additionally, other issues were observed by the medical staff. One of the key concerns was the incorrect labeling of medications. This issue posed a significantly high risk to the safety of the patients and the authorities were alerted appropriately about it.

Due to the severity of the issues caused by the system, it was deemed unfit for complete rollout in the state. Due to the continuous issues faced in the 2 weeks of its launch, the rollout was paused. It has been paused until 2021. But, the government ended up losing a lot of capital in the implementation of the ieMR system, which was worth around 80 million Australian dollars (Layt, 2019). This system ended up costing millions of dollars of the taxpayer's income but had little to show for this extravagant spending.

One can learn a lot from this particular case. Software used in critical fields must be well tested and well implemented. The lesson we learn is twofold. Firstly, the software that we build must be robust and secure. It should have enough protection to protect it from security breaches,

especially in cases where it impacts millions of people. If the software used by the health system had been robust, their security would probably not be at risk. The implementation and design of any software system is important to ensure its smooth functioning. If the department of health would have studied and reviewed the design in terms of security, there is a miniscule chance that the security flaws would have been identified. Secondly, what we can glean from this incident is the important feedback that testing is extremely important in case of all software projects. If comprehensive testing of the security patches would have been conducted, the impact of these patches would have been evident in the test process. Even though software updates are sometimes not given a lot of importance, their potential impact must be tested. Their integration with the existing system must be seamless so as to make sure that its addition doesn't cause any issues. In addition to individual testing, end-to-end testing must be done as well. In this case, the impact of the security patches on the rest of the system wasn't tested in detail. Its behavior with the rest of the system should have been tested to make sure that it works without a hitch. The designers and development team should potentially have thought of the possible issues that could occur due to the introduction of this new code. Login functionalities are one of the basic and straightforward items in any software system and hence they must be well tested and designed. Their response times are a usual action item on the list of any testing team which was overlooked in this case. This bug caused a lot of mayhem at several hospitals across Queensland.

Another lesson we learn is that assumptions that impact the entire system should not be made in software development. Assumptions that impact the entire application must be questioned. All assumptions made must be agreed upon by all the parties involved in the project. In this case, an assumption made by the software provider Cerner, was that the security patch would not impact the existing system in any way. This assumption was probably made keeping the individual testing of the security patches in mind. It was not a correct assumption as the compatibility of the new patch with the old system wasn't given a lot of thought. It was simply assumed that the patches work well with the existing system and probably end to end testing was not performed to verify and validate this particular assumption. Had proper analysis and system testing been done, it probably would have sniffed out the regression. This false assumption caused the system to regress, react, and perform badly in a situation when lives are at risk. Patient care was delayed due to this bug which is a huge risk and this shouldn't occur in critical applications.

Another thing to pick up from this incident is that the actual delivery process should not be neglected as well. Detailed thought must be given to the rollout process as well. The time and manner in which the delivery of any update is done is important too. Proper delivery times help minimize the risk involved. In this situation, the delivery of the security patches was done at a time that was not ideal. First of all, the security patches were installed all at once without any buffer. Secondly, the patches were installed during the day. Usually, hospitals are extremely busy during the day. The patches were installed at a time when several patients were being checked into the system which was not ideal. The rollout was done without thinking of the most appropriate time to carry out the delivery. In this instance, the hospitals should have been contacted and an ideal time should have been selected for the rollout of the patches. Ideally, a time when the hospital is least busy and overwhelmed should be selected so as to minimize any risk to the system. Since hospitals are critical and time sensitive, thought should have been given to the ideal time to deliver the patches which could have helped a lot. If the timings would have been ideal, the issue would have impacted fewer people and they probably would have had sufficient time to deal with the issue before more people got affected by it. There should possibly have been a ‘monitoring period’ when the impact and integration of the patches into the system were observed. This could have alerted the stakeholders that issues were present and needed to be addressed. Additionally, possible technical support for the rollout could have been arranged so as to help out if and when any issues occurred. The catastrophic bug affecting some of the major hospitals in Queensland could have been easily managed had the rollout been done in a more effective way.

4.9. CASE 9: UBER WAS SUED DUE TO A SOFTWARE BUG

This is a particularly interesting and exceptional case where a software bug resulted in the divorce of a couple. In a comical situation, a software bug in the popular ride sharing software application directly led to the divorce of a French man. An adulterous French businessman living in southern France, a resident of Côte d’Azur claimed that the application led to his divorce. The man, in this case, had once made use of his wife’s iPhone to book rides. But, despite having logged out of the application, his wife kept on receiving push message notifications on her phone. These push messages permitted his wife to monitor his rides and suspect that he was being unfaithful. The

bug in the system sent notifications and transportation details to his wife's phone despite the fact that he was no longer logged into his account on her phone. These push messages led his wife to discover his affair and eventually led to their divorce. Even after he had logged out, the wife kept receiving notification messages on her phone which should not have happened.

The man was exposed and his privacy was compromised. His wife had good reason to be suspicious as she received notifications of her husband using the application to hail a ride to visit his mistress. The angered man, who is now divorced decided to sue Uber as a result of this software bug. He claims that the bug caused problems in his personal life as his privacy was invaded.

He is suing the company for a sum of around 45 million dollars in damages. The man claimed that he was a victim of the software bug. The ethics of the situation can be left open to discussion, but what can be understood from this incident is that sometimes software bugs can prove costly if not tested properly and create unpleasant situations.

The details of the bug that was revealed by means of this interesting and intriguing case have not been made public. But BBC did its own test to reproduce the bug (The Guardian, 2017). They logged into an Uber account from one phone and then logged out of it. Then, they logged into the same account on another phone and ordered a ride from it. The application then sent notification messages to both phones about the trip (BBC News, 2017). This bug is claimed to have impacted only iPhones before a software patch had been added later the same year. No additional details about the bug were given to the public but one can assume it is related to the session authentication and security part of the application.

This incident reveals that a simple bug can create big problems; leading a wife to her husband's mistress resulting in divorce for the couple and a lawsuit for the creators of the application.

This incident raises questions regarding software security and user privacy and legal liabilities (The Guardian, 2017). What we learn from this is that security and privacy are important factors to be considered when building software applications that collect sensitive data of the users that use it. It has to be given a lot of thought when implementing such features into applications. Ride sharing applications collect a lot of sensitive information of the user such as their names, email addresses, phone numbers, credit card details and so on. This information should be well protected and secured by the software application.

What one takeaway from this particular incident is that software security is important and adequate measures are essential to maintain and ensure that the privacy of the users is maintained. As we saw from this particular case is that the privacy of a customer was compromised. The customer's private messages were sent to someone else despite the fact that the customer wasn't logged into his account. This was probably due to a caching issue where the login credentials were maintained on the phone despite the person being logged out. This highlights that proper authentication and session management must be done properly in software applications. In this case, the application did not check if the person was still logged in or not and kept sending messages. The application was probably not aware that the person had logged out due to the incorrect caching of the person's login session. The details are not known but we can learn that proper cleanup functions on logout must be performed each time a user logs out of his account.

The next thing to be extracted from this is that testing of real life scenarios is important when it comes to software applications. In addition to regular methods of testing, testing of real life scenarios must be done as well. The reaction of the system to real time events that can occur when people interact with the system in a way that is not usual should always be analyzed and investigated. The developer of the application failed to test the reaction of the system after someone has logged out it. Maybe an assumption was made that since the person has logged into his account on a particular phone, it is his personal phone and that it would probably never change. This assumption is highly unlikely as people change their phones all the time. But, nonetheless, this scenario was not properly tested by the testing team. The sequence of events that was followed by the customer in this case was not an unusual occurrence and hence the application should have been capable of dealing with the event in a reasonable and safe manner. It certainly should not have continued to send messages on the same phone without checking if the person was logged in first. Certainly, this situation should have been considered in the test plan during the testing of the application. Even though the result was a little comical and intriguing, we should remember that detailed testing should be done on all software applications.

Now that we have covered some cases that demonstrate the vital part testing plays in the software development cycle, we will now take a look at software testing and different methods of software testing in the next chapter.

Methods of Software and Systems Testing

CONTENTS

5.1. Objectives of Software Testing.....	118
5.2. Methods of Software Testing.....	124
5.3. Levels of Software Testing.....	142
5.4. Classification of Test Types	149
5.5. Types of Testing And Their Application To Test Levels.....	153
5.6. Factors To Be Considered When Choosing A Testing Technique.....	156
5.7. Steps To Perform Dynamic Testing.....	159

As covered in the previous chapter, the role of testing is quite crucial in the development of software systems. Testing goes a long way in helping build a software system or software product that is of a good quality. It helps in obtaining a level of confidence in the system by giving a level of quality to the system. In addition to this, the process of testing helps identify and fix defects in the software application during its development rather than after it has been put into production.

In this chapter, we shall look at the main objectives of software testing followed by concrete methods of software testing.

5.1. OBJECTIVES OF SOFTWARE TESTING

Software testing has certain key objectives that help achieve the development of a better software product. The organization ISTQB or International Software Testing Qualification Board is an organization that works at an international level and is a software testing certification board.

Some of the principal objectives of the field of software testing as defined by the ISTQB syllabus are as follows (Graham, Van Veenendaal, and Evans, 2008):

- **Prevention of defects by evaluation of different products such as the requirements, the design, the code and the user stories:** Any software that is developed goes through several phases such as requirements specifications, documentation, design, and eventually the development phase. Any items that form a part of the initial requirements such as the requirement specifications, the user stories, the design should be evaluated before they are presented to a developer for coding.

This evaluation helps in uncovering any ambiguities and inconsistencies in the requirements or needs before coding work is started. A walkthrough and inspection of code before it gets integrated into the system helps evaluate the code and maybe figure out issues in the requirements itself. Analysis done before such items are developed and help in creating a software product that is stable and saves time and effort. Evaluation of each of these items at each phase of the life cycle of software development, also called verification helps find ambiguities or issues or discrepancies in the requirements thereby saving a lot of time and effort.

- **Verification of the fulfillment of all the specified requirements:** Any software application begins with the requirements. The requirements are what define the trajectory of development and how the software is designed. The document that lists the needs of the client is a major component of the development life cycle. Hence, its verification has to be a part of the testing process. One objective of testing is to verify if the end result satisfies the entire requirement criterion that was put forth by the customer. It is essential and important that the needs and requirements of the client are met. The test team is responsible for testing the software product and making sure that all the requirements have been properly implemented. They are responsible for checking if the implementation is done in a correct manner and point out if there are some discrepancies or ambiguities or some defects in the system. This is a standard objective as it helps verify the functionality of the system and ensure that it conforms to the requirements given by the client. Development of test cases is a part of this objective. Test cases are made. Any type of testing technique involves the development of a test plan and test cases that make sure to verify the correct implementation of the functionality requested.
- **Building confidence in the level of quality of the software system or application:** A major and critical target of software testing is the constant attempt of providing and maintaining good quality. Hence, by consequence, software quality needs to be improved in order to achieve this objective. A software system that has a high level of quality means that it is well tested and would have fewer defects. If a software application is of high quality, it directly means that it has been well tested and that fewer errors will be found in the final product. High-Quality software directly translates to a lesser number of defects. The effectiveness of the test plan and process is what decides the level of quality of the software. A higher level of quality indicates a product of a better quality. This in turn helps gain the customer's confidence in the end product. A product that is tested efficiently is one of a good quality thereby ensuring the customer of its reliability resulting in the gaining of his/her confidence in the software system. Further, a high level of quality of the software system directly contributes to reduced or lowered maintenance costs and a significantly high level of customer satisfaction with the product.

- **Prevention of defects in the software application:** Another one of the key targets or aims of the process of software development is to elude any bugs in the system. The idea is to avoid the possibility of having serious defects by carrying out high quality testing in the initial phases of software development. The idea is to eliminate defects early on in the life cycle of the product so that in the later stages fewer defects are found. The detection of issues in the early stages of the development process reduces the project's costs significantly. Fixing of bugs after completion of the product is always more expensive. The anticipatory detection and identification of bugs, faults, or errors also reduces the effort involved by the development team. The team usually has to spend lesser amounts of time and effort on fixing a bug during the development phase rather than in the maintenance phase. To prevent bugs from occurring in the later stages, an analysis of the previously found bugs is done to determine the root cause of the defects and once this is completed, steps are taken to avoid such errors in the later stages of the development cycle. The occurrence of further errors in the system is prevented by this process of analysis in the early stages. It helps in looking for patterns and also maintenance of a repository of defects faced. Skillful testing helps prevention of bugs and develops an application that is less prone to errors or faults. Prevention of bugs directly and indirectly helps in reducing the number of defects in the system. This, in turn, will help develop a product of high quality, which, in turn will help gain the trust of the customer. Hence, prevention of bugs is an important objective of the process of software testing.
- **Finding defects in the system:** A key and vital target of the process of software testing is the identification of defects in a software product. This is important as detection of issues early on helps in the long run. If defects are detected beforehand, before the product is handed over to the client, the chances of discovering bugs later on are lesser. The key intention is to find as many defects as possible during the development phase itself combined with the key process of validation of the conformation of the system to the requirements. The uncovering of defects helps validation of the functionalities and enables the identification of any misunderstanding of the requirements, or any ambiguities or any possible unidentified issues in the requirements itself. This

is important as it is done during the development phase and such issues can be easily fixed during this phase rather than later. It is always much more costly to fix defects once the software is in use. Any potential defects in the system must be identified early on in the cycle so as to deliver a product of good quality.

- **Verification of the completion of test objectives and ensuring that the business and user's requirements have been met:** A prime objective of the test process is to ensure that all the objectives of the process are finished. This means that the product has been properly tested and that the needs of the user have been met. The goal is to validate the business requirements and verify if the product is working as it is supposed to be working. Ensuring that the expectations of the clients and all the stakeholders in regards to the functioning of the software application is a prime factor in the test process. The process of testing ensures that the requirements have been implemented correctly and that they function as desired by the users. This is the process of validation in software testing. This is done once the product has been coded or developed. Once development is done, testing techniques for this phase are implemented to validate the functioning of the system. Different means of validation are available to check this. Some techniques involve the customer beta testing some version of the product and providing their feedback. In essentiality, they use the system and verify if it meets their requirements and provide their views and give information regarding their overall level of satisfaction with the product. It can also be done in the form of a demonstration to the customer. As the customer is the end user and the person who will be using the product regularly, his opinion matters a lot. There is a prominent necessity to satisfy the requirements of the customer in any business which is done in case of software with the help of the software testing process.
- **Providing stakeholders with sufficient information so that they can take informed decisions mostly in regards to the level of quality of the software system:** Testing is a way of providing the stakeholders of the system with information regarding the system. Statistics, level of quality and validation of requirements is done via testing. The stakeholders can be given a lot of information pertaining to the software application such as certain risks involved, technical or functional restrictions,

ambiguities, defects, coverage reports and so on. Several different forms of reports can be given to the customer so that they can understand the system in a better way. They can be given details of the tests, the coverage of tests, the defects found, the issues with the requirements if any, etc. Testing can also reveal if some requirements are missing and help them understand the impact of missing or ambiguous requirements. Such information can bring clarity to the stakeholders and help them determine the risks associated with the software product as a whole. They can then make decisions regarding the level of quality required, the risks, and other things based on this information. The prime intention is to achieve a level of transparency in the communications with the stakeholders and give them information so they can understand different factors that affect software quality.

- **Ensuring that the level of risk of low or insufficient quality of the software is low:** Every software system possesses some amount of risk. No software is perfect, and it is not possible to develop a software system devoid of bugs. There is, therefore, some amount of risk always associated with software. The goal of software testing as a process is to reduce the possibility of high risk. Any probability of loss in terms of effort, product function or use and time is a risk. Each and every software system or application that is developed can incur risk from different sources. Some major sources of risk are uncertainties related to the technology, the budget constraints, design, implementation uncertainties, product maintenance and so on. The risks involved in these phases need to be minimized during the development phase of the product. If the uncertainties related to the project that could potentially impact the project are not managed beforehand, there could be issues after the completion of the software product. Improper handling of such aspects that bring out uncertainties could give way to more risks in the development of the software which is not the goal. If unhandled, such unpredictable issues could pose risks even after the product has been deployed. The entire life cycle is affected by this and hence the potential risk must be mitigated during the phase of software testing. Hence, identification and mitigation of risks is a key aim of the testing process. The intention is to uncover any risks early on and manage them so that no issues are seen in the future.

- **Ensuring the compliance of the contractual, legal or regulatory standards or requirements, and also verification of the compliance of the product with these standards:** Whenever software is developed, it is supposed to be deployed in a specific country or a number of countries. Each country could and usually does have its own rules and regulations pertaining to software. Software developed for specific regions is subject to the rules and regulations of that region and hence must follow them and conform to them. Legal rules must be followed in the implementation of the product. For instance, sometimes the law in some countries regarding the use of personal information differs and when software is implemented in such countries, the rules of that country must be followed and the data must be protected by the software. Failure of compliance with the rules of such countries would directly result in a compliance failure and thereby result in failure of the product itself. Such situations should be avoided and hence testing is done. In most cases, software needs to meet certain standards of testing as well. Different national and international standards of testing have been introduced over the years and software is bound by these standards. Part of testing involves checking and verification that the software conforms to the regional and international standards of software development and testing.

In addition to the objectives of software testing, it is important to know that the entire process of testing aims at finalizing the software product or the application as per the requirements of the business and the user. Software testing does this by means of tests and good test coverage.

The concept of good test coverage will be discussed in the upcoming sections. But it should be noted that a good coverage helps getting assured of the well working or functioning of the software and verifying that the software meets its requirement specifications. Different software projects have different individual needs of testing that are met in the software testing process.

The goal is to finalize the end product and ensure that it satisfies the requirements of the business and the users.

Now that we have seen the objectives of software testing, we shall take a look at different techniques or methods of software testing.

5.2. METHODS OF SOFTWARE TESTING

There exist different types of software testing methods that can be applicable to software projects. Broadly, testing techniques are classified into two different categories namely dynamic testing and static testing.

Dynamic testing is when the software is in use or operation. In this mode, the software system is executed and different types of testing are performed on the system. It is always performed in an environment that is executing the system. A runtime environment is needed for this type of testing. The code in the software is executed by giving it an input and the output provided by the system is compared to the expected value of the output. This is done to observe the behavior of the system and ensure it works correctly. Different behaviors of the software system are observed such as the functional behavior, the performance, memory management, the response time and so on. The system is also monitored in this type of testing. This testing aims to validate the system and its behavior. It serves to validate and evaluate the final version of the software product. It can also be called validation testing. Different categories and classifications of dynamic testing will be discussed in the upcoming sections.

5.2.1. Static Testing

Static Testing is a type of testing or testing technique where the code remains static and is not in use. The code is not in operation in this case. This process is done offline, in other words. This process can be done either in a manual way or by use of a set of tools meant for testing the code in a static way. It is a way of testing the code before it has been executed. Static testing is meant to check the code and the documentation. It aims to check the code, the documents pertaining to the requirements, the design level documents, etc. Review of all such documents is done in order to look for possible issues in the code, the requirements, or the design. This is done so as to find issues before the code is in operation. The software is analyzed for issues or flaws in an environment that is not runtime. During the period of the software when it is inactive, security analysis can also be done so as to check for possible points of weakness. In the non-operational condition of the code, it is looked at with the intention of sniffing out errors by observation and analysis. It aims to verify the system before the code is deployed.

This avenue or type of testing serves to identify possible bugs, errors, flaws in code, security issues or malicious code in the software application. This is done in the early stages of the software development process. It is

also called verification testing as it ensures that the code and documentation conform to the original requirements of the system. This testing is done on the project documents such as the requirement specifications, the design documents, the source code, the test plans, the test cases, the test scripts, the content of the web pages, etc.

Static testing is done in several different ways. Analysis and review of the software is done in a static or offline mode. Some of the ways of static testing are as follows:

1. **Informal Review:** This is generally the first step in the process of static testing. This is not a formal process and either the code or documentation is reviewed by a couple of people. This is always done in the earlier stages of the testing process. The objective is to look at the code or a document before discussion regarding the same in a more formal setting such as a meeting. The target is to look for particular deviations from the norms and to identify possible ambiguities in the requirements. The number of people that can participate in this process is not limited to two people. There can be many people that take part in the process of an informal review.

During the development of any software, a developer always checks for errors in his code. Similarly, one must make an assumption that there will be some errors in the product as well. No work is without error and it is safe to assume that there would be faults or errors in the system as well. In software, often, there are mistakes made by the programmers for a variety of reasons such as misunderstanding of the need, unclear requirements, blind spots and so on. A developer must expect errors in his/her code. Hence, if all developers move forward with such an assumption, they naturally must check their own work. Sometimes, unclear or ambiguous requirements lead to errors in the code. Some mistakes are a result of a simple misunderstanding of the needs to begin with. Sometimes, bad assumptions are made and the developer may end up making the same assumption during testing his/her own work thereby making the test redundant. It is possible that the person who developed or wrote the code will be unable to discern faults in the code he produced. Ideally, the person who developed the code should not be the one who is responsible for finding flaws in the code. This job should be entrusted to another person who isn't aware of the code as he is more likely to spot errors or faults in the system. Hence, informal reviews take place so that the view of an impartial authority can be gained.

Generally, the author of the code or documentation presents it to another colleague. They discuss the topic at hand and comments are provided by the colleague. It is basically a top down analysis and review of the content at hand. The document or the code is analyzed in an informal setting and feedback is collected. This process can be repeated many times. The end result is to get constructive feedback from the process and to improve the quality of the content developed by the author. Sometimes, notes can be taken. But, in most cases, no formal documentation is done during or after an informal review. In this type of static technique, the reviewer is not prepared in advance regarding the content of the review. This way of static testing is a way to get feedback in a non expensive way. Pair programming can be done or a reviewer such as the lead can be the person who analyses the content such as a document or the code. This method of static testing is a simple yet effective way to detect any anomalies in the software application without spending a lot of time and effort.

The final target is to detect bugs or faults as early as possible.

2. **Walkthrough:** This process is another informal technique of static testing. This is usually a guided process. The author of the code or the document under review usually presents the items to the reviewing team. He explains the code or the document to the team and ensures that everyone understands the subject at hand. The intent of this method of static testing is to achieve a common understanding of the topic at hand. Collective feedback is obtained from the team regarding the code or documents and notes can be taken by the author. Although a walkthrough is an informal technique of static testing, a thorough review of the code or documents is conducted by the group before it is passed on to the next stages of static testing.

This process can be executed in the form of a meeting that is always led by the author of the code or the documents. The author can present the code or the documents in different ways. The product can be presented to the audience. For instance, the author can present a particular code to the team and explain its working. It would be similar to a dry run where the possible behavior is analyzed by everyone. The team can walk through several scenarios and anticipate how the code would react under different conditions. The team can then share their views on the subject and give feedback in the form of comments and observations. The team can pose questions regarding the product and enhance their understanding of the code. The end goal is to learn about the working of the software application from

the meeting and gain insight into the code or the document. The objective is to gain proper understanding of the content and identify bugs in the early stages of static testing. This process can be documented. At times, notes are taken and any bugs identified are documented.

This process of static testing is useful in case of documents of high level. Documents such as the requirement specifications and design documents can be effectively reviewed by an informal walkthrough. In this process, the team is not prepared regarding the code, product, or the documents beforehand. The objective is to describe the items to the team and evaluate the quality of the content or subject at hand. Information about the topic at hand is collected and studied to achieve a common level of understanding which makes it easier to detect any potential anomalies in the system. Alternative solutions are sometimes discussed and reviewed and comparisons are performed. The value and quality of each solution is discussed and comments are provided. The underlying goal is to identify issues before presenting the product to a formal reviewer or moderator.

3. **Technical or Peer Review:** This is a method of static testing that is a discussion focusing on the technical content of the software application. It is another informal review process but slightly more formal than the previous methods of static testing. Ideally, this process is guided by a trained individual. A moderator who is trained in methods of testing is responsible for guiding the process and the review itself. This process can be performed among peers without the presence of managers in the discussion. This is a slightly more formal process as it is a documented process. This is a round of tests that reviews the technical part of the software code and its conformity is checked. The technical details of the system are analyzed and reviewed and their validity is checked offline. The moderator checks if the system complies with the software standards and that the technical specifications of the application have been met. The moderator, in ideal cases, has greater knowledge of testing and review and is qualified to spot bugs, errors or inconsistencies in the product at hand.

The moderator is tasked with bug detection, checking for compliance with standards and evaluation of the code. In this case, the moderator and the team both are prepared beforehand regarding the product or application under consideration. Discussions in a meeting are conducted regarding the code and the documents. Peers and experts in the technical domain participate in this meeting. This method may employ the use of reports,

documents, and checklists for efficiency. The purposes of meetings carried out with the intent of a technical review are usually multifold including discussions regarding the product, evaluation of the quality of the product, defect detection, evaluation of alternative approaches, fixing of technical issues and conformity checks.

As the moderator is an experienced individual, he is capable of helping the team in fixing any bugs that they uncover. The reviewer provides alternative technical approaches in case issues are raised regarding the implemented approaches. The goal is to identify issues, fix the issues, and provide alternatives approaches. If the code does not conform to the prescribed standards, the moderator provides alternative technical solutions that conform to the standards. In this method, different items of the application can be tested. Test plans, the code, test cases, the testing strategy, the test scripts can be reviewed and discussed in this stage of static testing.

The meeting is usually a precursor to the formal process of static testing. The plan is usually to find any missing requirements, look for code that cannot be maintained long term, identify bugs, and find design issues.

4. **Inspection:** This is the most formal method of static testing. An inspection is generally lead and guided by one or more trained reviewers or moderators. The people attending this review meeting are prepared in advance. They have knowledge of the system and previous feedback documentation, if any, is known to the moderators. They prepare for the meeting beforehand by reading and studying the existing documentation. The primary purpose of an inspection in static testing is to uncover bugs or defects. The product is examined and a formal walkthrough of the code is conducted in an inspection. Peers are present and they help examine and review the product and the documents as well. Defects are identified and logged and documented in a formal setting. The moderators are highly skilled and proficient in the knowledge pertaining to the product and its requirements. They spot defects and provide feedback and suggestions regarding how the defects can be corrected.

Once done, a follow up that is also formal is done again. Entry and exit criteria and conditions are prepared in advance for this type of testing. This process includes a detailed checklist that is followed. The checklist defines the conditions that need to be met by the documents and the code and is used by the moderators to perform their analysis. The inspection method

is extremely helpful to all the parties involved as it helps solve issues in a collective manner. The author or authors get constructive feedback on their content and they can now improve the quality of the content they create. The documents and code are reviewed and it helps maintain the quality of the application. Defects are identified and tracked so that such bugs can be prevented in the future stages. If such bugs or defects occur again, there is a written record of it and the product can be analyzed further to determine the root cause of such errors which can be the requirements, the implementation or the design. Information regarding the product is obtained and a common collective comprehension of the system is generated.

This process can involve the use of metrics and other forms of measurement. In addition to the bug reports, reports on the inspection are maintained as well. Formal documentation for each inspection is maintained. All efforts of the inspection are directed to improving the quality of the software product.

5.2.2. Dynamic Testing

As we have discussed previously, dynamic testing is a testing technique that checks the behavior of the software application in an environment where it is being executed. The dynamic or runtime behavior of the code or the application is tested in this case. The working of the software application under different conditions is tested. The goal/purpose of this type of testing is to make sure that the software application works in a correct manner after it has been deployed or installed. The intent is to deliver a stable application to the client. Testing of this type or category intends to identify and fix any major issues in the software before it is handed over to the users (Fairley, 1978). The idea is to uncover as many defects as possible before the product or rather a version of the product is released. The goal is to detect and fix bugs as detection of bugs is the way to proving that the software quality has been maintained. Testing cannot prove that the software is bug or fault free, its intent is to demonstrate the presence of bugs which can serve as a measure of product quality. Dynamic testing of the application serves to assure that the software behaves in a consistent manner under normal and abnormal conditions.

Dynamic testing is further divided in two categories. They are as follows:

- 1 White box testing; and
2. Black box testing.

We shall now discuss each of these categories in detail.

5.2.2.1. White Box Testing

White box testing is a type of dynamic testing technique where the internal design, structure, and code of the software solution are known (Graham, Van Veenendaal, and Evans, 2008). This category of testing permits the tester to know the code of the software system. In simpler terms, white box testing focuses on the internal working of the solution where the internal components are known to the person performing the test. Other synonyms of White Box testing are Clear Box Testing, Transparent Box Testing, Structural Testing, Glass Box Testing, Code Based Testing, etc. This type of testing concentrates primarily on the architecture, the design, the internal structure, or the code of the software application.

There are three main techniques of white box testing. They are as follows:

1. **Statement Coverage:** This technique involves testing of each line of code in the software system. In software programming or coding, a statement is simply a line or piece of code. It is an instruction given to the computer to execute. When the code is compiled, the statement has the status of being 'executable.' Statement Coverage testing is a method of testing that demands each line of code be executed at least once by means of different test cases.

Statement coverage is calculated as the count of the number of statements executed by the test divided by the total number of executable statements, in the form of a percentage (Spillner et al., 2007).

For instance, consider the following code:

```
public void isNegative (int input) {  
    if(input > 0)  
    {  
        System.out.println("The input is positive");  
    }  
    else  
    {  
        System.out.println("The input is negative");  
    }  
}
```

In the above code, if each and every line is executed by a set of tests, then all the statements have been covered.

Hence, different inputs are provided in a test set that cover all the statements in the code.

If we consider a test case with an input, say 5, for instance.

In this case, only the highlighted portion of the code will be executed as '5' is a positive integer.

```
public void isNegative (int input) {  
    if(input > 0)  
    {  
        System.out.println("The input is positive");  
    }  
    else  
    {  
        System.out.println("The input is negative");  
    }  
}
```

Let us calculate the statement coverage in this case.

The total number of executed statements is 6. The total number of statements is 10.

This means that the statement coverage is $6/10 * 100$ which is 60%.

Let us consider another test case. If we provide an input value of -5 to the system, the message "The input is negative" is printed. Most of the code is executed which is shown below:

```
public void isNegative (int input) {  
    if(input > 0)  
    {  
        System.out.println("The input is positive");  
    }  
    else  
    {  
        System.out.println("The input is negative");  
    }  
}
```

Let us calculate the statement coverage in this case.

The total number of executed statements is 9. The total number of statements is 10.

This means that the statement coverage is $9/10 * 100$ which is 90%.

2. **Branch Coverage:** This subcategory of white box testing aims to cover all the decision statements in the code. The true/

false values of the code are evaluated and tested. Basically, any code that provides a choice between two values and the code it executes when one of the values is chosen is tested. This path of testing intends to execute all the decisions in the code. Hence, this category is also called decision coverage testing.

This testing technique focuses on all the control statements in the code. Any flows from a point of decision (an IF statement for instance) are tested in this process. All the possible flows of the code are required to be tested. Sometimes, when the code is a little complex, the expressions in the code become a little difficult to cover. Different metrics are used to measure decision coverage.

The most common formula used is:

Number of decisions tested/Total number of decisions \times 100.

We shall now perform decision coverage calculation on a simple piece of code. The code is as follows:

```
public void isNegative (int input) {  
    if(input > 0)  
    {  
        System.out.println("The input is positive");  
    }  
    else  
    {  
        System.out.println("The input is negative");  
    }  
}
```

In this case, the decision point is the IF statement. Two decisions need to be tested: when the value is greater than 0 and when it is less than 0.

To test this, let us consider a simple test case. If we provide the number 6 as an input to the above function, we get the statement “The input is positive” as an output.

Let us calculate the branch coverage for this input.

The number of decision outcomes that were tested is 1. The total number of decision outcomes is 2; one for the input value greater than 0 and one for a value less than 0.

So, the decision coverage percentage is $\frac{1}{2} * 100$ which is equal to 50%. Now, let us consider a test case which meets the second decision.

In this case, we provide the input value as -5.

The output we get is “The input is negative.” Again, in this test case, the number of decisions that was executed is 1. Hence, the decision coverage is again 50% in this case.

It should be noted that it is not possible to get 100% decision coverage with just one test case (Spillner et al., 2007).

As demonstrated by the two test cases, it can be seen that each test case accounts for only 50% of the decision coverage. To obtain a complete 100%, we need two test cases.

- **Test case 1:** Input = 6.
- **Test case 2:** Input = -5.

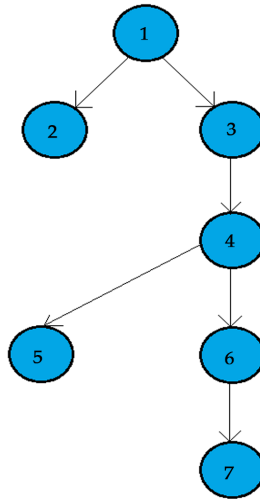
Branch or decision coverage offers a lot of advantages in testing. It helps validate the working and behavior of each decision taken in the code. It helps ensure that no branch in the code exhibits abnormal or disruptive behavior. Branch coverage also helps test parts of the code that would not have been tested by other methods. It helps overcome any omissions that could have occurred during the statement coverage testing. This is because statement coverage does not guarantee that all the decision points have been covered.

It should be noted that if 100% decision coverage is achieved, 100% statement coverage is also achieved. But the reverse is not true. 100% statement coverage cannot guarantee 100% decision coverage. As seen from the example presented for statement coverage, the second test case ensures that all the statements have been covered but not all the decisions have been considered by this particular test case.

3. **Path or Condition Coverage:** This technique of white box testing is a method where each condition in the code along with each potential path is tested. Each condition in the code has several possible paths associated with it and this method of testing aims to test each and every one of the paths. This type of coverage usually exploits the relationship between the variables and paths in the code. The way in which each variable or expression is evaluated is tested using this technique. Expressions with different logical operands are usually tested in this coverage. For instance, expressions like AND, XOR, OR provide different possibilities and paths that are tested in this coverage.

This coverage helps identify linear paths throughout the code. Each path is then tested via a test case. The test team comes up with something like a control flow analysis of the code and each path that is visible is tested.

For instance, consider the following flow chart:



Now, the testing team will analyze this chart and design test cases for each of the possible paths that are present.

The following possible paths are present in the above chart:

- 12;
- 13,467; and
- 1345.

Test cases that cover each of the above paths are written and the paths are tested. The intent is to find paths that are inefficient or redundant or broken. Errors and issues in the paths are identified in this method of testing.

In simple worlds, in the code, each of the expressions present in the code is evaluated to either true or false and each of the corresponding paths for the true/false value is tested.

Let us see this with a simple example. Consider the following code:

```
public void testCondition(boolean a, boolean b, boolean c)
{
    if( (a && b) || c)
        System.out.println("Condition is true");
    else
        System.out.println("Condition is false");
}
```

In order to cover each of the operands in the Boolean expression, each one of the true/false values needs to be evaluated.

Hence, the following values will be tested to ensure complete condition coverage:

- $a = \text{true}, b = \text{true}, c = \text{true}$ (c is not evaluated as first expression is true);
- $a = \text{false}, b = \text{true}, c = \text{false}$;
- $a = \text{true}, b = \text{false}, c = \text{false}$;
- $a = \text{false}, b = \text{false}, c = \text{true}$;
- $a = \text{false}, b = \text{false}, c = \text{false}$;
- $a = \text{true}, b = \text{true}, c = \text{false}$.

As 3 variables are present, 6 combinations are possible:

- TTT;
- TTF;
- TFF;
- FTT;
- FFT;
- FFF.

As we can see, each of the operands or variables has been evaluated to true/false at least once and each of the corresponding paths is evaluated.

Condition coverage can be calculated as a metric as well.

The formula is as follows:

Condition coverage = $(\text{Number of executed paths} / \text{Total number of possible paths}) \times 100$

This coverage helps identify different decision points and paths in the code and helps uncover any issues in the code. Gaps in the requirements can be exposed as a result of this method of testing.

5.2.2.2. Black Box Testing

Black box testing is a type of dynamic testing where the internal structure of the code and/or its design is not known to the tester (Spillner et al., 2007). The intent of this type of testing is to verify that the software system is functioning correctly. This testing focuses primarily on the functionality of the software application as a whole. This testing is done from the perspective of an end user who has no information regarding the internal workings of the

system under consideration. This category of testing takes into consideration the behavior of the system and ensures that it conforms to its specifications.

Black box testing has different techniques in which it is implemented. Some of the common Black Box testing techniques are:

1. **Equivalence Partitioning:** This technique is also called equivalence class partitioning where the data is divided into classes or partitions. The members of each class or partition are supposed to be treated in the same way by the software program. Partitions for different sets of values are created. There are partitions or classes made for both valid and invalid values. Valid values are those that are accepted by the system or a component of the system. Invalid values are those that the system is programmed to reject. If needed, a partition is further divided in partitions based on conditions.

The target or aim of applying this particular technique is to reduce the amount of test cases that need to be written and executed. Instead of testing the software application against each and every value in the partition class, only one value belonging to each partition can be tested as the system will behave in a similar way for all the values in that particular partition.

This way, a lot of time and effort can be reduced and more coverage of the system can be achieved.

For instance, consider a software application that accepts the age of a person as an input. Let us consider a simple application for recruitment that recruits people that are aged between 18 and 64 years.

Hence, all input values between 18 and 64 are valid values. All values equal to or less than 17 are invalid. All values above 64 are invalid as well.

So, we have three equivalence classes as follows:

Equivalence Class Partition	Valid/Invalid Values
18–64	Valid
≤ 17	Invalid
> 64	Invalid

Therefore, for the purpose of a test case, one test case each using a value from each of the partitions is tested. This way, both valid and invalid inputs belonging to different partitions are tested. This reduces time and effort as each and every possible value for the age of a person doesn't need to be tested. Three test cases are sufficient for testing if the application is working

properly in terms of its acceptance of the correct age criteria.

Similar classes can be made for other input data across the entire application so as to reduce the effort and time and improve the rate and quality of testing.

- 2. Boundary Value Analysis:** This method is similar to the previous equivalence partitioning method but makes use of only the minimum and maximum values in each class or partition. As the name suggests, the minimum and maximum values or the boundary values are tested using this technique. The behavior of the software system at the boundaries is observed and tested. This region is tested as there are high chances of experiencing abnormal behavior at the boundaries rather than within the defined boundaries.

This testing is an ingenious way to check if the software system respects the boundaries and accounts for inputs that do not fall under the defined scope. This method helps determine potential bugs in the system caused by invalid inputs that are given to the system. Any requirements in the software application that need a fixed range of values can be tested using this type of testing.

For instance, let us consider the same example considering the age group of potential recruits. As we know, the recruitment process requires candidates that are aged between 18 and 64 years. Boundary value analysis will check the boundaries of each of the equivalence classes.

We have three classes with the following range: ≤ 17 , 18–64 and >64

In this case, the boundary values that would be potentially tested are 17, 18, 19, 63, 64, and 65.

One value after and before each boundary is tested (18+1, 18–1, 64+1, 64–1) and the actual boundary values are tested as well (18, 64). This type of testing helps uncover issues at the boundaries that could have been overlooked during the development process.

- 3. Decision Table Testing:** It is similar to the decision testing technique used in White Box testing. This testing mechanism serves to test logical decisions and relationships present in the code and evaluate the possible values (true, false) for each point of decision. A table is prepared in this case that holds all the rules implemented in the code. A decision table usually consists of inputs (conditions) and the supposed outputs for each of the

input conditions (Spillner et al., 2007). Generally, the table rows consist of the conditions at the top and the corresponding actions at the bottom. Each column in the table is a decision point that is a unique mix of conditions which culminates in the execution of the statements related to that particular rule. These conditions or decision points are generally depicted as either true or false. Discrete values can be used as well.

A decision table is defined as follows:

Decision Table		
Conditions	Rule 1	Rule 2
Condition-1	T	T
Condition-2	T	F
Actions		
Action-1		Action 1
Action-2	Action 2	

As we can see, the conditions are expressed in Boolean terms and the actions that correspond to a particular selection or set of conditions is defined.

What happens in this technique of testing is that the test team identifies a set of conditions that evaluate to a value set (true/false for example) and actions for each value are identified. On the basis of the potential scenarios that could arise based on the conditions, a decision table is created so as to help with the process of testing for developing test cases.

Let us see an example of a decision table.

Consider a software application that provides loans to people. Based on certain conditions, some clients are given lower interest rates. For example, let us say that people between the ages of 20 and 40 are given an interest rate of 8%. Further, if this condition is satisfied and if the person also is a doctor, he/she is eligible for a much lower interest rate of 6%. If the person's age is not between 20 and 40, then the interest rate is 10%. Also, we assume that any age below 20 is not allowed as an input by this system.

A decision table can be made to assess the number of possibilities that are possible for these conditions. This helps in deciding how many test cases need to be written in order to test these conditions properly.

A decision table that considers the two conditions in our example is shown below:

Decision Table				
Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Age is between 20 and 40	T	T	F	F
Is a doctor	F	T	F	T
Actions				
Interest rate is 8%	O			
Interest rate is 6%		O		
Interest rate is 10%			O	O

As we can see, each condition has two possible values-true or false. For a combination of each of the condition's values, different possible outcomes are present. As we can see, a total of four possible actions are present. Based on the table, different test cases can be written. Test cases can be drawn up to cover all the four scenarios which makes it easier to test the application. Similar tables can be made for other conditions that form part of the application and these tables can be used to drive the test case plan.

4. **State Transitioning Testing:** Each software system reacts in different ways in certain conditions. Different events trigger different reactions from the system. Based on certain conditions, the system changes its status. Events cause the system to move from state to state. As the system moves ahead in its functioning, several events occur the minute the application starts and is initialized. Each event changes the state the application is in. The history of the software system can be traced using the varied states it passes through in its execution.

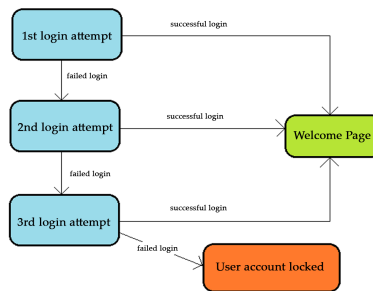
In this test technique, the various states of the system are tested and the behavior is analyzed. A simple state transition diagram for the software system is used here and it shows the different states of the system. Different states such as the software entry (initialization), transition, and exit are described by this diagram. Usually, the system moves from one state to another when it experiences an event (an input is given to the system, for example). When such an event occurs, the system then moves or transitions into another state. There is a possibility that a single event can cause the system to transition in more than one way from the same initial state. For instance, if, for a particular input, two tasks are needed to be completed, then two transitions are triggered from the same event.

Further, a change in the state may lead to an action performed by the software system. The goal that is to be achieved from a state transitioning

diagram is to show all the possible valid transitions and states a software system can be in at a given point of time. This diagram depicts all the valid states and transitions and the resulting actions, if any and also all the invalid transactions that could occur between the different states in the application. Such a diagram also shows the events that cause the system to transition and move from one state to another.

We shall now cover an example of a state transition test. Let us consider a simple and uncomplicated login page. Let us assume that we have an application that requires a username and password to sign in. The assumption is that the maximum number of incorrect tries is limited to three. A state transition diagram will help understand how events affect the system and how it transitions to another state.

The diagram below represents the different states and the transition between them for our login example:



As we can see from the above diagram, for the event of login there are several states that the software could be in. On a successful login, the system goes into the welcome state. On each failed login attempt, the next login is attempted and each of these attempts is a state of its own.

After three unsuccessful/failed tries, the account is locked and this is another state. The event that triggers this reaction is the action of login. This event is capable of driving the system into different states. Based on this diagram, test cases can be drawn up and designed so as to cover each of the possible scenarios of login. This diagram can also help in identifying potential invalid transitions and provide insight into any potential gaps in the requirements.

This diagram helps develop tests that execute a particular sequence of states, a particular series of events, execute all the states in the system, and also test invalid transitions between states.

5. **Error Guessing:** This is a type of a testing based on experience. It is an experience based technique of testing that attempts to anticipate possible errors in the system based on previous knowledge gained by the tester. A tester who has previously tested the same application has knowledge regarding the working of the system. He/she knows the kind of errors that are usually encountered and hence can anticipate failures of the same kind. Additionally, a seasoned tester who has worked on testing several applications has amassed experience in terms of defects and can methodically guess the type of errors a software system can encounter.

Defects that are usually found in most applications can easily be guessed by experienced testers. Generally, in this method of testing, the test team strategically prepares a set of test cases that expose a fixed list of guessable errors based on past experiences. A list of potential probable defects based on previous knowledge, defect data, and even common knowledge of defects in software systems is prepared and test cases are prepared based on this list.

For instance, some common errors in development are:

- Null pointer exceptions (null values in fields);
- Divide by zero errors;
- Invalid input errors;
- Incompatibility errors;
- Submitting a form without providing any input;
- Uploading to a page without any attachments;
- Failure to comply with the size limit on uploads.

Let us consider a simple example. Let us assume that we have a simple application that accepts the age and name of a potential candidate. The name and age are provided as inputs to the software system. Now, in this case, a seasoned tester would make a guess suggesting that sometimes, software system do not employ checks for the inputs.

A possible list of errors to test that can be drawn up for this scenario is as follows:

- Invalid input for age (character, special signs, floating point values);
- Invalid age (negative numbers, large or out of bound numbers);
- Invalid input for name (numbers, special characters, etc.);

- No input given for age;
- No input given for name;
- Both fields are empty;
- Null values entered.

As seen from this example, a list can be prepared and a set of tests can be drawn up to test these scenarios. Such lists can be made for each of the functionalities of the system and the corresponding tests are created for each of these scenarios. This method of testing helps eliminate the obvious and known errors that can be faced by a software tester and thereby improve the quality of testing.

Now that we have covered different techniques of white and black box testing, we take a look at the types of white of and black box testing. In the next section, we take a look at the different levels of testing and how different techniques of white box and black box technique are applied to different levels.

5.3. LEVELS OF SOFTWARE TESTING

In software testing, different levels have been defined based on the amount of detail and their location in the software development life cycle (Spillner et al., 2007). The tests are organized and managed together at different stages in the lifecycle of the software and hence we have levels based on certain criteria.

In software testing, there are four defined levels of software testing (Richardson and Wolf, 1996). They are:

- Unit/component testing;
- Integration testing;
- System testing; and
- Acceptance testing.

The aim of defining different levels in testing is to ease the process of testing and make the identification of test cases simpler at all levels in the software development life cycle. Each of the levels has its own target and purpose which adds value to the testing process. Let us take a look at these levels in brief.

1. Unit/Component Testing: A unit or a component is a small part of the software system that can be tested. A module/unit/component is a portion of the software application that can be tested individually (Richardson and

Wolf, 1996). This testing can cover different functional and non functional behaviors of the system under test. Structural aspects such as decisions and branches can be tested in this case. This testing usually requires access to the code of the software. This type of testing is usually done on isolated sections of the software system and is always done by the development team.

In this level of testing, usually components, units, modules, data structures in the code, classes, or database entities can be tested individually. Generally, development tools are needed for this type of testing. Several mocking tools are available that help developers write tests for testing a unit of the software system. The intention behind unit testing is to reduce the defects, preventing the defects from moving up to higher levels in the life cycle, to improve software quality and to verify the functional and non-functional aspects of the software system.

The goal of module testing is to test different modules separately and this testing is in most of the cases, performed by the developers itself. This is because intimate knowledge of the code is needed to perform this level of testing. Also, access to the code is needed to perform unit testing which is why developers usually write unit test cases. In most cases, the developer who develops the unit or module writes the unit tests for it.

2. Integration Testing: It is a level of testing that focuses on testing the interactions of combined groups of units or modules of the software application. The interaction between different units of the software system is tested in this case and the smooth functioning of the combination of the individually tested modules is checked in this process (Richardson and Wolf, 1996). The level of software testing aims to ensure that the combination of different modules has not broken down the system or has not added new defects to the system. The components have graduated from the unit/isolated level to the integrated or combined level and this composition of units is tested at this level to ensure that the quality of the software is maintained and risk is reduced. The intent is to build a substantial amount of confidence in the integrated system and help prove the changes and merges have not compromised the overall working and quality of the system.

Different parts of the software can be tested at this level. The sequences in the system, the design, and architecture of the system, the interfaces of the system, etc., can be evaluated and tested at this level. Different objects belonging to the software system such as the databases, infrastructure, workflows, API's, services, subsystems, etc., can be tested via integration testing (Richardson and Wolf, 1996).

Usually this level of testing is performed by the test team. If the components that are to be tested are small and cannot be delivered or isolated as a whole, the testing is done by the development team. The person who performs the integration testing is usually chosen based on the size and nature of the integrated component. If the integrated components are at the technical level, the development team tests it. Else, the testing team tests the system. In most cases, the flow of data between different components or subsystems of the software system is tested and this is performed by the test team.

Usually, integrated testing is done in an iterative life cycle as it helps minimize the risk of regression. This test focuses on the integration between the several components and not the individual functioning of the components itself. It aims to pinpoint any faults that may exist in the interaction between distinct components or subsystems. An incremental way of integration is generally recommended so as to simplify the process and to isolate defects rapidly and efficiently. This approach is considered because as the scope increases, the harder it is to isolate errors or bugs in the system. The harder it is to isolate and identify bugs, the higher is the risk of loss in quality. Hence, in software, an incremental approach where each component is tested individually and then integrated one by one is implemented so as to increase the efficiency of the testing process and to reduce risk.

3. System Testing: It is a crucial and important level of software testing. As the name suggests, it is a level of testing that aims to validate the software system as a whole. The complete and entirely integrated software system as a product is evaluated at this level. The intent of this testing is to validate the system from end to end (Richardson and Wolf, 1996). The conformity of the system to its requirement specifications is checked in this level of testing. The system is evaluated and tested to assess its level of conformity to its predetermined and preset requirements. The overall interaction between the different components of the software system is tested at this stage. This testing concentrates on the software system as a whole and targets the end to end working of the system and considers the non functional behavior of the system while execution.

Generally, software is just a small part of a large system that is computer based. In the end, software usually interacts with one or more hardware elements. System Testing chooses to focus on the bigger picture where the entire software and hardware interactions are tested. Generally, this testing aspires to test the complete software system and its functional as well as non-functional behavior. This level of testing is always done by the test team.

System testing does not require access or knowledge of the internal code of the software system. The system is tested for its behavior and how the integrated subsystems and peripherals interact with one another as a whole. The complete chain of events that can occur starting from the initial step until the last step is tested at this stage. The functionality and interaction within the system is checked and defects are uncovered. This is also called end to end testing. This test level aims to test the input output scenarios of the system. The system is evaluated and tested to verify that it gives the correct outputs. The system is inspected by means of tests in order to evaluate all the possible use cases that could be executed or performed by the user of the system. The needs that were specified by the client are verified by means of system testing.

The system level testing address different objects in the software such the application itself, subsystems, hardware and software integrations, operating systems, the system data, system configuration, etc. The goal is to uncover incorrect or unexpected behaviors, wrong data outputs, incorrect flow or interaction between components, failures in the end-end testing, failures in the functioning of the system in specific environments, behavior that is not as specified in the manuals, etc. The purpose of this testing is to reduce defects, reduce the risk, ensure compliance to the requirements, validation of the entire system as a whole, prevention of defects, verification of the functional and non-functional behaviors of the system and improving the quality of the system.

In all cases, the test team performs system testing. Detailed test cases are developed and test suites are made that consider each and every aspect of the software system as a whole. The test team looks at the system from the view of an outsider and comes up with real life scenarios that are tested. Different aspects of the software system are tested such as the load, its performance under certain conditions, its reliability in terms of access and availability in case of errors, its security and so on. The system is tested at all possible points from different angles so as to uncover as many defects as possible.

A simple analogy of the development of a ball point pen can be used. The cap, the ink cartridge, the body are made and tested separately. Each of these items is tested individually which is the unit testing part. When more than one unit is ready, they are combined and tested together incrementally adding a unit. This is the integration testing part of the process. When the entire assembly of the pen has been completed, it is now tested as a whole. This stage is the system testing stage where the pen will be tested

thoroughly. The pen is tested for its functional and non functional behavior and its interaction with other systems (different kinds of paper, tape, etc.) is checked.

This is the test that is the last one that is performed before the product is presented to the user or the stakeholders for testing.

4. Acceptance Testing: It is the final level of testing where the system is tested for compliance of its business needs and requirements (Bath and McKay, 2014). This testing focuses on testing the system, its behavior, and its capabilities as a whole which is similar to system testing. The difference from system testing is that here, the business needs are considered. The readiness of the system to be deployed in production is tested in acceptance testing. Briefly, the acceptability of the system by the end user or the business is tested at this level. The system is tested and evaluated and it is decided by the stakeholders whether the software system is acceptable for the purpose of the final delivery or not (Bath and McKay, 2014).

Acceptance testing is also used to ensure that the terms of the contract and legal requirements have been met by the software application. The system is tested and evaluated for use by the potential clients or end users. Whether the software system satisfies its legal and regulatory needs or standards is checked at this point and an informed decision is made. Different things are tested by means of acceptance testing. The user's acceptance is tested here, the contractual or legal acceptance of the system is tested, the operational acceptance is tested and sometimes, alpha, and beta acceptance test is performed as well. In most cases, a user's acceptance of the system under test is done at this stage.

The validity of the system for use by a real user in a target or simulated environment is done. The goal/target here is to ensure that the software applications performs the way it is supposed to and that it meets the needs of the user and business processes can be executed by the user seamlessly without a lot of difficulty. The intent is to make sure that the user can work with the system in a smooth and consistent fashion. The system is tested so as to validate its requirements and to ensure that it will work as expected. The quality/standard of the entire software system or application is established at this level. Ideally, not a lot of defects should be found at this stage.

The operational acceptance of a software system is done in order to ensure that the system performs well in case of major failures such as loss of resources and unforeseen circumstances. Administrative actions are tested in this case. The behavior of the system in case of failures is tested, the

data recovery is analyzed and tested, disaster handling is checked, backup mechanisms are tested, and recovery mechanisms are tested and so on. The system's performance under pressure is tested and how it reacts in adverse scenarios is evaluated. The system is tested for vulnerabilities in the security and faults are identified. This operational level testing is usually conducted by the system administrators.

The legal and contractual testing, as discussed, aims to test the compliance of the software system to the accepted and agreed upon acceptance criteria that is described in terms of a contract. The software system is tested and evaluated to ensure that regulatory standards at regional and national levels have been met by the software. Safety regulations that are required to be met are tested as well. This type of testing is performed generally by the users of the systems (Bath and McKay, 2014). In case of large applications, an independent group of testers are hired so as to perform this test.

Alpha and beta testing is a unique form of acceptance testing. It is a type of acceptance testing that focuses on acquiring some feedback from the users or customers of the application before it has been made available to the public. Generally, large scale commercial software providers employ the use of alpha and beta testing so as to gain useful feedback regarding the product from the end users itself. The goal of this type of testing is to get an idea from the potential pool of customers regarding their experiences with the use of the software application and gather their observations. The reactions of the potential users, customers, or operators of the system are obtained so as to ensure that the customer's needs have been satisfied.

Alpha testing is organized by the software provider but is conducted by the potential clients of the system. In some cases, an independent team of testers perform alpha testing. On the other hand, Beta testing is usually performed at the client or customer locations. Beta testing is always implemented by either the existing users of the systems, or the potential users of the system. Each one of the testers performs the test at their location depending on the context (Bath and McKay, 2014). Generally beta testing follows alpha testing. But, sometimes, alpha testing is skipped altogether. Usually, the decision to conduct either alpha or beta tests or both is made by the software provider.

The ultimate goal of these different types of acceptance tests is to gain feedback from potential or actual users of the application and the stakeholders of the system. This kind of testing serves to get quality feedback from the customer and expose defects and inconsistencies in the functioning of the system. The workflows of the system are tested by the users or stakeholders

so as to ensure that business requirements are met. This process tries to uncover defects or bugs that are non functional and also tries to identify security issues in the system. The stakeholders and users make sure that the system adheres to the legal standards that were determined and agreed upon. The functioning of the system under normal conditions is tested and its behavior is witnessed by the testers who report unseemly events or faults.

Acceptance testing of all types aims to build trust of all the stakeholders in the software system. The target is to make sure that all the people that would potentially use the system are confident in the ability of the system to perform well. This testing serves to ensure that users, clients, customers, and even the stakeholders can seamlessly use the system and interact with it under regular operational environment without facing a lot of issues. It strives to ensure that the users can perform all of their standard operations without facing severe system errors. They should be able to operate the system with minimal issues, no additional costs, and no risk. The goal is to ensure that the behavior of software system is consistent under all conditions. In an iterative software development life cycle, different forms of acceptance testing can be implemented at the end of every iteration or after a series of iterations.

The four different levels of testing can be summarized as follows:

Level of Testing	Purpose
Unit/component testing	Testing of the individual units or components
Integration testing	Testing of the integration of two or more components
System testing	Testing of the entire system as a whole
Acceptance testing	Testing of the final system as per the business requirements

Now that we have seen the different levels of testing, we proceed to understand how these levels apply to either black box or white box testing techniques in the upcoming sections.

5.4. CLASSIFICATION OF TEST TYPES

Testing is further classified into different types namely functional, non-functional, and regression. We shall investigate these types of testing now.

5.4.1. Functional Testing

This aims to test the functional requirements of the system under consideration (Bath and Van Veenendaal, 2013). The different tasks and functions that are meant to be performed by the system are evaluated and tested in this type of testing. Functional tests can be implemented to check the functions performed by the system. They can be performed at all the levels of testing but usually it is not done at the unit/component level. Functional testing specifies what the software system should do. Different items or products generated in the software development process such as the requirement specifications, the functional specifications, the user stories, the use cases, the business requirements, etc., can be used as a basis for creation of functional tests.

Mainly, functional testing focuses on the features of the software product. The test team creates test cases based on the specifications/requirements and test the system's functionality. Inputs are provided to the system and the generated outputs or results are compared with the expected values for validation of the test case. Generally, functional testing is carried out by the test team as the internal working of the code is not considered in this case. No previous knowledge of programming is needed for implementing functional test cases. This type of testing considers the behavior of the system and hence black box testing techniques are mainly used to implement the test cases for validating the functional requirements of the software system (Howden, 1987).

5.4.2. Non-Functional Testing

It focuses on the non-functional properties or characteristics of a software system such as its performance, usability, robustness, security, endurance, load management, etc. The ISO standard ISO/IEC 25010 classifies different characteristics of software products based on quality (Homes, 2013). The non-functional type of testing focuses on how well the system works and behaves in certain conditions. This testing concentrates on the non-functional requirements of the software system that help understand and evaluate the software system based on certain properties and reach conclusions regarding the behavior of the software system.

This testing can be performed at all the testing levels. Non-functional testing serves to answer the question-“How well does the software perform?” The intention is to test certain parameters of the software system which would never have been tested by means of functional tests. The readiness of the system for handling of non-functional parameters such as speed, performance, load, etc., is tested using this type of testing. The purpose is to address characteristics of the system that cannot be tested by other methods of testing such as functional testing. This method of testing aims to increase the quality of the software product in terms of its usability, performance, efficiency, and maintainability. Further, it helps to reduce costs and risks associated with the non-functional parts of software. Additionally, this method helps collect different measurements and overall data or metrics regarding the quality of the software under test. A list of quantifiable metrics can be generated by means of non-functional testing which helps build a strong base to justify the level of quality of the software product.

Furthermore, non-functional testing tries to optimize the behavior of the software system in terms of its quantifiable characteristics in order to make the product more users friendly, manageable, and maintainable. This method is an excellent way to generate quantifiable and objective metrics to demonstrate the quality of the software system. As non-functional parameters have to be of a quantifiable nature, there is no ambiguity in the way the measurements are interpreted which is good for gaining the client’s confidence. Each time a property is measured and tested, it can be documented, and the software system can be improved to generate optimal values for the calculations of the quality values of these characteristics. This process helps gaining knowledge about the software system and its behavior and the different technologies used by the software system.

For instance, a simple example of a non-functional attribute is checking the amount of load a software system can handle. Load can be subjective. Let us consider a simple example that consists of measuring the number of users who can login to the system at the same time. A non-functional test would be to simulate multiple login attempts at the same time and see how many users can be logged into the application at the same time without causing any problems in the functioning of the application.

There exist a wide range of characteristics that can be tested non-functionally. Some of them are: security, availability, performance, efficiency, integrity, usability, flexibility, portability, scalability, reliability, recovery, survivability, etc.

Performance testing is one of the most frequently implemented testing techniques of non-functional testing. The goal of this test is to observe the response of the system and ensure that it falls under the optimal range and that it handles the network load (Fairley, 1978).

Another characteristic that is tested in the process of non-functional testing is the recovery management of the system. The recovery testing process tests the ability of the software system to handle disaster situations. The goal of recovery testing is to see if the system can recover from software and hardware malfunctions. This helps gain insight into the behavior of the system and uncover opportunities to improve the recovery management at all test levels.

A further key property that is often tested is the security of a software system. The security of any software system is important and in some cases, critical to the end users. Security testing aims to ensure that the system is robust. The purpose of testing the security of the system is to ensure that proper authentication and authorization is done and to ensure that unauthorized access to the application is not allowed. The security parameters aim to check the safeguarding mechanisms of the software against malicious attacks. Different types of internal and external sources of attack are tested and metrics are obtained. This method helps in improving the robustness and safety of the system.

Another parameter of non-functional testing is the reliability of the software system. The target is to test whether the system is available and functioning in a continuous manner for longer periods of time without any major failure. This parameter helps determine how long the system can perform all of its functions continuously without issues (Afzal, Torkar, and Feldt, 2009).

Another crucial non-functional parameter that is tested is the scalability of the software system. A software system ideally should be scalable. It should be able to respond to a rise in demand without a lot of issues. It serves to determine the ability of the system to react or handle a growth in the need of its services. It is way to determine the point or load at which the system would degrade. The system should be able to handle changes in size or the volume of demands and see where it stops or degrades. The purpose of this exercise is to identify how scalable the application is and analyze if the scalability could be improved.

Non-functional parameters of testing are important to the quality of software in general. These parameters or properties help increase the quality of the software product and help reduce risk. Non-functional parameters must

be tested at all levels if possible as they help uncover issues early on. If non-functional parameters are not tested in time, late discovery of bugs related to non-functional characteristics can prove costly and hinder the success of the project. Non-functional test cases do not need specific knowledge of the internal workings of the software system and hence black box testing techniques are used to develop non-functional test cases. These tests that are generated using Black box testing techniques help in determining the quality of the software system in terms of non-functional characteristics.

5.4.3. Regression or Change-Related Testing

Regression testing or change related testing is performed when changes are made to the existing software system usually for correction of a bug or to change the functionality of the software system (Graham, Van Veenendaal, and Evans, 2008).

This type of testing is common and is generally done in order to confirm that the changes to the software system have not resulted in any unforeseen and disastrous consequences. Once a change is introduced to a particular unit or subsystem of the software, the test cases associated with the unit or the system are executed again. In addition to the existing test cases, there is a possibility that the change needs further test cases pertaining to its context.

This is because new changes that are introduced to the system may change the context for the older test suite and this may break the tests.

Furthermore, there is a chance that changes made in one part of the code can impact other sections of the code unwittingly. Software is inherently complex and sometimes a small change in one section of the code can trigger an unwarranted response in another section of the code. A miniscule change can cause the software to react in a poor way and introduce defects in the system.

Such a reaction is called a regression in the software. Regressions in software need to be identified and sniffed out after changes have been made to the software.

There could be implications of a change within the same unit, or within a similar unit in other section of the code or even within another sub-system in the software. Such regressions cannot be predicted all the time and hence regression testing is an important part of the testing cycle.

This testing can be performed at all levels of software testing. This is because regressions can occur in any part of the code which makes it

imperative that the system be tested after changes have been introduced to it. Frequent changes made to code can cause regressions and hence the system must be tested for defects of this kind. As software evolves over its life cycle, different versions are introduced and each version must be tested for regression related defects.

Test suites that test for regressions must be maintained and updated periodically. As regression testing should be done often, it is a good candidate for the process of automation. As it is repeated several times over the lifecycle of the software system, it can be automated so as to reduce the time and effort spent by the test team.

Regression testing focuses on testing the functionality of the system and verifies that the behavior remains the same after changes. Hence, no in depth knowledge of the system's technical implementation and code is required to perform this testing. Therefore, black box testing techniques are used to develop and generate test cases for regression testing.

Now that we have seen different types of testing, we shall see how each type of testing is applicable to different test levels.

5.5. TYPES OF TESTING AND THEIR APPLICATION TO TEST LEVELS

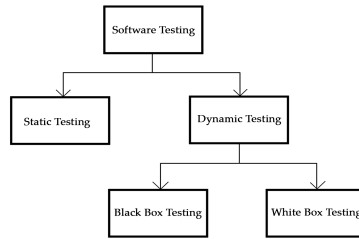
As we have seen in the previous section, there exist different types of testing and different techniques that can be used to write test cases for each of these testing types.

Each type of testing is applicable to different levels in software testing and is used in a way that is beneficial to the quality of the software in some way. In this section, we shall see how different techniques and types of testing can be applied and implemented at different test levels so as to improve efficiency of the testing process.

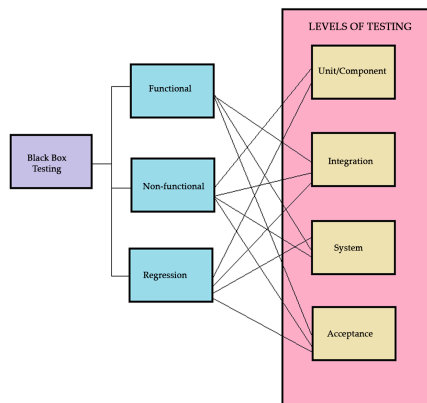
As we are aware, there are three types of testing: functional, non-functional, and regression.

We have covered that the test cases or test suites for each one of these types of testing can be derived or generated from using black box testing techniques. Hence, indirectly, the three types of testing can be classified as black box testing types.

The classification of different types of testing can be shown in the figure below:



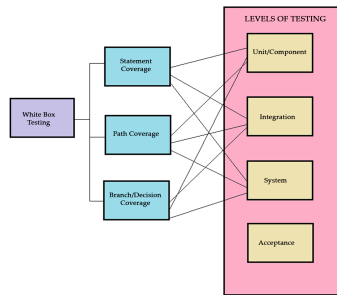
As we can see, we have two main categories of testing namely static and dynamic testing. Dynamic testing is further classified into block box and white box testing. We shall now see how Black Box testing is categorized.



As we have seen, almost all types of black box testing can be performed at all the test levels. But, in practice, block box testing is mostly done at the system and acceptance testing level.

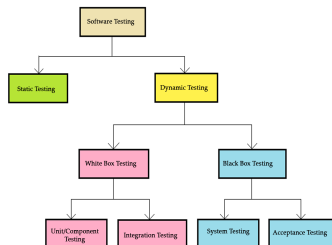
This is because, in case of some software systems, at the unit/component level and integration level, black box tests cannot be written as the software is at a stage where the working of the code is necessary to write the tests. Also, at times, the software system at the unit or integration level is not necessarily an independent subsystem that can be individually tested. Hence, sometimes, black box testing is not done at the unit and integration levels.

Now that we have seen the classification of different black box testing, we shall take a look at how white box testing can be classified.



As we can observe, there are three types of white box testing, applicable at the unit, integration, and system level. White box testing can be applied at the unit, integration, and system level, in theory. In case of system level white box testing, it is generally implemented to test path coverage between different internal subsystems of the software application. But, in most software projects, white box testing is limited to the unit and integration levels of the system. In some projects, it is limited to only unit testing.

Based on the test level, in practice, the classification of different testing approaches can be classified as follows:



It should be heeded that the above classification is based on the level of use of different testing approaches at different levels of test. In practice, software projects tend to lean towards making use of black box testing methods for system and acceptance testing and white box testing methods for unit and integration testing. But this is not true for all the projects. In theory white box testing techniques and black box testing techniques can be applied at all levels of software testing. The implementation of the testing techniques depends largely on the type of the software project and its needs. We shall now see some factors that need to be considered when choosing a test strategy for a software project.

5.6. FACTORS TO BE CONSIDERED WHEN CHOOSING A TESTING TECHNIQUE

When developing a test suite for any software project, it is important to consider the fact that not all the available test techniques would be suited to the software project. There is a chance that based on certain characteristics of the software project it would not be feasible to apply some of the test techniques. Hence, it is necessary to choose the test techniques that would be best suited to the software project under test. Different factors pertaining to the software project must be considered such as project budget, risk factors, SDLC model, customer requirements time and budget constraints, knowledge base of similar products and so on.

We shall now see some of these factors and how it affects the test technique that is chosen for testing the software product.

5.6.1. SDLC Model of the Project

In software, there exist different life cycle models that can be used for development of a project. Each software development life cycle model is unique in its own way and the way in which test cases are developed can change based on the model that is being used. For instance, in case of the waterfall model, which is not used as much now, as testing is done at the end, all the tests need to be developed by the end of the development cycle which gives the test team enough time to develop a strong test suite which offers a lot of code coverage. As the test team has time, they can come up with a sizeable quantity and quality of test cases before they start their testing. But, for other software models such as the agile model, where software is developed in quick iterations, the tests need to be identified in a practical manner. As it is a fast paced approach, the test team must be smart in the way they choose a test technique based on each iteration and must have a test suite that is flexible and ensures proper coverage in each iteration of the product. Just as the product is improved during every iteration of the product, the test cases need to be improved as well. The test team must wisely choose the techniques that work well for this model of software development.

5.6.2. Risk Factors

Each software project comes with an associated level of risk. Each software product requires that a certain degree of risk be handled by the process of quality assurance which defines how the test cases will be written for the

project. Certain software projects such as those that handle government services handling critical user data can be classified as high-risk projects. These services are critical and hence these software services must not be susceptible to attacks. Also, such services should be available at all times unless there is a scheduled upgrade or maintenance. These factors add to the risk of the project. In such cases, functional, and non-functional testing techniques can be implemented so as to test the functionality along with non-functional parameters of high importance such as security, availability, robustness, fault tolerance, etc.

On the other hand, an uncomplicated and regular software system such as an apparel delivery site does not possess the same level of risk. In this case, the non-functional testing that is done would be less extensive than as compared to the previous scenario. The testing techniques that should be chosen in this scenario differ from those that would have been chosen for the previous one. This is just a simple example, but in practice, several factors of risk need to be considered and tests need to develop accordingly.

5.6.3. Customer Requirements

In case of certain software projects, the customer has certain requirements that need to be fulfilled. These requirements have to be considered when developing test cases. A customer can have certain needs related to the availability of the product or the number of people that can login to the system or the type of recovery methods and so on. These needs should be considered when writing test cases for the software project as it helps prove to the client the quality of the product and that the product satisfies the client's requirements. For example, for a software product that is a website offering different services, if the client has a requirement that the software be available only during the day between certain hours, this should be considered while developing the test suites. This would help demonstrate to the client that the system works as requested.

5.6.4. Time and Budget Constraints

One of the major issues or constraints of any software project is money. The second issue is time. Both these points are interrelated. The amount of time spent on a software project directly impacts and is linked to the cost of the project. Hence, an optimal balance needs to be found between the amount of tests and the time required to develop the test suites. The size of the testing team should be considered as well. If the testing team is small, they probably

would not be able to execute a large number of tests in a limited amount of time. Hence, they would need to optimize their test strategy and choose the proper techniques so as to minimize the amount of test cases while still maintaining the test coverage.

5.6.5. System Complexity

The complexity of any software system plays a crucial part in the approach used for testing it. Different software systems have different levels of complexity based on the functioning of the project and its purpose. The internal complexity is also a factor that needs to be considered while developing the test cases. Additionally, the components or subsystems may differ in their complexity which needs to be considered during the testing phase. A simpler module or unit may need fewer tests whereas a complex system or subsystem that performs critical functions would probably require a substantial amount of test cases to test it. For instance, a system performing computations for NASA would be highly complex in its structure as compared to a simple software system that provides clothing to people. The approach to selecting testing techniques would be different in each case as the complexity of the software must be considered in each case.

5.6.6. Regulatory Standards

As we have seen, each software system that is developed is subject to the law of the country it is developed in and to the countries it will be deployed in as well.

Each country has its own rules of governance where software is concerned and these rules must be met by the software providers. The software must adhere to the standards set by the regulatory laws of the country and this should be considered when developing the test cases. Having knowledge of the standards and rules plays an important role in the process of testing as these standards are set as a rule which must be respected by the software system.

For instance, if we consider a simple, uncomplicated apparel selling website, it may have a return policy of 30 days in France whereas in the UK, this would be 15 days. This information is useful in determining the type of test cases to be written in order to test both the scenarios and ensure that the software meets the standards of each country.

5.6.7. Component or System Type

A software application is made up of several components with a varying degree of complexity and functionality. Each component is special in its own way which is why the test cases based on the components or subsystems of the software product vary as well. This is especially true in case of white box testing techniques where the number of test cases and type of tests depends on the component under test. Each component warrants a test suite or test cases that are best suited to test the component well.

5.6.8. Expected Functionality of the Software

Each software system is built with the intention of doing a specific set of tasks. The type of tasks can vary from high risk to low risk. The functionality provided by the software can also play a part in defining the test cases for the system. A high risk and critical software system would be tested extensively with focus on aspects such as security, fault tolerance, availability. For instance, there would be a difference in the test suite for a software component used by NASA as compared to a simple online apparel site. In case of a software application developed for NASA, the test cases would focus on testing for high security, availability, scalability, and impeccable fault tolerance. Such a system should be available at all times and should be extremely fault tolerant. The same cannot be said for an apparel website. The impact is low in this case and the functionality offered does not demand a higher quality of the non-functional parameters which should be considered during the test phase. These are just a few of the factors that are considered when one chooses a test technique. Based on the maturity and the version of the system in question, the experience of the test team with the system, whether one is at an early stage of the life cycle of the product or if one is simply trying to improve the quality of code, the decision of the test techniques will vary. We shall now see in brief the steps that are followed to perform both White Box and Black Box testing techniques.

5.7. STEPS TO PERFORM DYNAMIC TESTING

In this section, we will cover the steps taken to perform black box and white box testing on software.

5.7.1. Steps to Perform Black Box Testing

Black box testing can be performed by following the steps below (Khan and Sadiq, 2011):

- Examining the software specifications and requirements in the early stages of the development life cycle. A well written precise and accurate document or System Requirement Specifications document must be present.
- Selecting of valid input parameters for the test cases (both positive and negative scenarios of test) so as to check the response of the software system. This step involves evaluation of the sets of valid inputs and the test scenarios in order to ensure the highest coverage possible.
- Determining the expected results on the basis of the given output.
- Constructing the test cases with the selected input parameters.
- Executing the prepared test cases.
- Comparing the actual outputs with the expected outputs and marking them as passed or failed.
- The failed test cases and scenarios are sent to the development team so that they can fix the issues.
- Re-testing the system once defects have been fixed in order to ensure that the defects have been fixed.

Let us see a simple scenario that would make use black box testing.

We consider a webpage that has been developed to take in a username and password. In this case, the tester will test the functionality provided by the web page which is the login function. The tester will consider all the valid and invalid inputs for the fields and check to see if the login function works well for the valid inputs for the two fields. The login is a combination of the correct username and password and the tester will check a correct and an incorrect combination for the two fields. Here, the tester will not look at the code; he/she will only test if the functionality provided is working properly without issues. Different scenarios such as valid credentials, invalid credentials and one invalid credential will be considered. The tester will aim to check if the page provides the expected output values for the inputs provided. Comparison of the expected outputs and the actual outputs is done and the results are documented. If any tests fail, feedback is given to the development team so that they can correct the defects found, if any. Once corrected, the testers test the functionality once again.

5.7.2. Steps to Perform White Box Testing

White Box testing can be performed by following the steps presented below (Khan, 2011):

- Understand the code and learn about the functionality implemented in the code. The source code needs to be understood in depth in order to perform white box testing.
- Develop and finalize a test strategy.
- The next step is to design and outline different test scenarios and test cases and perform their prioritization. Based on the priority levels defined by the test techniques, each of the test scenarios and cases are assigned a priority which helps determine which test cases give the most value in terms of quality.
- Preparation of the test environment for the purpose of the execution of tests.
- Execution of the test cases at runtime is performed. It encompasses the study of the code when it is running and examining the utilization of resources by the software system. The results and behavior of the system are analyzed. Different parameters are observed such as the time taken, the resources used, areas of the code that were not accessed are studied.
- Testing of control and conditional statements is done in order to check the efficiency of the system for a variety of input parameters.
- Generation of test reports.
- The final step is the security testing where all the possible outlets or flaws in the security of the system. The code is analyzed for oversights and loopholes in its implementation of security.

Let us see a simple overview of a white box test. We consider a webpage that has been developed to take in a username and password. In this case, the person testing the piece of code that performs the process of logging in the user will consider all the valid and invalid inputs for the fields. We have two fields in this case, the username, and the password. So, the tester and/or the developer will study the implementation of each field. The goal would be to determine all the legal and illegal inputs for each of these fields. All the valid and invalid options would be tested. The inputs for these fields would be tested and the output would be verified against the expected values and the code would be studied. In short, the code would be examined in order to check if it is working correctly or not and if it isn't, it is investigated further. Now that we have studied how black box and white box testing is performed, we shall cover their practical implementations in the next chapter. We shall focus on practical implementations for unit testing of software applications.

Software Testing Frameworks

CONTENTS

6.1. Junit.....	164
6.2. Testng Framework	205
6.3. Selenium Framework	231
6.4. Mockito Framework.....	257

In this chapter, we will look at certain frameworks that can be used for developing unit tests for developers. We shall be focusing on frameworks that are meant for creating tests for software written using the Java programming language.

There exist a variety of frameworks that are now available to developers to help them write effective unit test cases for their code. We shall be taking a look at few of them in this chapter.

Some of the frameworks available for creating unit test cases in the Java programming language are: JUnit, JTest, Testng, JMockit, Mockito Cucumber, etc. In this chapter, we shall see the use of the following frameworks: JUnit, TESTNG, SELENIUM, and MOCKITO.

We shall discuss each of these frameworks and then proceed to see how these frameworks can be used for unit testing.

6.1. JUNIT

JUnit is a framework for unit testing software written in the java programming language. It is one of the most extensively used frameworks for unit testing of java code (Gamma and Beck, 2006). The latest stable release version of JUnit is JUnit 5 version number 5.6.2 (JUnit 5, 2020).

The JUnit platform is made up of three sub modules or projects namely the JUnit platform, JUnit Jupiter, and JUnit Vintage modules.

The JUnit platform is a foundation that helps launch testing frameworks on the Java Virtual Machine.

This module consists of the TestEngine API that helps develop a testing framework which can be run on the platform. This component also provides a command line type interface that permits running of tests from the command line called Console Launcher (JUnit 5, 2020).

JUnit Jupiter is a module that is composed of new programming modules and extension modules for developing test cases using JUnit5. New API and annotations have been added in this module (JUnit 5, 2020).

JUnit Vintage is the sub module that possesses a TestEngine that provides backward compatibility for older versions of JUnit. Test cases developed in previous JUnit versions such as JUnit 3 and JUnit 4 can be executed using this module (JUnit 5, 2020).

JUnit is fairly easy to understand and implement. JUnit 5 needs the java JDK version 8 or higher in order to execute at runtime.

Before we look at a concrete example, we shall take a few basic annotations that are used in this framework. The following annotations present in the JUnit Jupiter module are mainly used when developing unit tests using JUnit:

1. **@Test:** This annotation is used to indicate that the method is a test method.
2. **@TestFactory:** This annotation is used to designate a method to be a factory method meant for use in creation of dynamic tests.
3. **@ParameterizedTest:** This annotation is used to denote that test is a test method that is parameterized and accepts parameters while execution.
4. **@RepeatedTest:** This annotation is used to define a method that is a template for a test that can be repeated certain number of times. The total number of repetitions required is stated when using this annotation.
5. **@BeforeEach:** This annotation means that the method should be executed before each test method in the test class with either of the following annotations: `@Test`, `@TestFactory`, `@ParameterizedTest` or `@RepeatedTest`.
6. **@AfterEach:** This annotation means that the method should be executed each time after a test method is executed in the test class annotated with either of the following annotations: `@Test`, `@TestFactory`, `@ParameterizedTest`, or `@RepeatedTest`.
7. **@BeforeAll:** This annotation states that the method annotated with `@BeforeAll` should be executed before all the test methods are executed.
8. **@AfterAll:** This annotation states that the method annotated with `@AfterAll` should be executed after all the test methods are executed.
9. **@Disabled:** This annotation is used to disable the test case or the entire test class. This is used to exclude the class or the method from the test suite.
10. **@TestMethodOrder:** This annotation serves as a configuration annotation that is used to configure the order of execution of the test cases in a test class or a test suite. This annotation is used at the class level.

11. **@DisplayName:** This annotation is used to provide a custom name to the test class or test method. The custom names that are configured using this annotation are displayed in the test results and reports. This annotation can be used at the class or method level.

The latest stable release of the JUnit framework is 5.6.2 which is the version we shall be using in our examples. Now that we have seen some basic JUnit5 annotations, we shall first discuss the setup required followed by practical examples of test cases written using the JUnit framework in the succeeding sections.

6.1.1. JUNIT SETUP

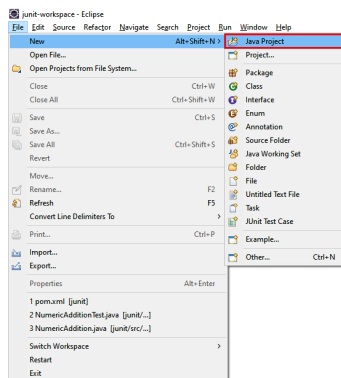
JUnit 5 is composed of several projects and hence requires several dependencies to be fulfilled before one can start writing a test case using the JUnit framework.

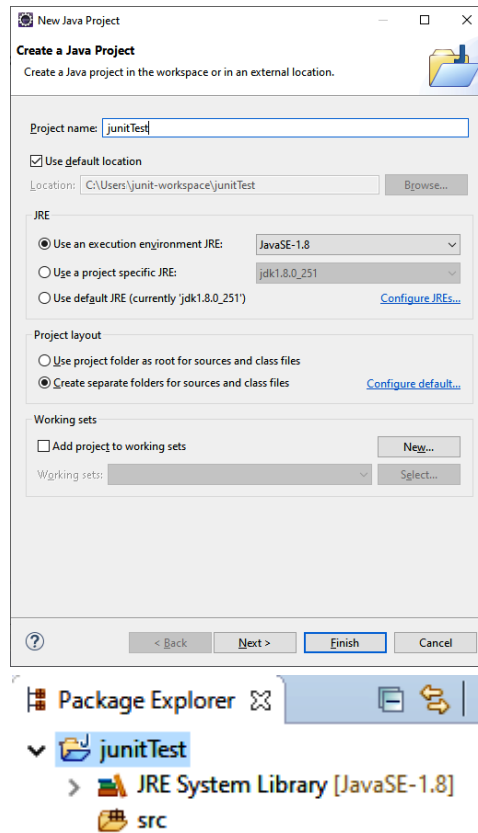
There are two ways in which JUnit can be configured to work in your java project. The first option is to add the jar files directly to the project. The second option is the use of maven to inject the required JUnit dependencies.

Let us see how each of these options can be implemented. We will be using Eclipse Oxygen version 1a for the examples in combination with JUnit version 5.6.2.

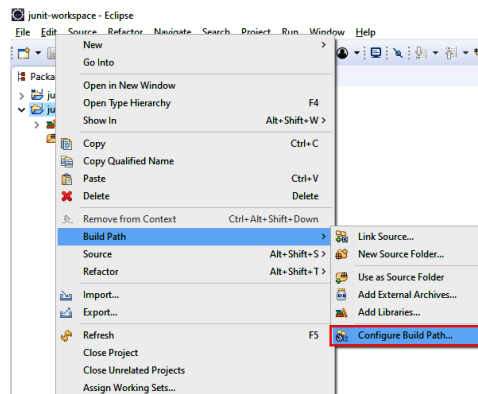
1. **Add Jar Files Directly to the Project:** A simple way of configuring JUnit in your project is the direct addition of the necessary jar files to the project. The jar files can be downloaded online. The link that we have used to download the jar files is: <https://jar-download.com/artifact-search/JUnit5>.

The steps to be followed are shown below. We first create a simple Java project.

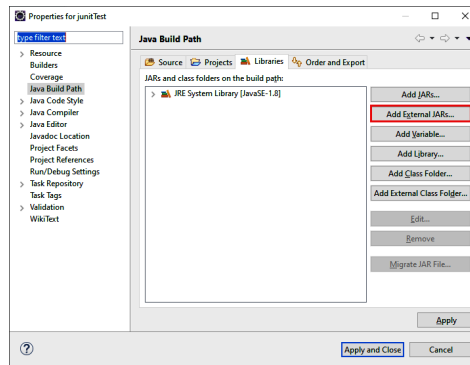




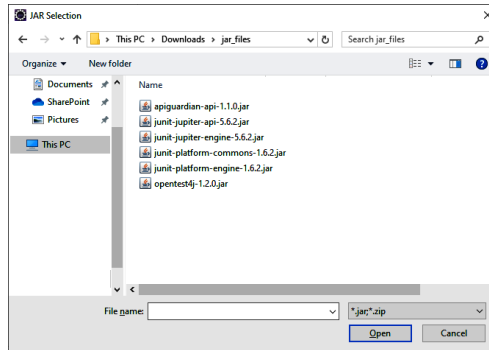
Once the project is created, we right click on the project and navigate to the Build Path -> Configure Build Path Option. This is shown below:



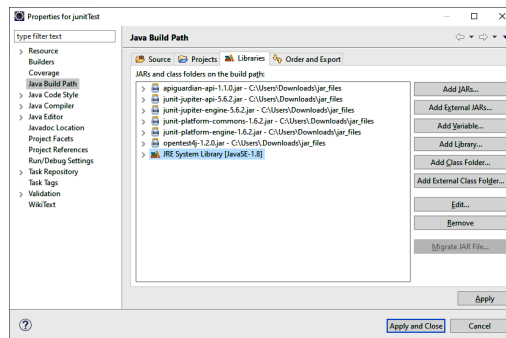
A popup is displayed.



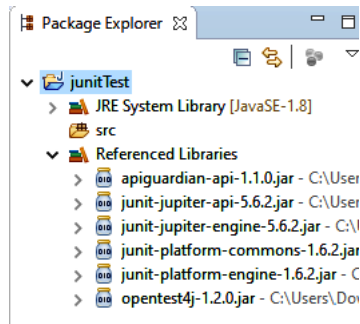
We click on the ‘Add External Jars’ button and navigate to the path where the downloaded jar files are present as shown below:



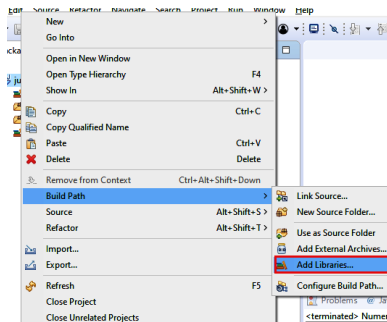
The jar files are added as shown below:



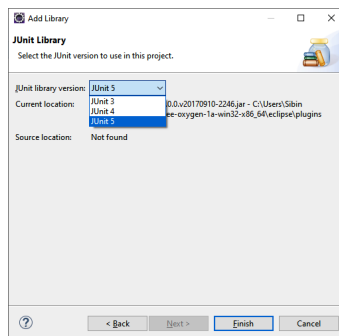
We click on the ‘Apply and Close’ button. As we can see from the diagram below, the dependencies have been added to our java project.



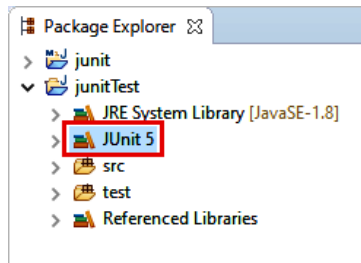
Now that the dependencies have been added, we also need to add the JUnit5 library to the project. This is done as follows:



The first step is to right click on the project, navigate to the Build Path tab and click on the ‘Add Libraries’ option. A popup is displayed as follows:



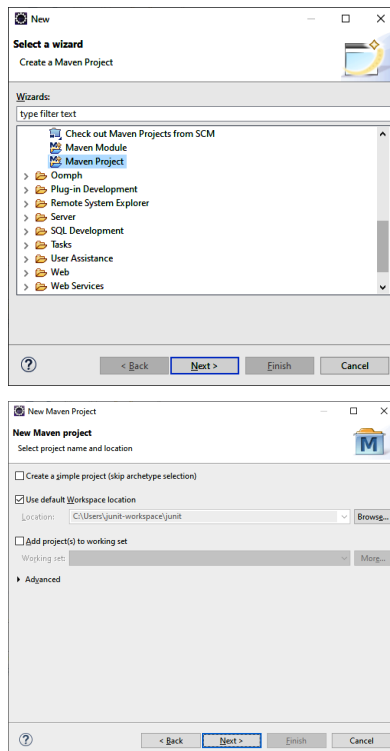
We select the JUnit5 library option from the dropdown list and click on ‘Finish.’ This successfully adds the JUnit5 library to our java project as shown below:

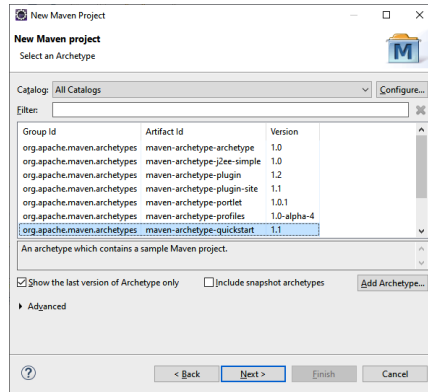


2. **Use Maven to Inject Dependencies:** Maven is a powerful tool that helps in automation of the build process for Java projects. It can be used to inject the required JUnit dependencies in the java project. The setup of JUnit5 using maven is easy and effortless to implement. The JUnit dependencies need to be simply added to the pom.xml file in the maven project and maven automatically downloads the required jar files and adds it to our project.

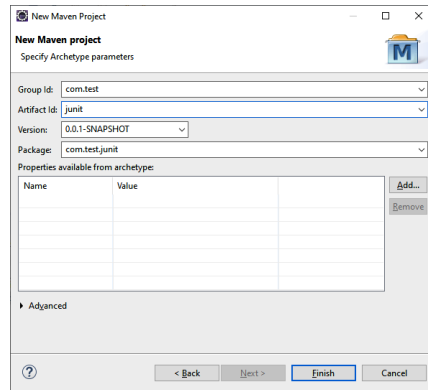
The steps to setup JUnit5 using maven are as follows:

We first create a simple maven project. We navigate to File -> New and choose to create a maven project.

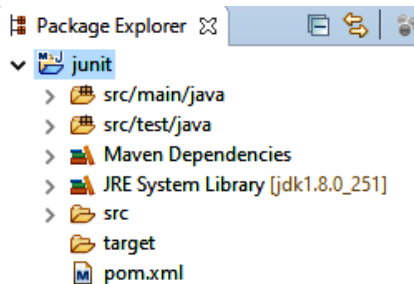




We use the quick start artifact to create the project. We provide the name of the project as shown below:



The project is created as shown below:



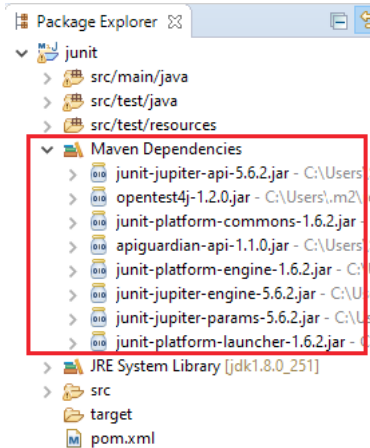
Once the project is created, we proceed to modify the pom.xml file to add the required JUnit5 dependencies. JUnit5 requires the following dependencies: JUnit-jupiter-api, JUnit-jupiter-engine, JUnit-platform-runner, apiguardian-api, JUnit-platform-commons, opentest4j.

This is shown below:

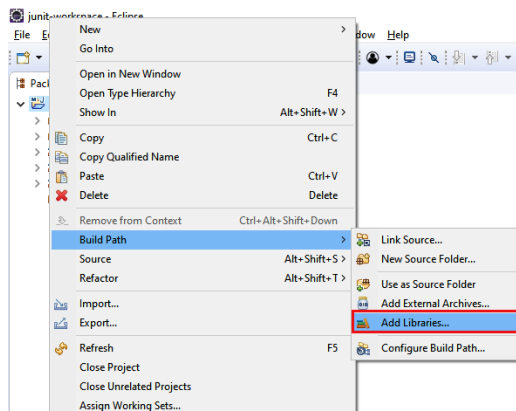
```

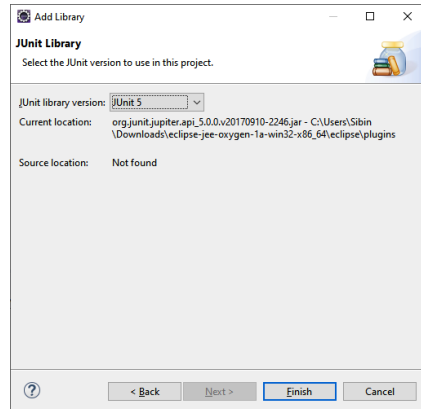
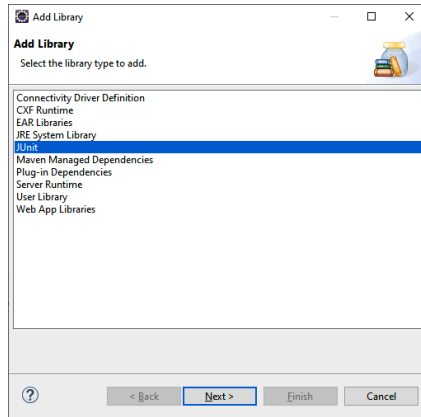
1  @project_urls = {
2    "homepage" = "http://www.apache.org/PODS/4.0",
3    "source" = "http://www.apache.org/dist/maven/4.0.0/src/maven-4.0.0-src.tar.gz",
4    "distribution" = "http://www.apache.org/dist/maven/4.0.0/maven-4.0.0-bin.tar.gz",
5  }
6
7  # Gradle Core
8  gradle_core = test {
9    artifact {
10      name = "gradle-core"
11      version = "4.0.1"
12      packaging = "jar"
13    }
14  }
15
16  core_jail_core = core {
17    url = "http://www.apache.org/"
18  }
19
20  @dependencies {
21    @project.build.sourceEncoding = "UTF-8"
22    @project.build.sourceEncoding {
23      @project
24    }
25  }
26
27  @dependencies {
28    @dependencies {
29      gradle_core {
30        artifact {
31          name = "gradle-core"
32          version = "4.0.1"
33          packaging = "jar"
34        }
35      }
36    }
37  }
38
39  @dependencies {
40    @dependencies {
41      gradle_core {
42        artifact {
43          name = "gradle-core"
44          version = "4.0.1"
45          packaging = "jar"
46        }
47      }
48    }
49  }
50
51  @dependencies {
52    @dependencies {
53      gradle_core {
44        artifact {
45          name = "gradle-core"
46          version = "4.0.1"
47          packaging = "jar"
48        }
49      }
50    }
51  }
52
53  @dependencies {
54    @dependencies {
55      gradle_core {
56        artifact {
57          name = "gradle-core"
58          version = "4.0.1"
59          packaging = "jar"
60        }
61      }
62    }
63  }
64
65  @dependencies {
66    @dependencies {
67      gradle_core {
68        artifact {
69          name = "gradle-core"
70          version = "4.0.1"
71          packaging = "jar"
72        }
73      }
74    }
75  }
76
77  @dependencies {
78    @dependencies {
79      gradle_core {
80        artifact {
81          name = "gradle-core"
82          version = "4.0.1"
83          packaging = "jar"
84        }
85      }
86    }
87  }
88
89  @dependencies {
90    @dependencies {
91      gradle_core {
92        artifact {
93          name = "gradle-core"
94          version = "4.0.1"
95          packaging = "jar"
96        }
97      }
98    }
99  }
100
101  @dependencies {
102    @dependencies {
103      gradle_core {
104        artifact {
105          name = "gradle-core"
106          version = "4.0.1"
107          packaging = "jar"
108        }
109      }
110    }
111  }
112
113  @dependencies {
114    @dependencies {
115      gradle_core {
116        artifact {
117          name = "gradle-core"
118          version = "4.0.1"
119          packaging = "jar"
120        }
121      }
122    }
123  }
124
125  @dependencies {
126    @dependencies {
127      gradle_core {
128        artifact {
129          name = "gradle-core"
130          version = "4.0.1"
131          packaging = "jar"
132        }
133      }
134    }
135  }
136
137  @dependencies {
138    @dependencies {
139      gradle_core {
140        artifact {
141          name = "gradle-core"
142          version = "4.0.1"
143          packaging = "jar"
144        }
145      }
146    }
147  }
148
149  @dependencies {
150    @dependencies {
151      gradle_core {
152        artifact {
153          name = "gradle-core"
154          version = "4.0.1"
155          packaging = "jar"
156        }
157      }
158    }
159  }
160
161  @dependencies {
162    @dependencies {
163      gradle_core {
164        artifact {
165          name = "gradle-core"
166          version = "4.0.1"
167          packaging = "jar"
168        }
169      }
170    }
171  }
172
173  @dependencies {
174    @dependencies {
175      gradle_core {
176        artifact {
177          name = "gradle-core"
178          version = "4.0.1"
179          packaging = "jar"
180        }
181      }
182    }
183  }
184
185  @dependencies {
186    @dependencies {
187      gradle_core {
188        artifact {
189          name = "gradle-core"
190          version = "4.0.1"
191          packaging = "jar"
192        }
193      }
194    }
195  }
196
197  @dependencies {
198    @dependencies {
199      gradle_core {
200        artifact {
201          name = "gradle-core"
202          version = "4.0.1"
203          packaging = "jar"
204        }
205      }
206    }
207  }
208
209  @dependencies {
210    @dependencies {
211      gradle_core {
212        artifact {
213          name = "gradle-core"
214          version = "4.0.1"
215          packaging = "jar"
216        }
217      }
218    }
219  }
220
221  @dependencies {
222    @dependencies {
223      gradle_core {
224        artifact {
225          name = "gradle-core"
226          version = "4.0.1"
227          packaging = "jar"
228        }
229      }
230    }
231  }
232
233  @dependencies {
234    @dependencies {
235      gradle_core {
236        artifact {
237          name = "gradle-core"
238          version = "4.0.1"
239          packaging = "jar"
240        }
241      }
242    }
243  }
244
245  @dependencies {
246    @dependencies {
247      gradle_core {
248        artifact {
249          name = "gradle-core"
250          version = "4.0.1"
251          packaging = "jar"
252        }
253      }
254    }
255  }
256
257  @dependencies {
258    @dependencies {
259      gradle_core {
260        artifact {
261          name = "gradle-core"
262          version = "4.0.1"
263          packaging = "jar"
264        }
265      }
266    }
267  }
268
269  @dependencies {
270    @dependencies {
271      gradle_core {
272        artifact {
273          name = "gradle-core"
274          version = "4.0.1"
275          packaging = "jar"
276        }
277      }
278    }
279  }
280
281  @dependencies {
282    @dependencies {
283      gradle_core {
284        artifact {
285          name = "gradle-core"
286          version = "4.0.1"
287          packaging = "jar"
288        }
289      }
290    }
291  }
292
293  @dependencies {
294    @dependencies {
295      gradle_core {
296        artifact {
297          name = "gradle-core"
298          version = "4.0.1"
299          packaging = "jar"
300        }
301      }
302    }
303  }
304
305  @dependencies {
306    @dependencies {
307      gradle_core {
308        artifact {
309          name = "gradle-core"
310          version = "4.0.1"
311          packaging = "jar"
312        }
313      }
314    }
315  }
316
317  @dependencies {
318    @dependencies {
319      gradle_core {
320        artifact {
321          name = "gradle-core"
322          version = "4.0.1"
323          packaging = "jar"
324        }
325      }
326    }
327  }
328
329  @dependencies {
330    @dependencies {
331      gradle_core {
332        artifact {
333          name = "gradle-core"
334          version = "4.0.1"
335          packaging = "jar"
336        }
337      }
338    }
339  }
340
341  @dependencies {
342    @dependencies {
343      gradle_core {
344        artifact {
345          name = "gradle-core"
346          version = "4.0.1"
347          packaging = "jar"
348        }
349      }
350    }
351  }
352
353  @dependencies {
354    @dependencies {
355      gradle_core {
356        artifact {
357          name = "gradle-core"
358          version = "4.0.1"
359          packaging = "jar"
360        }
361      }
362    }
363  }
364
365  @dependencies {
366    @dependencies {
367      gradle_core {
368        artifact {
369          name = "gradle-core"
370          version = "4.0.1"
371          packaging = "jar"
372        }
373      }
374    }
375  }
376
377  @dependencies {
378    @dependencies {
379      gradle_core {
380        artifact {
381          name = "gradle-core"
382          version = "4.0.1"
383          packaging = "jar"
384        }
385      }
386    }
387  }
388
389  @dependencies {
390    @dependencies {
391      gradle_core {
392        artifact {
393          name = "gradle-core"
394          version = "4.0.1"
395          packaging = "jar"
396        }
397      }
398    }
399  }
400
401  @dependencies {
402    @dependencies {
403      gradle_core {
404        artifact {
405          name = "gradle-core"
406          version = "4.0.1"
407          packaging = "jar"
408        }
409      }
410    }
411  }
412
413  @dependencies {
414    @dependencies {
415      gradle_core {
416        artifact {
417          name = "gradle-core"
418          version = "4.0.1"
419          packaging = "jar"
420        }
421      }
422    }
423  }
424
425  @dependencies {
426    @dependencies {
427      gradle_core {
428        artifact {
429          name = "gradle-core"
430          version = "4.0.1"
431          packaging = "jar"
432        }
433      }
434    }
435  }
436
437  @dependencies {
438    @dependencies {
439      gradle_core {
440        artifact {
441          name = "gradle-core"
442          version = "4.0.1"
443          packaging = "jar"
444        }
445      }
446    }
447  }
448
449  @dependencies {
450    @dependencies {
451      gradle_core {
452        artifact {
453          name = "gradle-core"
454          version = "4.0.1"
455          packaging = "jar"
456        }
457      }
458    }
459  }
460
461  @dependencies {
462    @dependencies {
463      gradle_core {
464        artifact {
465          name = "gradle-core"
466          version = "4.0.1"
467          packaging = "jar"
468        }
469      }
470    }
471  }
472
473  @dependencies {
474    @dependencies {
475      gradle_core {
476        artifact {
477          name = "gradle-core"
478          version = "4.0.1"
479          packaging = "jar"
480        }
481      }
482    }
483  }
484
485  @dependencies {
486    @dependencies {
487      gradle_core {
488        artifact {
489          name = "gradle-core"
490          version = "4.0.1"
491          packaging = "jar"
492        }
493      }
494    }
495  }
496
497  @dependencies {
498    @dependencies {
499      gradle_core {
500        artifact {
501          name = "gradle-core"
502          version = "4.0.1"
503          packaging = "jar"
504        }
505      }
50
```

We compile and build the project. The maven dependencies are added once the maven build is completed. This is shown below:

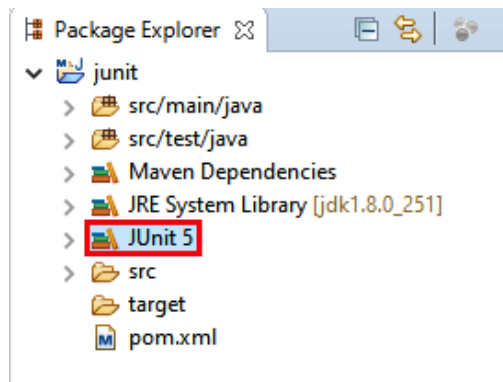


We also add the JUnit5 library to the build path as follows:





The JUnit5 library is added successfully to the project as shown below:



Now, test cases using JUnit 5 can be written and executed. A maven project is the easiest way of adding all the required dependencies of JUnit to your project.

Now that we have seen how to setup JUnit in Eclipse, we will see how to write simple test cases using JUnit in the next section.

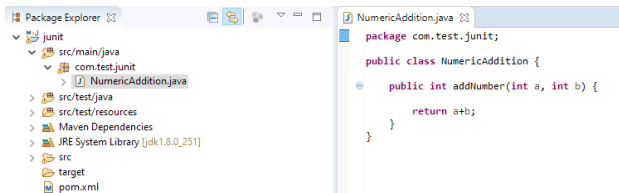
An important point to consider is the version of Eclipse that is being used. The older versions of eclipse do not support JUnit5. Hence, versions of Eclipse equal or higher to the Oxygen 1a must be used in order to execute test cases that are written in JUnit5. Even though JUnit5 is not supported in the earlier versions of Eclipse, tests written in JUnit5 can be run on a JUnit4 runner. We shall see a few examples of this in the next section.

6.1.2. JUnit Examples

- Example 1: Simple JUnit Test

In this example, we see the use of the ‘Test’ annotation in JUnit. We will be using Eclipse Oxygen version 1a as it provides support for JUnit test cases.

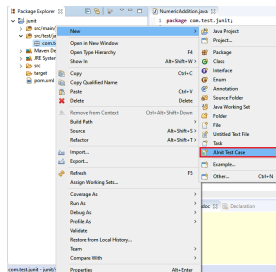
We take a simple java class containing a single method that performs addition of two integers as shown below:



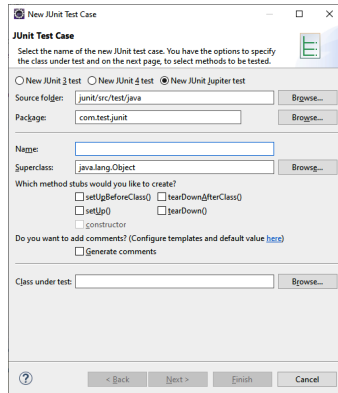
The class is created under the src/main/java folder and is named NumericAddition.

Now, we proceed to write a simple test case for the method addNumber() present in this class in the src/test/java folder.

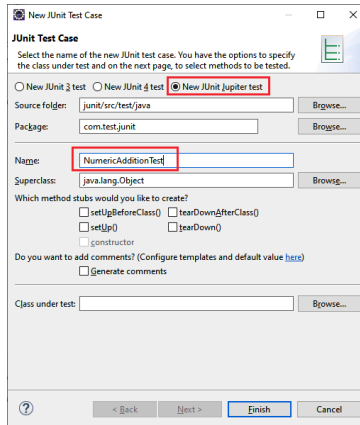
We right click on the package in the src/test/java folder and navigate to the new tab and select ‘JUnit Test Case.’



A popup is displayed.

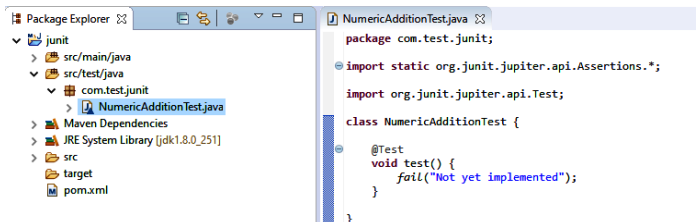


We provide the name of the test case and choose the type of test case to be ‘New JUnit Jupiter test’ as shown below:



Once we provide the details, we click on finish. A new class has been added to our package.

A default implementation is provided. This is shown below:



We modify this default implementation to test our addNumber() method from the class NumericAddition as shown below:

```

NumericAdditionTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class NumericAdditionTest {

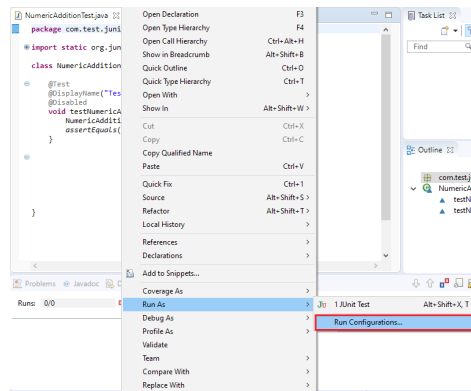
    @Test
    void testNumericAddition() {
        NumericAddition adder = new NumericAddition();
        assertEquals(5, adder.addNumber(2, 3));
    }
}

```

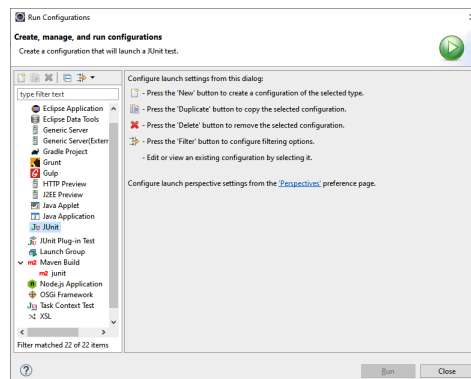
As we can see, a simple method with Test annotation has been created to test the addNumber() method. We make use of the Assertions class to use the assertion method assertEquals().

In order to execute the test case we perform the following steps:

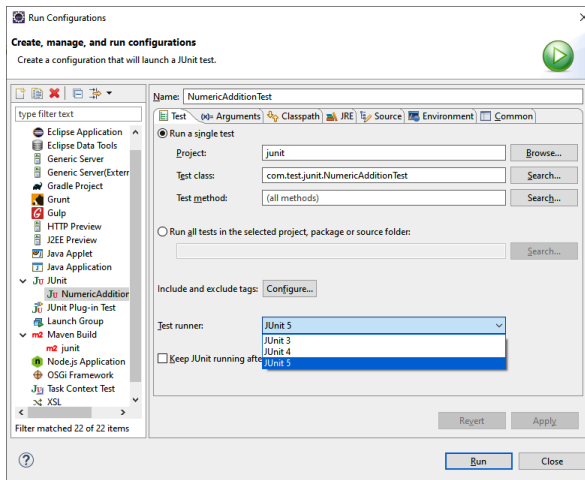
We right click on the class or navigate to the toolbar and click on Run Configurations.



A popup comes up. We double click on JUnit.



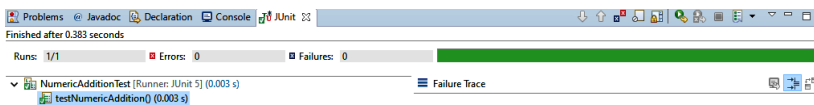
A new configuration creation page is opened.



We provide the name of the test class, the test methods, and the JUnit runner. In our case, we select JUnit5 and click on Apply followed by run.

Another way to execute the test case is to right click on the class and click on Run As -> JUnit. Although this method is easier, we should always ensure that the correct runner has been selected for the execution of the test class.

Now that a new configuration has been created successfully, we execute the test class. The results obtained are as follows:



As seen from the results, the test case has been executed with success.

- Example 2: Test Case With the DisplayName Annotation

As seen in the previous example, on execution of the test case, the method name is shown in the results section. In order to give the test case a more meaningful name, we can make use of the 'DisplayName' annotation. This annotation can be used to give sensible titles to your test cases.

In this example, we explore the 'DisplayName' annotation which is used to give a custom name to a test case or test class.

We consider the same test class from the previous example.

We add the annotation DisplayName to the method with a customized name. In addition, we have added another test method that tests the numeric addition of two integers with a custom display name.

This is shown below:

```

NumericAdditionTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class NumericAdditionTest {

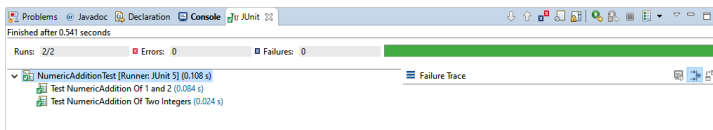
    @Test
    @DisplayName("Test NumericAddition Of Two Integers")
    void testNumericAddition() {
        NumericAddition adder = new NumericAddition();
        assertEquals(5, adder.addNumber(2, 3));
    }

    @Test
    @DisplayName("Test NumericAddition Of 1 and 2")
    void testNumericAdditionWithMessage() {
        NumericAddition adder = new NumericAddition();
        assertEquals(3, adder.addNumber(1, 2));
    }
}

```

As seen from the screenshot, we have two test methods that have been annotated with the `DisplayName` annotation with a custom display message for each test method.

On execution, the results obtained are as follows:



As observed in the results, the display names of the test cases are the same as specified using the `DisplayName` annotation.

- Example 3: Parameterized Test Cases

In this example, we shall study the use of the `ParameterizedTest` annotation. A test that is parameterized allows the test to be executed multiple times with different input parameters or arguments.

As a means to program parameterized test cases and classes in our code, the `JUnit-jupiter-params` dependency must be added to our project.

We update our `pom.xml` file to add this dependency as shown below:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <groupId>com.test</groupId>
    <artifactId>junit</artifactId>
    <version>5.8.0</version>
    <packaging>jar</packaging>
    <name>junit</name>
    <url>http://maven.apache.org/</url>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.8.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
            <version>5.8.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-commons</artifactId>
            <version>1.8.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-engine</artifactId>
            <version>1.8.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-runner</artifactId>
            <version>1.8.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>

```

Once the dependency is added in the pom.xml file, we re-build the project so that the required files are downloaded by maven.

Once the project has been created, we proceed to create a new test class named `NumericAdditionParameterizedTest.java` under the `src/test/java` folder.

The content of the test class is as shown below:

```

NumericAdditionParameterizedTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class NumericAdditionParameterizedTest {

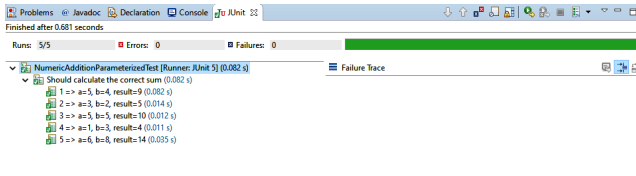
    @DisplayName("Should calculate the correct sum")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, result={2}")
    @CsvSource({
        "5, 4, 9",
        "3, 2, 5",
        "5, 5, 10",
        "1, 3, 4",
        "6, 8, 14"
    })
    void testAddition(int a, int b, int result) {
        NumericAddition adder = new NumericAddition();
        assertEquals(result, adder.addNumber(a, b));
    }
}

```

In the test class, we have created a simple test case and annotated it with the `ParameterizedTest` annotation. Within this annotation, we pass a template for setting the way in which we want the results to be displayed. The `DisplayName` annotation is used to give a custom name to the test method whereas in the `ParameterizedTest` annotation, the `name` attribute is used to specify a custom way of displaying the input arguments and the results during each invocation of the method.

The third and most crucial annotation that is being used in this test class is the `CsvSource` annotation. This annotation is used to provide a source of arguments or inputs to a parameterized test method. A parameterized test requires an argument source and `CsvSource` is one of the possible input or argument sources that are available for parameterized tests. The `CsvSource` annotation is used to provide an argument list in the form of simple comma separated values. String literals in the form of values that comma separated (or separated by other types of delimiters) are permitted to be passed using this annotation. By default, the comma separator is used. To make use of other delimiters the `'delimiter'` attribute.

We execute the test case. The results are:

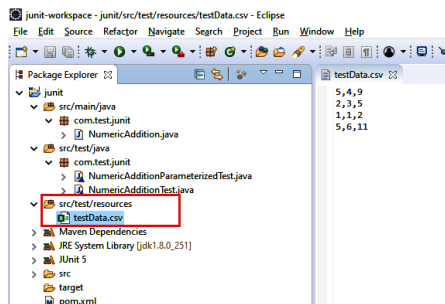


As can be observed from the results, the test case is executed 5 times with different inputs that have been provided using the CSVSource annotation. As we can see, the display name is the same one that was provided in the test method. The test case adds the first and second argument using the method `addNumber()` and compares it with the expected result which is provided in the input source list. The result is displayed in a clear and concise manner with an index, followed by the two inputs, the result, and whether the test case passed or failed. The results are displayed using the template specified by the name attribute of the `ParameterizedTest` annotation.

The input arguments can be passed to a parameterized test method using a csv file as well. This can be done with the help of the `CSVFileSource` annotation. This annotation permits the use of a CSV file present in the classpath to pass the arguments to the test method. Every line that is present in the CSV file leads to an invocation of the parameterized test method. Such a way of testing is important when the required inputs are requested from a stakeholder who is not physically writing the test cases but wishes to test a particular set of data. Also, such a way of testing helps externalize the inputs so that they can be changed with no impact on the test suite.

To make use of this annotation, we first create a CSV file named `testData.csv` and add it to the project's classpath under the `src/test/resources` folder.

This is shown below:



We then proceed to use the `CSVFileSource` annotation instead of `CSVSource` in our test class. This is shown below:

```

NumericAdditionParameterizedTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

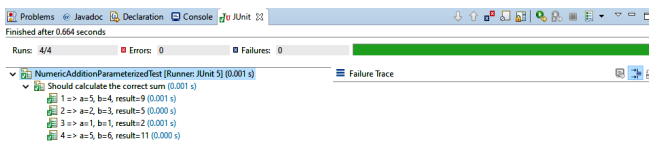
class NumericAdditionParameterizedTest {

    @DisplayName("Should calculate the correct sum")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, result={2}")
    @CsvFileSource(resources = "/testData.csv")
    void testAddition(int a, int b, int result) {
        NumericAddition adder = new NumericAddition();
        assertEquals(result, adder.addNumber(a, b));
    }
}

```

As seen in the class, the `CsvFileSource` annotation is used with the `resources` attribute which provides the name and path of the csv file that is to be used as an input source.

We execute the test class and obtain the results as shown below:



It is clear from the test results that the data provided in the csv file has been used as input for the test case.

If the csv file has a header, the `numLinesToSkip` attribute of the `CsvFileSource` annotation can be used.

The data file with a header is as follows:

```

testData.csv

a, b, result
5, 4, 9
2, 3, 5
1, 1, 2
5, 6, 11

```

The test class has been updated as follows:

```

NumericAdditionParameterizedTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

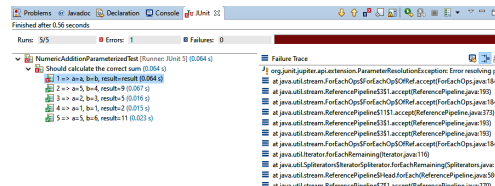
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

class NumericAdditionParameterizedTest {

    @DisplayName("Should calculate the correct sum")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, result={2}")
    @CsvFileSource(resources = "/testData.csv", numLinesToSkip = 1)
    void testAddition(int a, int b, int result) {
        NumericAddition adder = new NumericAddition();
        assertEquals(result, adder.addNumber(a, b));
    }
}

```

On execution, we receive the following results:



As demonstrated by the failure in the results, the header has not been ignored. This because we are using a version of Eclipse that is not the latest. JUnit5 support is better in the recent versions.

If we execute the same test in Eclipse Oxygen 3a, the test runs successfully. As covered, multiple parameters can be passed to the test method by using a csv file. But, there is a constraint when using this annotation. There is a restriction on the type of the parameters that are passed via this annotation. The limitation is that the arguments that are passed to the method using this annotation need to be supported by the class `DefaultArgumentConverter` whose official Javadoc asserts that this class supports conversion of strings to a small set of classes/types such as primitive types, the wrappers of primitive types (Double, Byte, Short, Integer, Long, and Float) and the date/time types.

This roadblock can be easily overcome by use of a custom or factory method. Complex data types can be passed by using factory methods. We shall see how this can be done in the next example.

- Example 4: Parameterized Tests Using Factory Methods

In this example, we shall look at the use of factory methods using the `MethodSource` annotation for parameterized tests that is another input source provider used to provide input arguments to parameterized tests. It allows us to access one or more factory methods of the test class or external classes as a source of input arguments for a parameterized test (JUnit 5, 2020). There are few conditions to using factory methods to pass parameters to test cases or methods. They are:

- The factory method cannot have any arguments in its signature.
- The factory method used should return objects of type that are either a Stream, Iterable, Iterator or an array of objects of type Arguments.
- The factory method must be static.
- The arguments passed in this method must contain all the input parameters need for the execution of the test method.

The arguments are passed by means of using the static method of() from the Stream type. The parameters are passed to this method.

This annotation MethodSource is used to denote that a factory method will be used to provide the arguments. This annotation is simple to use. The name of the factory method or class is provided in quotes to the annotation as shown in the class below:

```

NumericAdditionFactoryMethodProviderTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;
import java.util.stream.Stream;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

class NumericAdditionFactoryMethodProviderTest {

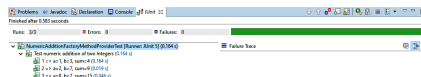
    @DisplayName("Test numeric addition of two integers")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
    @MethodSource("testDataProvider")
    void sum(int a, int b, int result) {
        NumericAddition adder = new NumericAddition();
        assertEquals(result, adder.addNumber(a, b));
    }

    private static Stream<Arguments> testDataProvider() {
        return Stream.of(
            Arguments.of(1, 3, 4),
            Arguments.of(2, 7, 9),
            Arguments.of(8, 7, 15)
        );
    }
}

```

In the test class, we have added a method called testDataProvider that returns a stream of data that is passed as an argument to the test method. The MethodSource annotation carries the name of the factory method that provides the input data.

We execute the test case and obtain the results as follows:



As visible in the results, the data provided in the factory method has been used as an input for the test method.

A factory method is useful when all the test methods belong to the same test class and when all the methods using the factory method are in the same class. But, if this is not the case, and if the argument provider is required across classes or package, a custom argument provider is more helpful

which is the focus of the next example.

- Example 5: Custom Argument Provider

When the test methods using the test data belong to different test classes or packages, or when the test data implementation logic is highly complex and needs to be implemented outside the test class, a custom argument provider class can be used instead.

A custom class that provides arguments can be easily implemented by means of the `ArgumentProvider` interface of JUnit. The class needs to implement this interface and provide the logic for generation of the test data. This interface comes with a `provideArguments()` abstract method whose implementation needs to be provided. The signature of this method is such that a `Stream` object of the type `Arguments` must be returned. After we have created this class, we have to implement the `provideArguments()` method that returns a `Stream` of `Arguments` objects. Similar to a factory argument provider method, the object of type `Arguments` must provide all the parameters required for the test method and the class must be static and make use of the `Stream.of()` method to provide the arguments.

A simple test class implementing a custom argument provider class is shown below:

```

NumericAdditionParameterizedArgumentSourceTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.assertEquals;
import java.util.stream.Stream;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.ArgumentsProvider;
import org.junit.jupiter.params.provider.ArgumentsSource;

class NumericAdditionParameterizedArgumentSourceTest {

    @DisplayName( "Numeric Addition Test with @ArgumentsSource" )
    @ParameterizedTest( name = "{index}: ({0} + {1}) => {2})" )
    @ArgumentsSource( ParameterProvider.class )
    void test( int firstInput, int secondInput, int result )
    {
        NumericAddition adder = new NumericAddition();
        assertEquals(result, adder.addNumber(firstInput,secondInput));
    }

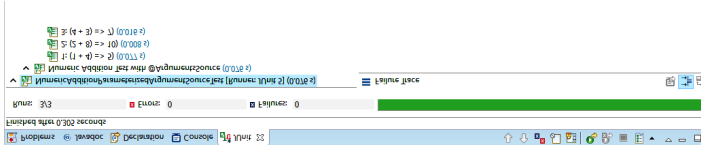
    static class ParameterProvider implements ArgumentsProvider
    {
        @Override
        public Stream< ? extends Arguments > provideArguments( ExtensionContext context )
        {
            return Stream.of(
                Arguments.of( 1, 4, 5 ),
                Arguments.of( 2, 8, 10 ),
                Arguments.of( 4, 3, 7 )
            );
        }
    }
}

```

As visible in the class, we have created a new static class called `ParameterProvider` that implements the `ArgumentsProvider` interface and implements the `provideArguments()` method which returns a stream of input arguments meant for use by a parameterized test method.

We pass the name of the custom class using the `ArgumentSource` annotation which accepts a class type.

On execution of the test class, we obtain the following results:



As seen from the previous examples, a test can be invoked several times with the help of parameterized tests. In the next example, we shall see how tests can be repeated.

- Example 6: Repeated Tests

JUnit5 provides the functionality to execute a test for a specific number of times. This type of testing is useful in case of testing the stress the system can handle and the performance of the system under certain conditions as well. This can be done by use of the `RepeatedTest` annotation.

This annotation is used at the method level and supports the following placeholders that can be passed or set with the annotation (JUnit 5, 2020):

{displayName}-the display name or custom name of the `@RepeatedTest` test method

{currentRepetition}-the current repetition count

{totalRepetitions}-the total number of repetitions of the test method

We make use of the `RepeatedTest` annotation to test our method from the class `NumericAddition` as shown below:

```

1 RepeatedTests.java
2 package com.test.junit5;
3
4 import static org.junit.jupiter.api.Assertions.*;
5
6 import org.junit.jupiter.api.DisplayName;
7 import org.junit.jupiter.api.RepeatedTest;
8
9 class RepeatedTests {
10
11     @RepeatedTest(value = 4, name = RepeatedTest.LONG_DISPLAY_NAME)
12     @DisplayName("Test NumericAdditionOfTwoIntegers")
13     void testNumericAddition() {
14         NumericAddition adder = new NumericAddition();
15         assertEquals(5, adder.addNumber(2, 3));
16     }
17
18     @RepeatedTest(value = 2, name = RepeatedTest.SHORT_DISPLAY_NAME)
19     @DisplayName("Result of numeric addition of two integers must be >= 0")
20     public void testAdditionNotNegative() {
21         NumericAddition adder = new NumericAddition();
22         int result = adder.addNumber(3, 7);
23         assertTrue(result >= 0);
24     }
25 }

```

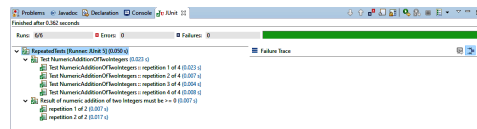
We have created a new test class with two test methods bearing the `RepeatedTest` annotation. One method tests the `addNumber()` method of the `NumericAddition` class and the second test method tests if the result of the addition is a positive integer. We pass a value that denotes the number of

repetitions as an argument to the annotation declaration and also fill in the name attribute. The name attribute is used to define the pattern that will be used to display the results of each invocation of the test method. In the above class, two patterns (static) that are defined in the RepeatedTest class have been used.

They are:

1. **LONG_DISPLAY_NAME:** It is a display name pattern that is long in nature and gives extra information. The results are displayed in the following pattern-`{displayName}::repetition {currentRepetition} of {totalRepetitions}`”
2. **SHORT_DISPLAY_NAME:** It is a display name pattern that is small and concise in nature and gives the required information regarding the test method invocation. The results are displayed in the following pattern-`repetition {currentRepetition} of {totalRepetitions}`”

We execute the test case and obtain the results as shown below:



As visible from the results, the tests have been repeated 4 times and two times respectively, as stated by means of the value parameter. We can also observe the way in which each of invocations is displayed using the pattern specified by the name attribute. As we can see, the display name for the second test case has been shortened and simplified using a customization in the name parameter. In the first case, a different way of displaying the results is used which accounts for the longer title but gives the same information.

The display name that we have passed using the name parameter can be customized as well. We can make use of the `displayName`, `currentRepetition`, and `totalRepetitions` placeholders to provide a display pattern that is customized to our needs. This is shown below:

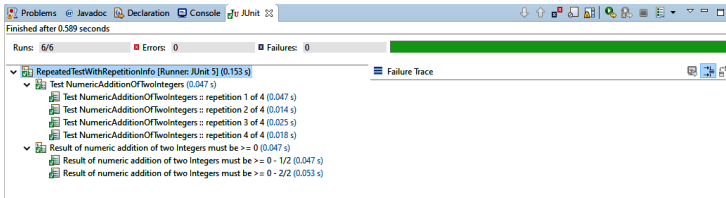
```

1 RepeatedTest.java
2
3 import org.junit.jupiter.api.DisplayName;
4 import org.junit.jupiter.api.RepeatedTest;
5 import static org.junit.jupiter.api.Assertions.*;
6
7 class RepeatedTests {
8     @RepeatedTest(value = 4, name = RepeatedTest.LONG_DISPLAY_NAME)
9     @DisplayName("Test Numeric Addition Of Two Integers")
10    void testNumericAddition() {
11        NumericAddition adder = new NumericAddition();
12        assertEquals(5, adder.addNumber(2, 3));
13    }
14
15    @RepeatedTest(value = 2, name = "{displayName} - {currentRepetition}/{totalRepetitions}")
16    @DisplayName("Result of numeric addition of two integers must be >= 0")
17    public void testAdditionNotNegative() {
18        NumericAddition adder = new NumericAddition();
19        int result = adder.addNumber(3, 7);
20        assertTrue(result >= 0);
21    }
22 }

```

We have updated one of the test methods to have a customized display pattern.

The results of execution are:



As seen in the results, the modified pattern is used to display the results of execution of the test cases. Another important detail that is available to us in the JUnit framework is the ability to generate and extract information about each repetition. This is done with the help of an interface called `RepetitionInfo`. This interface can be used to get information about the current repetition and the total repetitions for the test method that is repeated (`RepetitionInfo (JUnit 5.0.0 API,`” 2017).

This interface must be passed as a parameter to the method whose repetition information is needed. This parameter is used for methods annotated with one of the following annotations: `@RepeatedTest`, `@BeforeEach` and `@AfterEach`. If this parameter is given, JUnit will provide an instance of the interface `RepetitionInfo` that corresponds to the current repeated test. It should be noted that the argument of type `RepetitionInfo` can only be passed to test methods that are repeated. If the test is not repeated, a `ParameterResolutionException` is thrown at runtime.

`RepetitionInfo` is useful when additional data regarding the tests is needed. This interface is especially useful when debugging of issues is needed and tests data can help uncover bugs or defects.

A simple use of `RepetitionInfo` is shown below:

```

1 package com.test.junit;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 import org.junit.jupiter.api.DisplayName;
6 import org.junit.jupiter.api.RepeatedTest;
7 import org.junit.jupiter.api.RepetitionInfo;
8
9 class RepeatedTestWithRepetitionInfo {
10
11     @RepeatedTest(value = 4, name = RepeatedTest.LONG_DISPLAY_NAME)
12     @DisplayName("Test NumericAdditionOfTwoIntegers")
13     void testNumericAddition(RepetitionInfo repetitionInfo) {
14         NumericAddition adder = new NumericAddition();
15         int currentRepetition = repetitionInfo.getCurrentRepetition();
16         int totalRepetitions = repetitionInfo.getTotalRepetitions();
17         System.out.println(String.format("Executing repetition %d of %d for %s", currentRepetition, totalRepetitions,
18             "testNumericAddition()"));
19         assertEquals(5, adder.addNumber(2, 3));
20     }
21
22     @RepeatedTest(value = 2, name = "[displayName] - {currentRepetition}/{totalRepetitions}")
23     @DisplayName("Result of numeric addition of two integers must be >= 0")
24     public void testAdditionWithNegative(RepetitionInfo repetitionInfo) {
25         NumericAddition adder = new NumericAddition();
26         int result = adder.addNumber(1, 7);
27         int currentRepetition = repetitionInfo.getCurrentRepetition();
28         int totalRepetitions = repetitionInfo.getTotalRepetitions();
29         System.out.println(String.format("Executing repetition %d of %d for %s", currentRepetition, totalRepetitions,
30             "testAdditionWithNegative()"));
31         assertTrue(result >= 0);
32     }
33 }

```

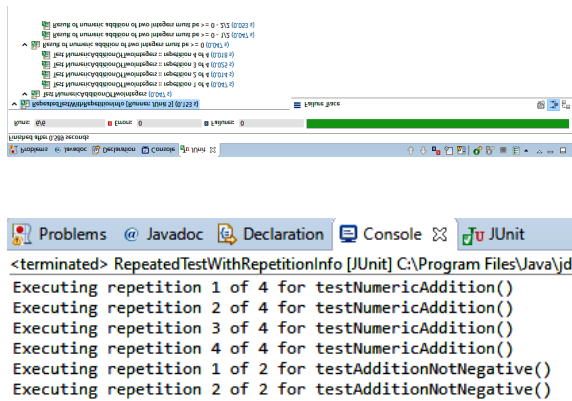
We have created a new test class which has two tests that are repeated and have the parameter `RepetitionInfo` that is present in its definition. This parameter is used to obtain information about the current invocation/

repetition of the method and print it out to the console. The test method that accepts a parameter of type `RepetitionInfo` which is initialized and supplied by the JUnit framework that corresponds to the test under consideration.

This information can also be directed to a log file which can immensely help in the process of debugging.

On execution, we now have the results in two different tabs, one is the resulting JUnit execution of the test case, and the second one is the console output coming from the print statement in the method.

They are as follows:



The screenshot shows an IDE with two tabs open: 'JUnit' and 'Console'. The JUnit tab displays the results of a test run, showing that the test passed. The Console tab shows the output of the test, which includes the following text:

```
<terminated> RepeatedTestWithRepetitionInfo [JUnit] C:\Program Files\Java\jdk
Executing repetition 1 of 4 for testNumericAddition()
Executing repetition 2 of 4 for testNumericAddition()
Executing repetition 3 of 4 for testNumericAddition()
Executing repetition 4 of 4 for testNumericAddition()
Executing repetition 1 of 2 for testAdditionNotNegative()
Executing repetition 2 of 2 for testAdditionNotNegative()
```

As seen from the results on the console, information about each repetition is now available.

In the test class, we have added the information in every method which is not feasible and makes the code messy and redundant. To improve this, we can add the data about the repetitions in a single method that will be executed before or after each of the test methods. This can easily be done using the `BeforeEach` or `AfterEach` annotation.

This is done as follows:

```
1 RepeatedTestWithRepetitionInfo.java 11
package com.test.junit;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.RepetitionInfo;
import org.junit.jupiter.api.TestInfo;

class RepeatedTestWithRepetitionInfo {

    @BeforeEach
    void beforeEach(TestInfo testInfo, RepetitionInfo repetitionInfo) {
        let currentRepetition = repetitionInfo.getCurrentRepetition();
        let totalRepetitions = repetitionInfo.getTotalRepetitions();
        String methodName = testInfo.getTestMethod().getName();

        System.out.println(String.format("Executing repetition %d of %d for %s", currentRepetition, totalRepetitions,
            methodName));
    }

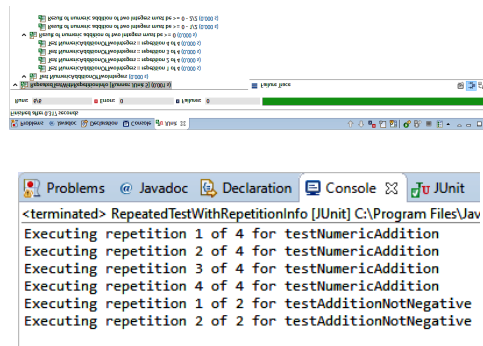
    @RepeatedTest(value = 4, name = "RepeatedTest.LONG_DISPLAY_NAME")
    @DisplayName("Test NumericAdditionNotInteger")
    void testNumericAdditionNotInteger(RepetitionInfo repetitionInfo) {
        NumericAddition adder = new NumericAddition();
        assertEquals(5, adder.addNumber(2, 3));
    }

    @RepeatedTest(value = 2, name = "TestAdditionNotNegative")
    @DisplayName("Test AdditionNotNegative")
    public void testAdditionNotNegative(RepetitionInfo repetitionInfo) {
        NumericAddition adder = new NumericAddition();
        let result = adder.addNumber(5, 7);
        assertEquals(12, result);
    }
}
```

The test class has now been updated to include a new method that is annotated with the `BeforeEach` annotation. The `RepetitionInfo` argument is passed as a parameter to this method. This is used to extract the data regarding the current and total repetitions of the currently executing repeated test method as seen in the previous example. Another important parameter that is passed is the `TestInfo` parameter. The `TestInfo` interface is another useful interface which is a part of the JUnit framework and it provides information regarding the current test or the current container to a method that is annotated with one of the following annotations: `@Test`, `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll`.

In this example, we use the `TestInfo` argument to get information about the current test case that is being executed.

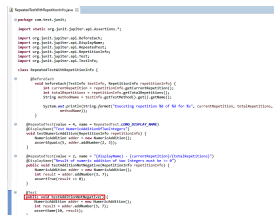
The results of execution are as follows:



```
<terminated> RepeatedTestWithRepetitionInfo [JUnit] C:\Program Files\Java
Executing repetition 1 of 4 for testNumericAddition
Executing repetition 2 of 4 for testNumericAddition
Executing repetition 3 of 4 for testNumericAddition
Executing repetition 4 of 4 for testNumericAddition
Executing repetition 1 of 2 for testAdditionNotNegative
Executing repetition 2 of 2 for testAdditionNotNegative
```

As one can observe, the results have not changed from the previous example. The idea behind the use of the `BeforeEach` method was to reduce repetitive code and make the code more concise and presentable, thereby making it easier to understand.

Now that we have seen the use of `RepetitionInfo`, let us see what happens when a method that is not annotated with the `RepeatedTest` annotation tries to use this information.



```
import org.junit.Test;
import org.junit.Assert;

import static org.junit.Assert.*;

public class RepeatedTestWithRepetitionInfo {

    @Test
    public void testNumericAddition() {
        // ...
    }

    @Test
    public void testAdditionNotNegative() {
        // ...
    }
}
```

The class has been updated and a new method annotated with the `Test` annotation has been added to the test class. On execution, we get the following results:

```

<terminated> RepeatedTestWithRepetitionInfo [JUnit] C:\Program Files\Java\j
Executing repetition 1 of 4 for testNumericAddition
Executing repetition 2 of 4 for testNumericAddition
Executing repetition 3 of 4 for testNumericAddition
Executing repetition 4 of 4 for testNumericAddition
Executing repetition 1 of 2 for testAdditionNotNegative
Executing repetition 2 of 2 for testAdditionNotNegative

Finished after 0.327 seconds
Runs: 7/7 Errors: 1 Failures: 0

Failure Trace
org.junit.jupiter.api.extension.ParameterResolutionException: No ParameterResolver registered for param
info.org.junit.jupiter.api.RepetitionInfo).
at java.util.stream.ForEachOps$ForEachOp$Offset.accept(ForEachOps.java:184)
at java.util.stream.ReferencePipeline$S251.accept(ReferencePipeline.java:175)
at java.util.Iterator.forEachRemaining(Iterator.java:116)
at java.util.Spliterators$IteratorSpliterator.forEachRemaining(Spliterators.java:1801)
at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:482)
at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:472)
at java.util.stream.ForEachOps$ForEachOp.evaluateSequential(ForEachOps.java:151)
at java.util.stream.ForEachOps$ForEachOp$Offset.evaluateSequential(ForEachOps.java:174)

```

The console output remains the same, but the JUnit output reports an error. The newly added test method `testAdditionNotNegative()` fails and throws a `ParameterResolutionException` as expected.

- Example 7: Miscellaneous Argument Sources

In this example, we look at additional types of argument sources that can be used for parameterized test cases. We consider the following argument sources:

1. **NullSource:** This annotation is used to provide a solitary null argument to a test method annotated with the `ParameterizedTest` annotation (JUnit 5, 2020).
2. **EmptySource:** This annotation is used to provide a solitary empty argument to a test method annotated with the `ParameterizedTest` annotation. This annotation supports and provides an argument that is empty for the following data types: `java.lang.String`, `java.util.List`, `java.util.Set`, `java.util.Map`, primitive type arrays such `int[]`, `char[][]`, and object arrays (for instance, `String[]`, `Integer[][]`, etc.) (JUnit 5, 2020).
3. **NullAndEmptySource:** This annotation is a combination of the `NullSource` annotation and the `EmptySource` annotation and provides one null and one empty argument for a parameterized test method (JUnit 5, 2020).
4. **EnumSource:** It is an annotation that helps make use of enumerations in the test cases (JUnit 5, 2020).

5. **ValueSource:** The ValueSource is an argument provider annotation which is the simplest source provider. This annotation permits us to pass a string array of literal values to the test method. This annotation allows us to provide a single argument only. This means that for each invocation of the parameterized test, only a single value passed using this source provider is used. This annotation supports the following types of values that can be passed to it: int, long, float, double, short, byte, char Boolean, String, and Class (JUnit 5, 2020).

In order to demonstrate the use of these different annotations, we consider the following factory class consisting of a single method that tests a String and returns true if the String contains an integer.

```

NumericStringCheckerFactory.java
package com.test.junit;

public class NumericStringCheckerFactory {

    public static boolean isNumericString(final String str) {

        if(str == null || str.isEmpty()) return false;

        for(char c : str.toCharArray()){
            if(Character.isDigit(c)){
                return true;
            }
        }
        return false;
    }
}

```

We proceed to write a test class that tests the isNumericString() as shown below:

```

MiscellaneousParameterizedArgumentSrcTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EmptySource;
import org.junit.jupiter.params.provider.NullAndEmptySource;
import org.junit.jupiter.params.provider.NullSource;
import org.junit.jupiter.params.provider.ValueSource;

class MiscellaneousParameterizedArgumentSrcTest {

    @Test
    @DisplayName("Test to check if string contains an integer")
    void testIfStringContainsInteger() {
        assertTrue(NumericStringCheckerFactory.isNumericString("Test123"));
    }

    @ParameterizedTest(name="#{index} - Test with input null argument={0}")
    @NullSource
    void testNullSource(String argument) {
        assertTrue(argument == null);
    }

    @ParameterizedTest(name="#{index} - Test with empty argument={0}")
    @EmptySource
    void testEmptySource(String argument) {
        assertTrue(argument.isEmpty());
    }

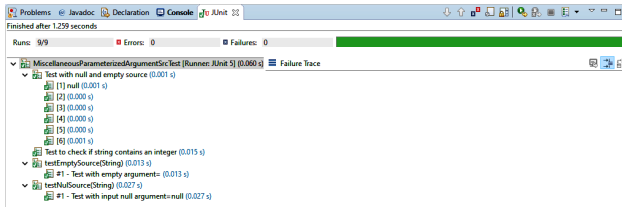
    @ParameterizedTest
    @DisplayName("Test with null and empty source")
    @NullAndEmptySource
    @ValueSource(strings = { " ", " ", "\t", "\n" })
    void testNullAndEmptySource(String argument) {
        assertTrue(argument == null || argument.trim().isEmpty());
    }
}

```

We have written four different test cases using different argument sources. The first test case is a simple non-parameterized test that checks the functionality of the method. It checks whether the method provides the functionality that it is programmed to deliver. The input string that is passed is "Test123" which contains an integer and hence should return true.

The next test case is a parameterized test which passes a null argument by implementing the `NullSource` annotation. The third test case is another parameterized test case which passes an empty argument by means of the `EmptySource` annotation. Lastly, the final test case makes use of the `NullAndEmptySource` annotation and the `ValueSource` annotation to check for null and empty values. The `ValueSource` is used to supply four different types of blank and empty string inputs to the parameterized test.

The results obtained following the execution of the above test class are as follows:



The non-parameterized test passes which confirms that the `isNumericString()` function works properly. The parameterized test cases using the `NullSource` and `EmptySource` also pass which affirms that one null source and one empty source was provided to the respective test methods.

The results of the last test case which implemented the `ValueSource` and the `NullAndEmptySource` annotation are interesting. As we can see, this particular test method was invoked a total of 6 times. This is because it is invoked once for the Null source, once for the Empty source (due to the `NullAndEmptySource` annotation) and four times for the 4 different blank values that were passed using the `ValueSource` annotation.

We consider another example that makes use of the `EnumSource` annotation.

We consider a simple Enumeration class that contains the days of the work week excluding the weekends. The enum is as follows:

```
Weekdays.java
package com.test.junit;

public enum Weekdays {

    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY
}
```

A simple test case to test an enumeration using the `EnumSource` annotation is shown below:


```

EnumSourceParameterizedTest.java 32
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;
import java.util.EnumSet;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EnumSource;

class EnumSourceParameterizedTest {

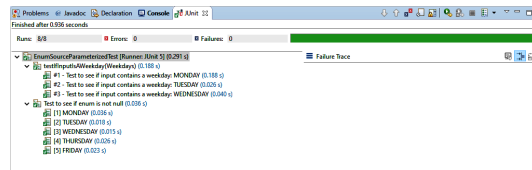
    @ParameterizedTest
    @DisplayName("Test to see if enum is not null")
    @EnumSource(Weekdays.class)
    void testEnumNotNull(Weekdays weekdays) {
        assertNotNull(weekdays);
    }

    @ParameterizedTest(name = "{0}[index] - Test to see if input contains a weekday: {0}")
    @EnumSource(value = Weekdays.class, names = {"MONDAY", "TUESDAY", "WEDNESDAY"})
    void testIfInputIsAWeekday(Weekdays weekdays) {
        assertTrue(EnumSet.allOf(weekdays.class).contains(weekdays));
    }
}

```

We have generated a new test class with two parameterized tests that make use of the EnumSource as an input source provider. The EnumSource annotation accepts the type class which must be an enumeration. The test method testEnumNotNull() checks if Enumeration passed as a parameter does not contain null values. The test method testIfInputIsAWeekday() is programmed to check whether the values passed using the EnumSource annotation belong to the enum group Weekdays.

On execution of this test class, the results are as follows:



As we can observe from the results of the execution, for the method testEnumNotNull() there are five invocations as five members are present in the enumeration Weekdays. For the method testIfInputIsAWeekday() there are three invocations of this test method as three input values have been provided using the EnumSource annotation. The EnumSource annotation is similar to the ValueSource annotation as it permits only one argument to be passed to the test method per invocation.

One detail to be observed from the results is that the tests are executed in a random order and not the order in which they are written. But, JUnit5 provides ways to define the order of execution of test cases. The test cases written in JUnit can be programmed to be executed in a certain sequence which will be discussed in the next example.

- Example 8: Ordered Tests

In this example, we study different functionalities provided by the JUnit framework that help in ordering the execution of the tests.

With the intent to control and pre-define the order of execution of the tests, the TestMethodOrder annotation has been provided by JUnit.

TestMethodOrder is an interface provided by JUnit5 which is used to configure a type called MethodOrderer. The TestMethodOrder annotation accepts the value for the parameter MethodOrderer which is an interface that defines the API for ordering the test methods belonging to a particular test class. The TestMethodOrder annotation must be used at class level.

The MethodOrderer interface is applicable to test methods that are annotated with any one of the following annotations: `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, or `@TestTemplate`.

Three nested class implementations are provided and each of these implementations orders or sorts the tests based on some condition. The three in-built implementations that have been provided for the MethodOrderer interface are:

1. **MethodOrderer.Alphanumeric:** This implementation sorts the test methods in an alphanumeric order on the basis of the name of the method. It makes use of the `String.compareTo()` method.
2. **MethodOrderer.OrderAnnotation:** This is an implementation of the MethodOrderer interface that orders the methods in the test class on the basis of the Order annotation. This annotation is used to provide a numeric sequential value to the test methods. The order or sequence is a parameter passed the Order annotation. This annotation is used at the method level.
3. **MethodOrderer.Random:** This implementation orders the test methods in a pseudo-random manner.

To demonstrate the use of the TestMethodOrder, we consider the test class from the previous example that tested miscellaneous argument sources such as `NullSource`, `EmptySource`, `NullAndEmptySource`, and `ValueSource`.

This is the test class:

```

1 MiscellaneousParameterizedArgumentSrcTest.java
2 package com.test.junit5;
3
4 import static org.junit.jupiter.api.Assertions.*;
5
6 import org.junit.jupiter.api.DisplayName;
7 import org.junit.jupiter.api.Test;
8 import org.junit.jupiter.params.ParameterizedTest;
9 import org.junit.jupiter.params.provider.EmptySource;
10 import org.junit.jupiter.params.provider.NullAndEmptySource;
11 import org.junit.jupiter.params.provider.NullSource;
12 import org.junit.jupiter.params.provider.ValueSource;
13
14 class MiscellaneousParameterizedArgumentSrcTest {
15
16     @Test
17     @DisplayName("Test to check if string contains an integer")
18     void testStringContainsInteger() {
19         assertTrue(NumeralStringCheckerFactory.isNumeralString("text123"));
20     }
21
22     @ParameterizedTest(name="{0}index")
23     @Test
24     @NullSource
25     void testNullSource(String argument) {
26         assertTrue(argument == null);
27     }
28
29     @ParameterizedTest(name="{0}index")
30     @Test
31     @EmptySource
32     void testEmptySource(String argument) {
33         assertTrue(argument.isEmpty());
34     }
35
36     @ParameterizedTest
37     @DisplayName("Test with null and empty source")
38     @NullAndEmptySource
39     @ValueSource(strings = { "", " ", "a", "n" })
40     void testNullAndEmptySource(String argument) {
41         assertTrue(argument == null || argument.trim().isEmpty());
42     }
43 }

```

We proceed to update this class and order the methods using the Order annotation as shown below:

```

MiscellaneousParameterizedArgumentSrcTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EmptySource;
import org.junit.jupiter.params.provider.NullAndEmptySource;
import org.junit.jupiter.params.provider.NullSource;
import org.junit.jupiter.params.provider.ValueSource;

@TestMethodOrder(OrderAnnotation.class)
class MiscellaneousParameterizedArgumentSrcTest {

    @Test
    @DisplayName("Test to check if string contains an Integer")
    @Order(1)
    void testIfStringContainsInteger() {
        assertTrue(NumericStringCheckerFactory.isNumericString("Test123"));
    }

    @ParameterizedTest(name="#[index] - Test with input null argument-{0}")
    @NullSource
    @Order(2)
    void testNullSource(String argument) {
        assertTrue(argument == null);
    }

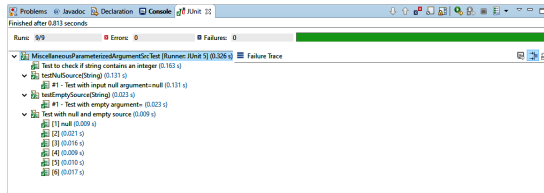
    @ParameterizedTest(name="#[index] - Test with empty argument-{0}")
    @EmptySource
    @Order(3)
    void testEmptySource(String argument) {
        assertTrue(argument.isEmpty());
    }

    @ParameterizedTest
    @DisplayName("Test with null and empty source")
    @NullAndEmptySource
    @ValueSource(strings = { " ", " ", "\t", "\n" })
    @Order(4)
    void testNullAndEmptySource(String argument) {
        assertTrue(argument == null || argument.trim().isEmpty());
    }
}

```

At the class level, we provide the `TestMethodOrder` annotation which specifies the `MethodOrdered` type which is the `OrderAnnotation` in this example. At the method level, we make use of the `Order` annotation to provide the sequence of execution for the test methods.

We execute this test class. The results are:



As we can see from the results, the test methods have been executed in the order that was specified using the Order annotation.

- Example 9: Negative and Boundary Tests

Up until now, all the examples that we have covered, test whether the functionality implemented by a particular method or class has been met or not. The test cases that we have written so far only consist of tests that use values that are accepted by the method that is being tested. This means that the tests consider the range of values for which the functionality is supposed to function. The tests covered so far have respected this boundary. But any test class is incomplete without the values that do not fall under a fixed range. One must always consider the values that are not supposed to be accepted by the class or method and test if the behavior is consistent. The

values that do not form part of the accepted input or the values that are out of bounds must be tested as well. We have not considered a scenario where we expect the test to fail. For instance, if we need to check whether a potential new candidate's age falls between a specific range, we also need to ensure that it does not fall in the unwanted list of values. If we wish to check that the system does not accept all values, we need to test the behavior of the system when forbidden values are provided as an input. So, if the valid range of the age is between 18 and 55, we must have a test case that inputs a value that is below 18 or above 55 to make sure that the application works. Such tests that focus on testing the negative or out of bounds values are also called negative tests. We shall consider some ways of implementing these tests. JUnit5 provides different ways in which such test cases can be written. A couple of helpful methods such as `assertFalse()`, `assertNotSame()` and `assertNotEquals()` can be used to write test cases for values are out of bounds or values that are not permitted.

The `assertFalse()` method allows us to test whether a Boolean value of false is returned by the test method. The `assertNotEquals()` method helps verify if two objects or values are not equal to one another. This method helps write test scenarios where the program's boundary values can be tested.

The `assertNotSame()` method is similar to the `assertNotEquals()` method and helps verify that two objects are not equal to one another.

We consider the `NumericAddition` class that has a simple method `addNumber()` that performs numeric addition of two integers and proceed to write negative test cases for this method.

```

NumericAdditionNegativeTests.java
package com.test.junit5;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class NumericAdditionNegativeTests {

    @Test
    void testInvalidInput() {
        NumericAddition adder = new NumericAddition();
        assertNotEquals(4, adder.addNumber(2, 3));
    }

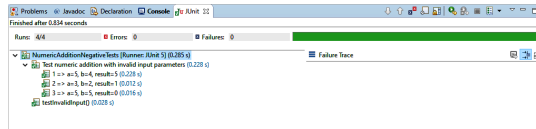
    @DisplayName("Test numeric addition with invalid input parameters")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, result={2}")
    @CsvSource({
        "5, 4, 5",
        "3, 2, 1",
        "5, 5, 0"
    })
    void testAdditionWithInvalidInputs(int a, int b, int result) {
        NumericAddition adder = new NumericAddition();
        assertNotEquals(result, adder.addNumber(a, b));
    }
}

```

We have created a new test class with two test cases—one parameterized and the other non-parameterized. We make use of the `assertNotEquals()` to check whether the result of addition of two values is not equal to incorrect

result provided in the test case. These test cases ensures that the addNumber() method is executing correctly and that the behavior does not change when incorrect data is provided.

The results of the execution are as follows:



As seen from the results, the test cases pass successfully.

The next test class tests a factory method isNumericString() using the assertFalse() method.

The java class implementing the factory method is as follows:

```

NumericStringCheckerFactory.java
package com.test.junit;

public class NumericStringCheckerFactory {

    public static boolean isNumericString(final String str) {

        if(str == null || str.isEmpty()) return false;

        for(char c : str.toCharArray()){
            if(Character.isDigit(c)){
                return true;
            }
        }
        return false;
    }
}

```

We write a simple test case to check all the values where the Boolean value false should be returned.

The test class is as follows:

```

NumericStringCheckerNegativeTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class NumericStringCheckerNegativeTest {

    @Test
    @DisplayName("Test null input")
    void testNullInput() {
        assertFalse(NumericStringCheckerFactory.isNumericString(null));
    }

    @Test
    @DisplayName("Test empty input")
    void testEmptyInput() {
        assertFalse(NumericStringCheckerFactory.isNumericString(""));
    }

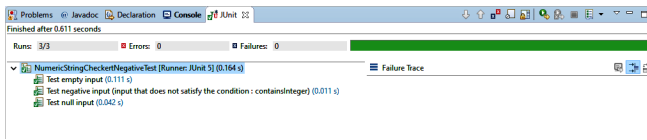
    @Test
    @DisplayName("Test negative input (input that does not satisfy the condition : containsInteger)")
    void testStringNotContainsDigit() {
        assertFalse(NumericStringCheckerFactory.isNumericString("TEST"));
    }
}

```

We have created a new test class named NumericString Checker NegativeTest containing three test methods that test different valid and invalid inputs that can be passed to the isNumericString() method.

In each case, the method is programmed to return a Boolean value that is false.

The results are:



As seen from the results, the test cases are executed with success.

Another way of checking invalid or out of bound input scenarios for testing is the use of the `assertThrows()` method. This method helps write test cases that test the exceptions thrown by the method to be tested. The `assertThrows()` method of the `Assertion` class provided by JUnit Jupiter takes any one of the following types as an input parameter:

- An object of the type `Class` which indicated the type of the exception that is supposed to be thrown.
- An object of the type `Executable` which invokes the system or program under test.
- An error message that is optional.

This method returns an object of type `Throwable`.

We consider a simple class that has a method `divideIntegers()` which performs division of two integers. This method throws an `ArithmeticException` when we try to divide any integer by zero.

The class is as follows:

```

NumericDivision.java
package com.test.junit;

public class NumericDivision {

    @public int divideIntegers(int a, int b) {

        if(b== 0)
            throw new ArithmeticException("Illegal Operation: cannot divide by zero");
        else
            return a/b;
    }
}

```

Now, we proceed to write a test class that tests whether this exception is thrown.

The test class is as shown below:

```

NumericDivisionTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

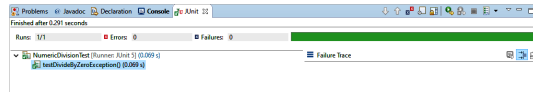
class NumericDivisionTest {

    @Test
    void testDivideByZeroException() {
        NumericDivision numericDivision = new NumericDivision();
        Exception exception = assertThrows(ArithmeticException.class, () ->
            numericDivision.divideIntegers(4, 0));
        assertEquals("Illegal Operation: cannot divide by zero", exception.getMessage());
    }
}

```

The test class consists of a simple test method that performs an assertion of the Exception thrown by the `divideIntegers()` method. As seen from the test method, the `assertThrows()` method returns an object of type `Exception`. This object can be used to extract the error message thrown and another assertion can be performed to check whether the correct error message is received.

We execute the tests and obtain the results as follows:



As seen from the test results, the test case passes.

Similar to the `ArithmeticException`, different exceptions can be tested using the `assertThrows()` method.

An example that tests the `IllegalArgumentException` and the `NumberFormatException` is shown below:

```

ExceptionTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class ExceptionTest {

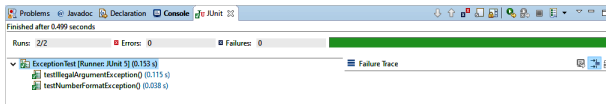
    @Test
    void testIllegalArgumentException() {
        Exception exception = assertThrows(IllegalArgumentException.class, () -> {
            throw new IllegalArgumentException("Illegal input argument provided");
        });
        assertEquals("Illegal input argument provided", exception.getMessage());
    }

    @Test
    void testNumberFormatException() {
        assertThrows(NumberFormatException.class, () -> {
            Integer.parseInt("TEST");
        });
    }
}

```

The above test class checks two different exception classes with the help of the `assertThrows()` method.

On execution, we obtain the following results:



The tests pass and the exceptions have been tested.

Although we have tested the exceptions that would be thrown when incorrect arguments are passed to the method under test, we can also test for the superclass of the exception.

Let us consider the following piece of code that throws a `NumberFormatException`.

```

IllegalArgumentTester.java
package com.test.junit;

public class IllegalArgumentTester {

    public static void main(String args[])
    {
        Integer.parseInt("5d");
    }
}

```

When we run this code, we get an exception as shown below:

```

Problems  Javadoc  Declaration  Console  JUnit
<terminated> IllegalArgumentTester [Java Application] C:\Program Files\Java\jdk1.8.0_251\bin\javaw.exe
Exception in thread "main" java.lang.NumberFormatException: For input string: "5d"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:590)
    at java.lang.Integer.parseInt(Integer.java:615)
    at com.test.junit.IllegalArgumentTester.main(IllegalArgumentTester.java:7)

```

As seen from the exception above, the error message of the exception starts with 'For input string.'

We write a simple test case that tests the same piece of code, but, in the assertion checks for a super class of the NumberFormatException. We check if the exception thrown is of the type RuntimeException as seen in the diagram below:

```

TestSuperClassException.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

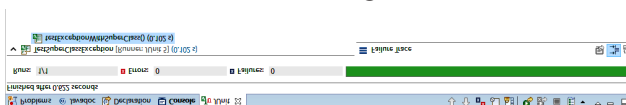
class TestSuperClassException {

    @Test
    public void testExceptionWithSuperClass() {
        Exception exception = assertThrows(RuntimeException.class, () -> {
            Integer.parseInt("5d");
        });
        String expectedErrorMessage = "For input string";
        String actualErrorMessage = exception.getMessage();
        assertTrue(actualErrorMessage.contains(expectedErrorMessage));
    }
}

```

The test method checks for an exception that is of the superclass RuntimeException which is true. Further, as the initial message of the expected error message is known to us, another assertion is done and the error message obtained is verified as well.

On execution, we obtain the following results:



As visible in the results, the test case pass successfully.

- Example 9: Disable a Test Case or Class

In this section, we try to understand how a test case can be programmed to be disabled. This action can be achieved easily with the help of the Disabled annotation. This annotation can be used on a test method or a test class.

To demonstrate the use of this annotation, we consider the test class `NumericAdditionTest` that we have previously covered in Example 2.

```
1 NumericAdditionTest.java
2 package com.test.junit;
3
4 import static org.junit.jupiter.api.Assertions.*;
5
6 import org.junit.jupiter.api.DisplayName;
7 import org.junit.jupiter.api.Test;
8
9 class NumericAdditionTest {
10
11     @Test
12     @DisplayName("Test NumericAddition Of Two Integers")
13     void testNumericAddition() {
14         NumericAddition adder = new NumericAddition();
15         assertEquals(5, adder.addNumber(2, 3));
16     }
17
18     @Test
19     @DisplayName("Test NumericAdditionWithMessage()")
20     void testNumericAdditionWithMessage() {
21         NumericAddition adder = new NumericAddition();
22         assertEquals(3, adder.addNumber(1, 2));
23     }
24 }
```

Now, we proceed to disable the test method `testNumericAddition()` as follows:

```
1 NumericAdditionTest.java
package com.test.junit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class NumericAdditionTest {

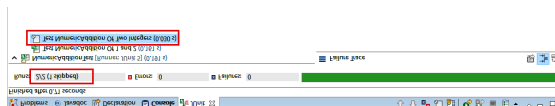
    @Test
    @DisplayName("Test NumericAddition Of Two Integers")
    @Disabled
    void testNumericAddition() {
        NumericAddition adder = new NumericAddition();
        assertEquals(5, adder.addNumber(2, 3));
    }

    @Test
    @DisplayName("Test NumericAddition Of 1 and 2")
    void testNumericAdditionWithMessage() {
        NumericAddition adder = new NumericAddition();
        assertEquals(3, adder.addNumber(1, 2));
    }
}
```

We simply make use of the Disabled annotation to deactivate the test method.

Now, we execute the test class.

The results are:



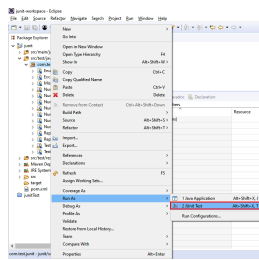
As we can observe from the results, 1 test case has been skipped and goes unexecuted. In a similar manner, a test class can be disabled by using the Disabled annotation at the class level.

- Example 10: Execution of Several Test Classes or a Test Suite

JUnit5 does not support creation of a test suite. If we wish to execute several test classes in succession, it can be done by executing all the tests present in the package.

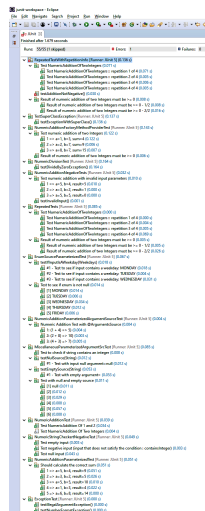
The steps are:

First, we navigate to the package that contains all the test cases as shown below:



We right-click on the package name and navigate to the Run As option. We go to Run As -> JUnit Test and click on it.

All the test classes that are present in the package are executed as shown below:



As seen from the results, all the test classes that are present in the package have been executed.

- Example 11: Backward Compatibility with JUnit4

The JUnit-vintage project helps maintain backward compatibility with JUnit4 and JUnit3. We shall look at a simple example demonstrating the use of the vintage dependencies to effectively execute tests written using JUnit4 within the JUnit5 context.

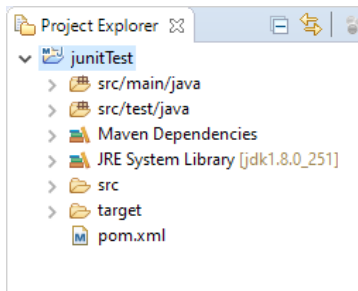
In order to execute a test written in JUnit4, the following two dependencies are needed:

```
<dependency>
<groupId>org.junit.vintage</groupId>
<artifactId>JUnit-vintage-engine</artifactId>
<version>{JUnit-jupiter-version}</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>JUnit</groupId>
<artifactId>JUnit</artifactId>
<version>{JUnit4-version}</version>
<scope>test</scope>
</dependency>
```

In addition to the standard JUnit5 dependencies that include JUnit-jupiter-api and the JUnit-jupiter-engine, we provide the dependencies of JUnit-vintage and the version of JUnit4 respectively.

We consider a simple maven project containing test cases written in JUnit4.

The project hierarchy is as shown below:



The project has a simple test class written in JUnit4 as shown below:

```
JUnit4Test.java
package com.test.junitTest;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class JUnit4Test {

    @Test
    public void testAddition() {
        assertEquals(5, 3 + 2);
    }
}
```

If we try to execute this test case in the JUnit5 context, we get an exception.

Hence, we add the two dependencies required to the pom.xml file as shown below:

```
JUnitTest/pom.xml
<?xml version="1.0" encoding="UTF-8" ?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.test</groupId>
    <artifactId>junitTest</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

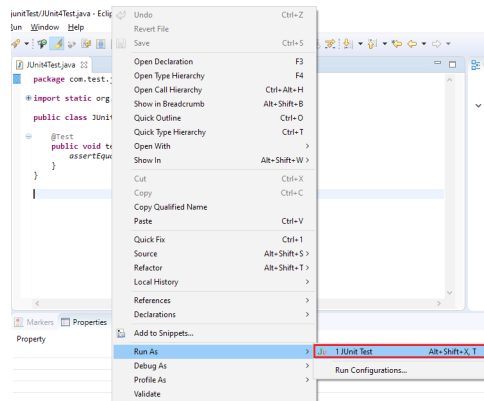
    <name>junitTest</name>
    <url>http://maven.apache.org/</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

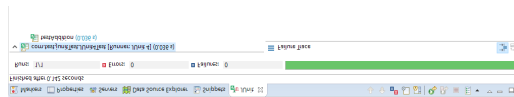
    <dependencies>
        <dependency>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
            <version>5.6.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.6.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.6.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-engine</artifactId>
            <version>1.6.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.opentest4j</groupId>
            <artifactId>opentest4j</artifactId>
            <version>1.2.0</version>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-commons</artifactId>
            <version>1.6.2</version>
        </dependency>
        <dependency>
            <groupId>org.apiguardian</groupId>
            <artifactId>apiguardian-api</artifactId>
            <version>1.1.0</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.2</version>
                <configuration>
                    <excludedGroups>slow</excludedGroups>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

We now proceed to our test class, right click on the class and select Run As -> JUnit Test.



The test executes without error as seen from the following results:



Now that we have covered JUnit5 in detail, we will proceed to take a look at some other testing frameworks. In the next section, we will study the TestNG and the Selenium testing frameworks respectively.

6.2. TESTNG FRAMEWORK

TestNG is a testing framework created by Cédric Beust in 2004. This is a simple automated testing framework that bore its inspiration from JUnit and NUnit. The ‘NG’ in its name stands for ‘Next Generation’ and was introduced to eliminate certain limitations of the preceding unit testing frameworks such as JUnit and NUnit. This framework introduced interesting and helpful features that gave it an edge over other testing frameworks (Beust and Sulemani, 2004).

Some of the features offered by this framework are:

- Flexibility in terms of configuration of tests;
- Annotations that are simple and easy to understand and implement;
- Ability to run tests in parallel;
- Ability to group multiple test cases;
- Cross browser testing is possible;
- Easy integration with build and integration tools such as maven, Jenkins;
- Support for testing that is data-driven;

- Ability to execute tests in large thread pools;
- Ability to generate readable test reports;
- Ability to provide logging of tests;
- Support for parameter and sequence-based testing;
- Support for writing tests that are dependent on one another.

The TestNG framework is designed in a manner that makes it possible to cover different types of testing such as unit, functional, integration, and end to end testing.

Additionally, the annotations are simple and easy to understand, which is an added bonus (Beust and Sulemani, 2004).

Some of the advantages of this framework over JUnit are:

- Provision for execution of tests parallelly;
- Grouping of tests is possible;
- Generation of logs;
- Generation of test reports;
- Simpler configuration;
- Easier annotations.

In the next section, we cover some basic annotations provided by TestNG.

6.2.1. TestNG Annotations

Some of most used annotations in TestNG are as follows (Beust and Sulemani, 2004):

1. **@Test:** This annotation is used to denote that the method annotated with this annotation is a test method of a testing test class.
2. **@BeforeSuite:** This annotation is used to denote that method will be executed before all the tests defined in the test suite are executed.
3. **@AfterSuite:** This annotation is used to denote that method will be executed after all the tests defined in the test suite are executed.
4. **@BeforeTest:** This annotation is used to denote that method will be executed before each test method in a test class that has been annotated with the @Test annotation is executed.

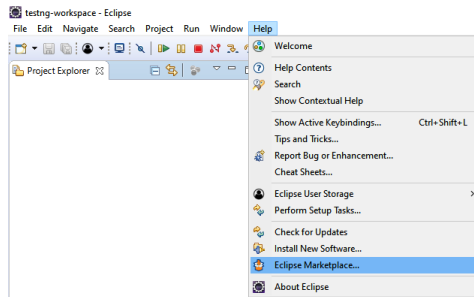
5. **@AfterTest:** This annotation is used to denote that method will be executed after each test method in a test class that has been annotated with the **@Test** annotation are executed.
6. **@BeforeGroups:** This annotation is used to define the configuration for a group of tests. It is used to denote that the method with this annotation will be executed before the list of groups specified using this annotation. A method annotated using this annotation will run before the very first method belonging to any of the groups specified in the list is executed.
7. **@AfterGroups:** This annotation is used to define the configuration for a group of tests. It is used to denote that the method with this annotation will be executed after the list of groups specified using this annotation. A method annotated using this annotation will run after the trailing or last method belonging to any of the groups specified in the list is executed.
8. **@BeforeClass:** The annotation is used to indicate that the method denoted by this annotation would be executed before any method belonging to the specified test class is executed.
9. **@AfterClass:** The annotation is used to indicate that the method denoted by this annotation would be executed after all the methods belonging to the specified test class have been executed.
10. **@BeforeMethod:** This annotation is used to indicate that the method annotated using this annotation will be executed before each test method is executed.
11. **@AfterMethod:** This annotation is used to indicate that the method annotated using this annotation will be executed after each test method is executed.

Before we study concrete examples of test cases written using TestNG, we shall see how this framework can be installed in the next section.

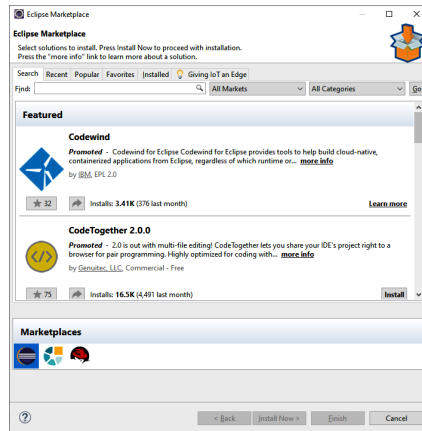
6.2.2. TestNG Setup in Eclipse

To start with, the setup, we proceed with Eclipse. Although there are other ways, we will just look at this method of setup.

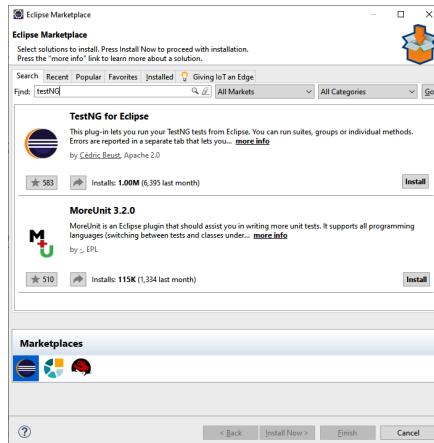
This is one of the simplest ways of configuring testing in our project. Eclipse provides a simple way to download and install the TestNG framework. We open Eclipse and navigate to the Help tab. We select 'Eclipse Marketplace.' This is shown below:



A window opens up as shown below:

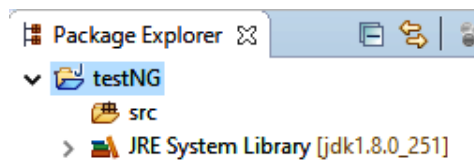
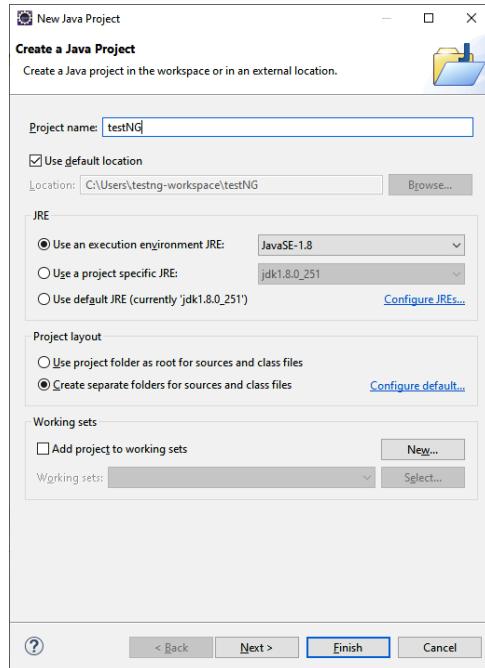
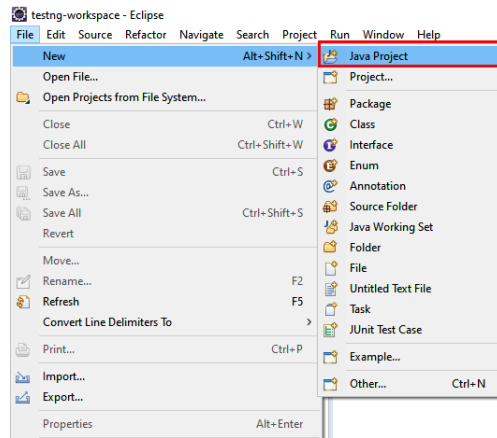


We proceed to search for the testng plugin as shown below:

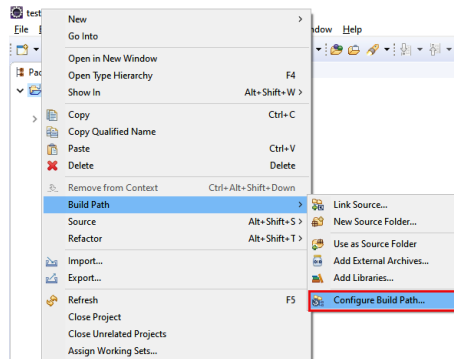


We select the first option-TestNG for Eclipse.

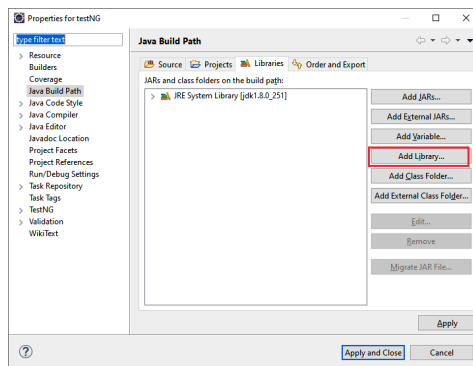
We follow the instructions from Eclipse marketplace and restart eclipse. Once the restart is done, we create a new Java project to write our test cases. This is done as follows:



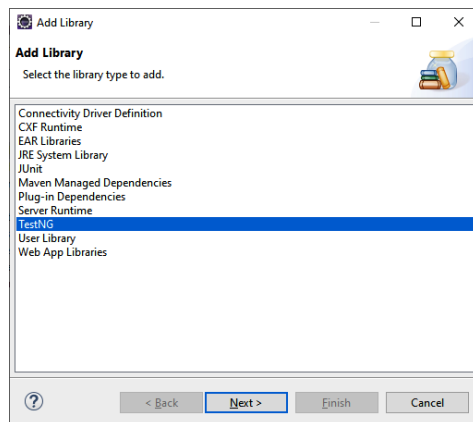
As seen from the image above, we have successfully created a project. We now proceed to add the TestNG library. We right click on the project and go to Build Path-> Configure Build Path.



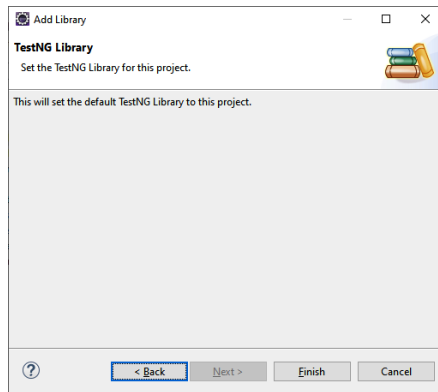
A screen is displayed as shown:



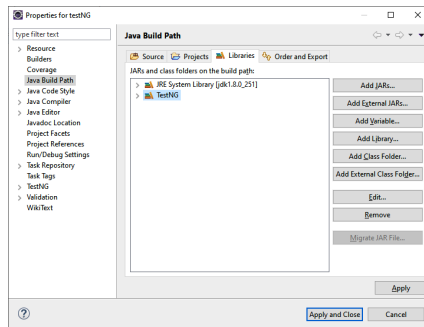
We click on Add Library and from the available options, we select TestNG.



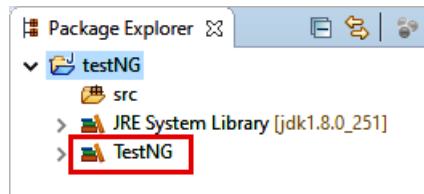
We receive the following popup.



We click on Finish which adds the library to the project's build path.



We apply the changes and close the window. As we can see below, the TestNG library has been added to our project.



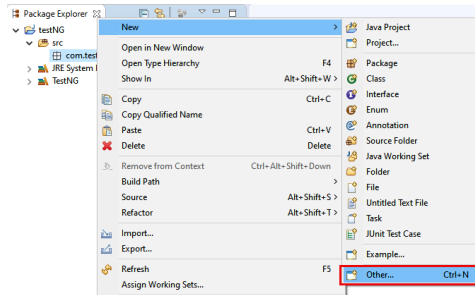
Now that we have covered the basic setup required for implementation of TestNG in our java project, we proceed to look at some examples of the use of TestNG in the upcoming section.

6.2.3. Examples Using TestNG

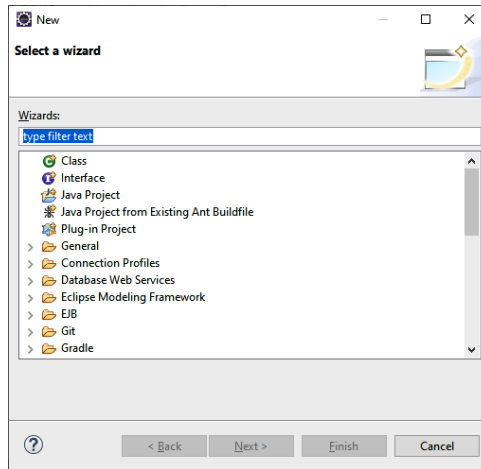
- Example 1: First TestNG Test Case

In our java project, we create a new package and proceed to create a TestNG test case as follows:

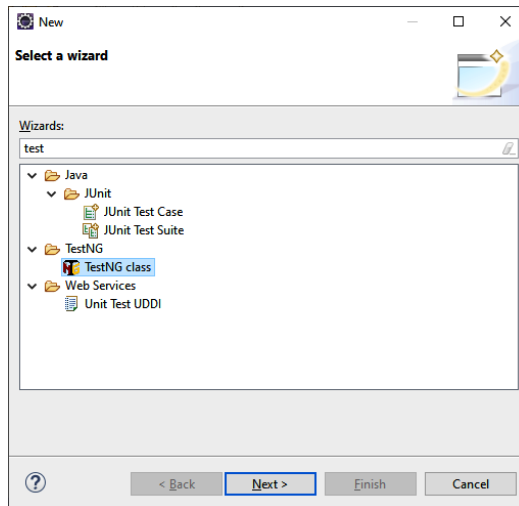
We right click on the package and navigate to New -> Other.



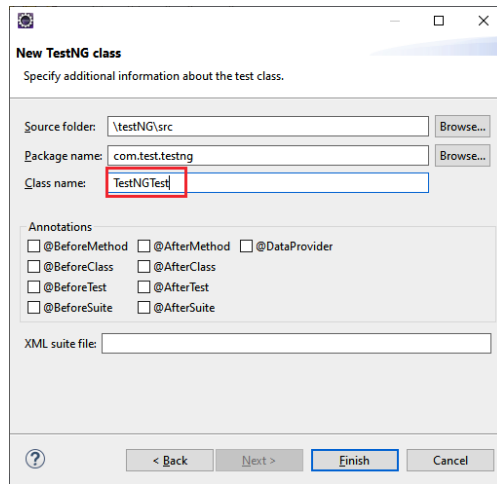
The New Wizard window is displayed as follows:



We search for the type TestNGTestClass as seen below and click on it.



A new window is displayed and it prompts us to provide details of the new test case.



We provide the name, check if the source folder and package names are correct, and click on Finish.

```

NumericStringCheckerFactory.java
package com.test.junit;

public class NumericStringCheckerFactory {

    public static boolean isNumericString(final String str) {

        if(str == null || str.isEmpty()) return false;

        for(char c : str.toCharArray()){
            if(Character.isDigit(c)){
                return true;
            }
        }
        return false;
    }
}

```

A new test class with a default implementation of a test method is generated as shown above.

We proceed to edit this class and update the test method as follows:

```

TestNGTest.java
package com.test.testng;

import static org.testng.Assert.assertEquals;

import org.testng.annotations.Test;

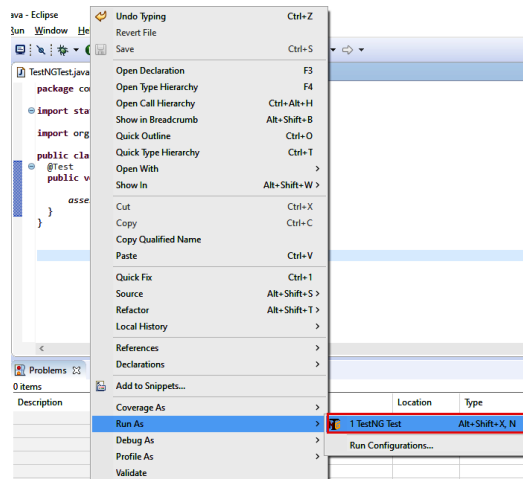
public class TestNGTest {

    @Test
    public void testAddition() {

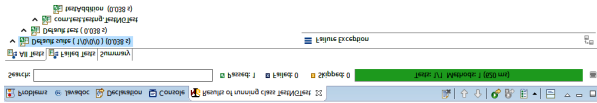
        assertEquals(5, 2 + 3);
    }
}

```

We have created a simple test method that performs an assertion to check if the addition operation works correctly. To execute this test case, we right click on the class and navigate to Run As -> TestNG Test as shown below:

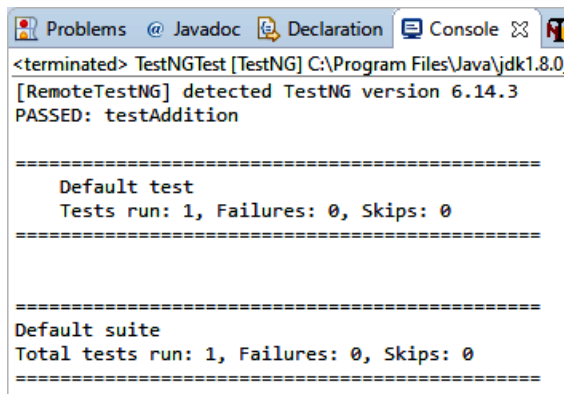


The test executes and the results of execution are shown below:

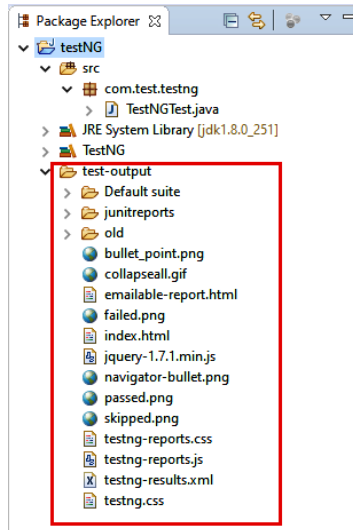


As seen from the results, the test case has been executed without failure. An interesting this to note is that when we use the TestNG framework, we obtain the results in two tabs. The first tab is the Results tab and the second one is the console. The results are printed on the console as well.

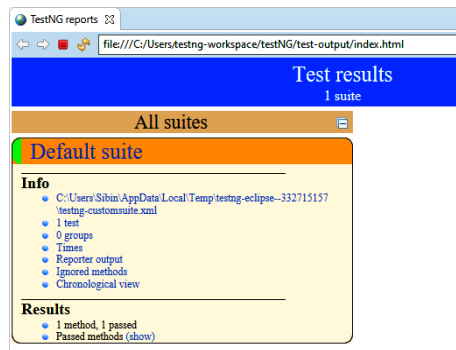
This is shown below:



In addition to printing/displaying the results in two different ways, a new folder is generated in our project as well. It should be noted that we need to refresh the project after each project to obtain the latest version of this folder. We refresh our project to locate the newly generated folder called test-output.



This folder provides additional data regarding the test cases such as a report, the passed tests, and the failed tests and so on. We click on `index.html` and open it using the default web browser provided by Eclipse. The results of the test suite are displayed as seen below:



This is one excellent feature that is provided by TestNG as it provides detailed information regarding the test cases and the test suite in an easy, accessible, and readable format.

Now that we have seen a simple test case using TestNG, we shall proceed to study different annotations provided by this framework.

- Example 2: Parameterized Tests

In this example, we study different ways in which we can write parameterized tests. Passing parameters to the test cases is an important functionality which is provided by TestNG.

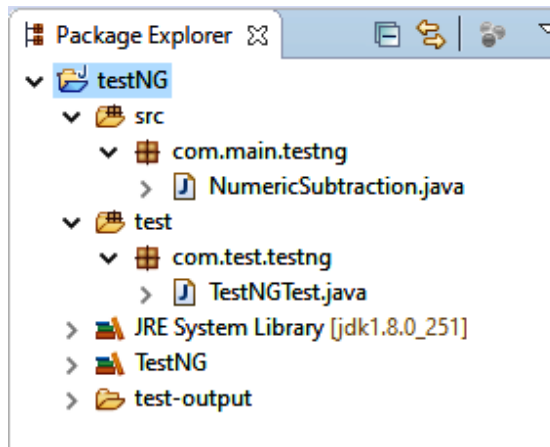
Parameterized tests can be written using the TestNG framework in two ways:

1. **Using an Xml File:** The first way of implementing parameterized test cases in TestNG is by means of a configuration xml file named testng.xml. This file contains details of the parameters that need to be passed to the test class and can be updated anytime.

We make use of the Parameters annotation which is used to pass the parameters to the test method.

To begin, we create a new package named com.main.testng with a simple java class named NumericSubtraction that performs subtraction of two integers.

The project structure is shown below:



The class NumericSubtraction contains of a single method subtract() that performs a subtraction operation on two integers.

This is shown below:

The screenshot shows the code editor for the file 'NumericSubtraction.java'. The code is as follows:

```
package com.main.testng;

public class NumericSubtraction {

    public int subtract(int a, int b)
    {
        return a-b;
    }
}
```

We proceed to create a simple test class with a test method that accepts parameters as shown below:


```

NumericSubtractionParameterizedTest.java
package com.test.testng;

import static org.testng.AssertJUnit.assertEquals;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

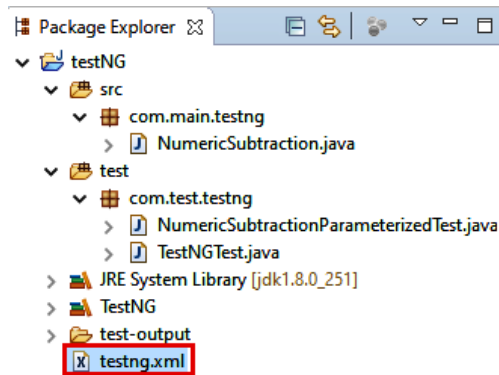
import com.main.testng.NumericSubtraction;

public class NumericSubtractionParameterizedTest {

    @Test
    @Parameters({ "a", "b" })
    public void testNumericSubtraction(int a, int b) {
        NumericSubtraction subtractor = new NumericSubtraction();
        assertEquals(7, subtractor.subtract(a, b));
    }
}

```

As we can see, a test method named `testNumericSubtraction()` accepting two input parameters has been created. The 'Parameters' annotation is used to define the names of the parameters needed as an input for this method. The values to be passed need to be defined in the `testng.xml` configuration file. We proceed to create this file at the project level as shown below:



We edit this file to include the test case and pass the parameters a and b along with their respective values as shown below:

```

testng.xml
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="parameterized-test">

    <test name="NumericSubtractionParameterizedTest">

        <parameter name="a" value="10" />
        <parameter name="b" value="3" />

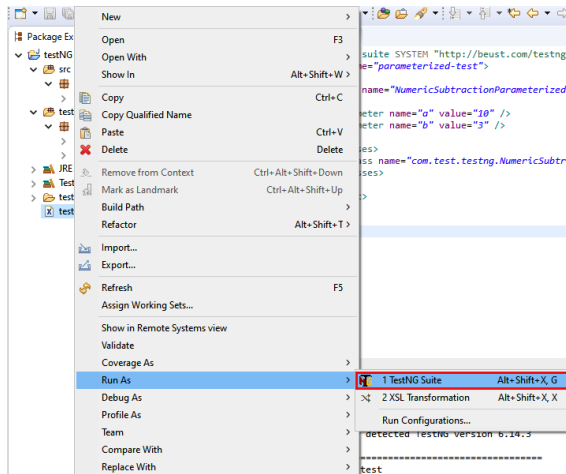
        <classes>
            <class name="com.test.testng.NumericSubtractionParameterizedTest" />
        </classes>

    </test>
</suite>

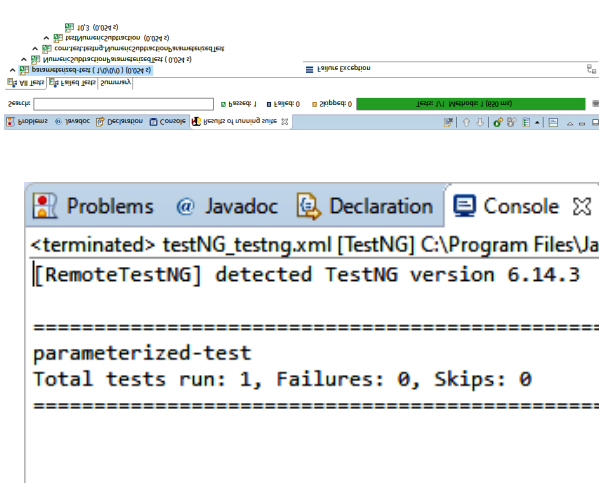
```

We provide a name for the test, we pass the name of the test class, and the parameters required.

We not run the test by right clicking on the file and navigating to Run As -> TestNG Suite.



We execute the test and obtain the results as shown below:



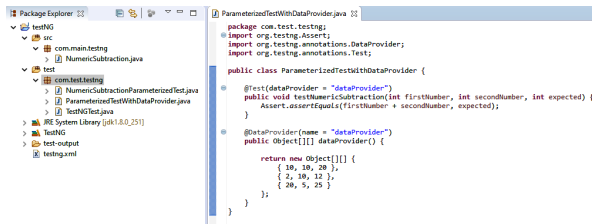
As visible from the test results, the parameterized test has passed successfully.

2. Using a Data Provider Method: Another useful way of providing data to test methods is by use of a data provider method.

This can be done by using the `DataProvider` annotation (Beust and Sulemani, 2004). We can write a simple method that provides the required data and returns an Object array and use this annotation to define the method to be a data provider method.

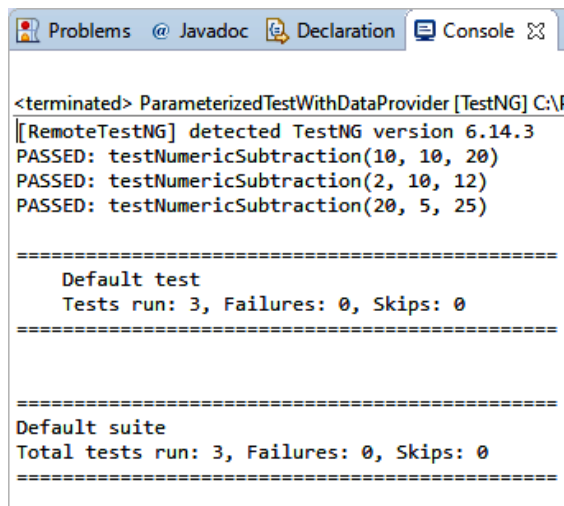
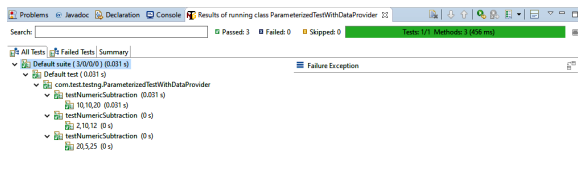
The annotation `Test` is used with a placeholder titled 'dataProvider' which is used to indicate that the input parameters will be provided by a method whose name is defined using this attribute.

To see the working of these annotations, we create a simple test class which tests the addition of two integers as shown below:



In the test class, we have added a method that provides the data for the test method. Using the Test annotation, we provide the value/name of the data provider method using the dataProvider attribute. In the test method, we use a simple assert statement that checks if the addition of the first two integers is equal to the expected value which is provided using the data provider method.

We execute the test case as a TestNG Test class. The results are as follows:

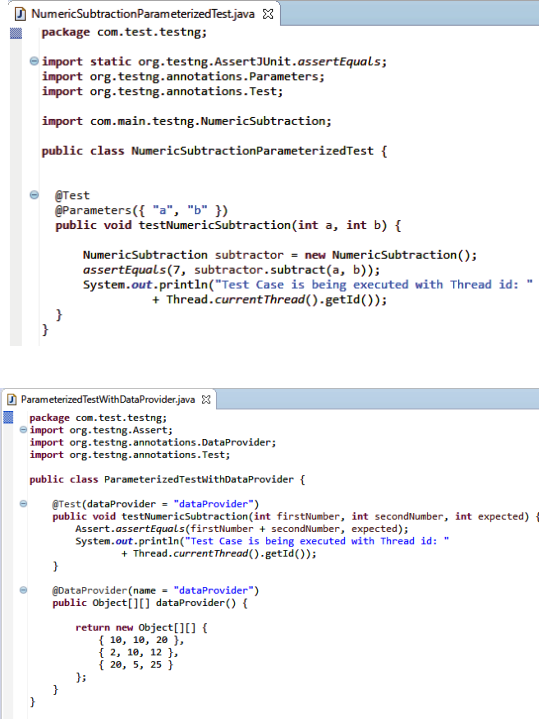


As visible from the results, for the three sets of data that has been provided, the test has been run for a total of three times.

- Example 3: Parallel Tests

TestNG provides an innovative feature that allows us to execute tests in a parallel manner.

In this example, we look at an example of tests that are run in parallel. We consider the following two test classes:



```

NumericSubtractionParameterizedTest.java
package com.test.testng;

import static org.testng.AssertJUnit.assertEquals;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

import com.main.testng.NumericSubtraction;

public class NumericSubtractionParameterizedTest {

    @Test
    @Parameters({ "a", "b" })
    public void testNumericSubtraction(int a, int b) {

        NumericSubtraction subtractor = new NumericSubtraction();
        assertEquals(7, subtractor.subtract(a, b));
        System.out.println("Test Case is being executed with Thread id: "
            + Thread.currentThread().getId());
    }
}

ParameterizedTestWithDataProvider.java
package com.test.testng;

import org.testng.Assert;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class ParameterizedTestWithDataProvider {

    @Test(dataProvider = "dataProvider")
    public void testNumericSubtraction(int firstNumber, int secondNumber, int expected) {
        Assert.assertEquals(firstNumber + secondNumber, expected);
        System.out.println("Test Case is being executed with Thread id: "
            + Thread.currentThread().getId());
    }

    @DataProvider(name = "dataProvider")
    public Object[][] dataProvider() {

        return new Object[][] {
            { 10, 10, 20 },
            { 2, 10, 12 },
            { 20, 5, 25 }
        };
    }
}

```

The first test class is the parameterized test class which receives its parameters via the testng.xml file and the second test class is a parameterized test class using a data provider method. In each of the test methods, we have added a print statement that prints the id of the thread which is executing the test method.

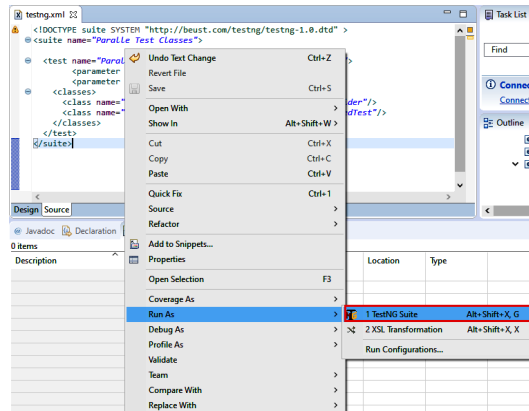
For the purpose of executing tests in parallel, we need the testng.xml configuration file. We provide the manner of execution of the test class or test method.

We make use of the ‘parallel’ attribute to specify whether classes or methods or suites are being executed in parallel. In addition, we also provide the thread count to configure the number of threads that must be used.

The updated testng.xml file is shown below:

```
<?xml version='1.0' encoding='UTF-8'>
<!DOCTYPE suite SYSTEM 'http://beust.com/testng/testng-1.0.dtd'>
<suite name='Parallel Test Classes'>
  <test name='Parallel Tests' parallel='classes' thread-count='2'>
    <parameter name='a' value='10' />
    <parameter name='b' value='3' />
    <classes>
      <class name='com.test.testing.ParameterizedTestWithDataProvider' />
      <class name='com.test.testing.NumericSubtractionParameterizedTest' />
    </classes>
  </test>
</suite>
</xml>
```

We have updated the file to add a test suite containing the names of two test classes that must be executed in parallel. We use the attributes 'parallel' and 'thread-count' to specify that the classes must be executed in parallel and that two threads must be used. For the test class that accepts parameters via the testing.xml file, we provide the required parameters. We proceed to execute the test suite as shown below:



The results obtained are as follows:

```
<terminated> testNG_testng.xml [TestNG] C:\Program Files\Java\
[RemoteTestNG] detected TestNG version 6.14.3
Test Case is being executed with Thread id: 12
Test Case is being executed with Thread id: 11
Test Case is being executed with Thread id: 11
Test Case is being executed with Thread id: 11

=====
Paralle Test Classes
Total tests run: 4, Failures: 0, Skips: 0
=====
```

As we can see from the results, both the test classes have executed with success. The console output clearly shows us that two different threads have been used to execute each test class. From the results received in the TestNG tab, we can observe that the entire test suite took only 0.14 seconds which is highly efficient.

Parallel testing is a value-add to traditional unit testing frameworks. In software projects, time is always a factor that plays a part in determining the amount of testing that can be done. If tests can be executed parallelly, a lot of time can be spent on execution and observation can be saved and more time can be allocated to developing detailed test suites. This helps the project in more than one way which is necessary to deliver a project with minimal defects.

Just as we executed two test classes in parallel, test methods can be programmed to execute in parallel. Let us see how this can be done.

We add a test case to the `ParameterizedTestWithDataProvider` test class as follows:

```

1 ParameterizedTestWithDataProvider.java 21
package com.test.testing;
import org.testng.Assert;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class ParameterizedTestWithDataProvider {

    @Test(dataProvider = "dataProvider")
    public void testNumericSubtraction(int firstNumber, int secondNumber, int expected) {
        Assert.assertEquals(firstNumber + secondNumber, expected);
        System.out.println("Test case is being executed with Thread id: "
            + Thread.currentThread().getId());
    }

    @Test(dataProvider = "invalidDataProvider")
    public void testNumericSubtractionWithIncorrectValues(int firstNumber, int secondNumber, int expected) {
        Assert.assertEquals(firstNumber + secondNumber, expected);
        System.out.println("Test case is being executed with Thread id: "
            + Thread.currentThread().getId());
    }

    @DataProvider(name = "dataProvider")
    public Object[][] dataProvider() {
        return new Object[][] {
            { 10, 10, 20 },
            { 2, 10, 12 },
            { 20, 5, 25 }
        };
    }

    @DataProvider(name = "invalidDataProvider")
    public Object[][] invalidDataProvider() {
        return new Object[][] {
            { 10, 10, 20 },
            { 2, 10, 12 },
            { 20, 5, 25 }
        };
    }

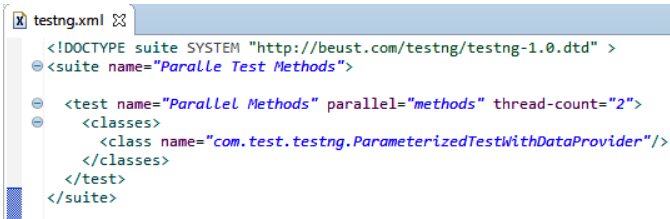
    @DataProvider(name = "invalidDataProvider")
    public Object[][] invalidDataProvider() {
        return new Object[][] {
            { 10, 10, 20 },
            { 2, 10, 12 },
            { 20, 5, 99 }
        };
    }
}

```

A new test method accepting parameters to check the `NumericSubtraction` class has been added to the test class as shown above. The method tests whether the `subtract()` method works correctly by providing invalid values of the expected results of subtraction. We make use of a data provider for this method and provide data with incorrect values of the final results of subtraction as shown above.

Now, we proceed to configure the `testng.xml` to run the test methods of this class in parallel manner.

We again make use of the ‘parallel’ and ‘thread-count’ attributes as shown below:

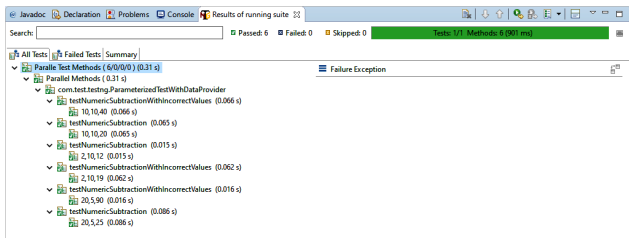
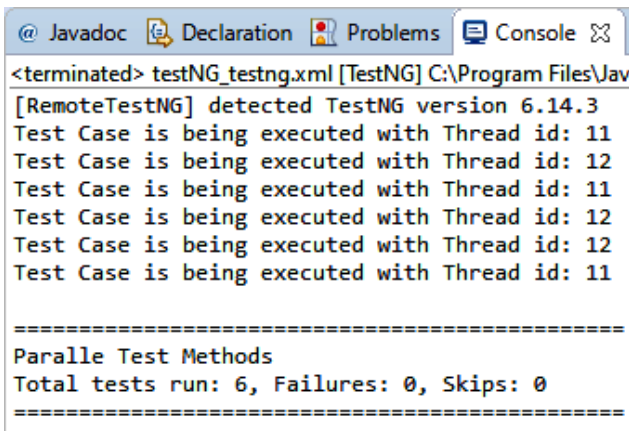


```

<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="Parallel Test Methods">
  <test name="Parallel Methods" parallel="methods" thread-count="2">
    <classes>
      <class name="com.test.testng.ParameterizedTestWithDataProvider"/>
    </classes>
  </test>
</suite>

```

We provide the name of test class, whose methods are to be executed in parallel as shown above. We then execute it as a test suite. The results are:

```

@ Javadoc Declaration Problems Console
<terminated> testNG_testng.xml [TestNG] C:\Program Files\Jav
[RemoteTestNG] detected TestNG version 6.14.3
Test Case is being executed with Thread id: 11
Test Case is being executed with Thread id: 12
Test Case is being executed with Thread id: 11
Test Case is being executed with Thread id: 12
Test Case is being executed with Thread id: 12
Test Case is being executed with Thread id: 11

=====
Parallel Test Methods
Total tests run: 6, Failures: 0, Skips: 0
=====

```

The results clearly demonstrate that two threads have been used to execute the test methods. Each of the methods has been executed three times as three sets of data have been provided for each of them. The results show that a total of 6 test cases have been executed and the time of execution is 31 seconds. This feature is a highly useful feature that can be instrumental in improving the quality of testing in projects.

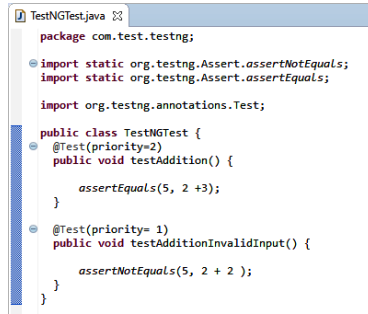
- Example 4: Prioritizing Tests

Similar to JUnit, TestNG provides features that help us execute test cases in a certain order.

Ordering of test can be done in different ways based on the context. Classes, methods, and suites can be ordered.

In case of methods, a simple means of achieving this is by using the priority placeholder in the Test annotation.

This is shown below:



```

package com.test.testing;

import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertTrue;
import org.testng.annotations.Test;

public class TestNGTest {
    @Test(priority=2)
    public void testAddition() {
        assertEquals(5, 2 + 3);
    }

    @Test(priority= 1)
    public void testAdditionInvalidInput() {
        assertEquals(5, 2 + 2 );
    }
}

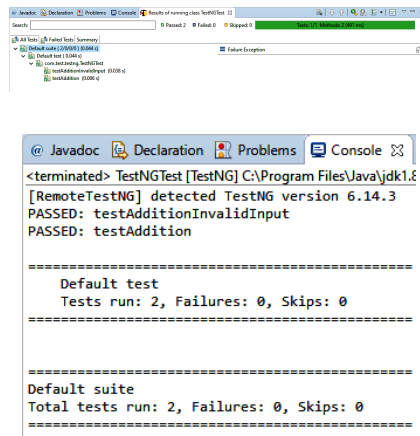
```

We have created a simple test class with two test methods—one that tests the addition of two valid inputs and another that checks the addition of two numbers against an invalid result.

We assign a priority to each of the test methods using the ‘priority’ attribute or placeholder.

We execute the test class by right clicking on it and executing it as a TestNG Test.

The results are as follows:



```

<terminated> TestNGTest [TestNG] C:\Program Files\Java\jdk1.8
[RemoteTestNG] detected TestNG version 6.14.3
PASSED: testAdditionInvalidInput
PASSED: testAddition

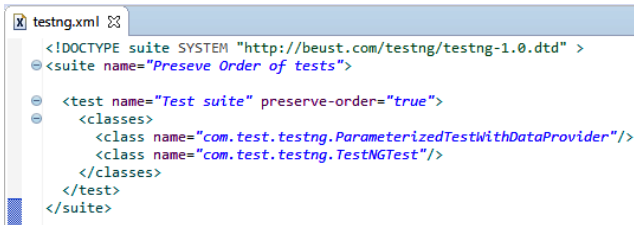
=====
Default test
Tests run: 2, Failures: 0, Skips: 0
=====

Default suite
Total tests run: 2, Failures: 0, Skips: 0
=====

```

The results distinctly demonstrate that the tests have executed in the order of their priorities. The method testAdditionInvalidInput(), with the higher priority has been executed first.

Similarly, ordering of test classes can be done using the testng.xml file. In case of test class, the testing.xml file can be used so that order of execution of the test classes is preserved. This is shown below:



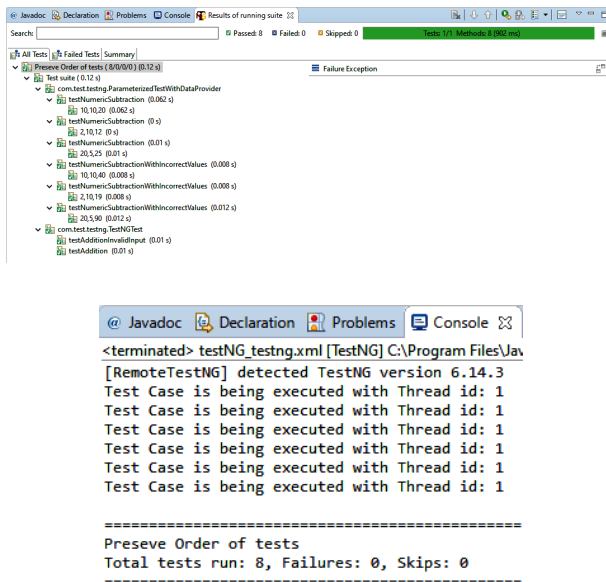
```

<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="Preseve Order of tests">
  <test name="Test suite" preserve-order="true">
    <classes>
      <class name="com.test.testng.ParameterizedTestWithDataProvider"/>
      <class name="com.test.testng.TestNGTest"/>
    </classes>
  </test>
</suite>

```

We execute two test classes `ParameterizedTestWithDataProvider` and `TestNGTest` (used in previous examples) in a sequential manner. As indicated in the file, we have made use of the ‘preserve-order’ attribute which preserves the order provided in the file. This means that the test classes will be executed sequentially.

We execute the test suite and obtain the following results:



The screenshot shows the IDE interface with the 'Results of running suite' tab active. The test results are as follows:

Test Case	Time (s)	Status
com.test.testng.ParameterizedTestWithDataProvider	0.062	Pass
testNumericsSubtraction (0.062 s)	0.062	Pass
testNumericsSubtraction (0 s)	0	Pass
testNumericsSubtraction (0.01 s)	0.01	Pass
testNumericsSubtraction (0.01 s)	0.01	Pass
testNumericsSubtraction (0.008 s)	0.008	Pass
testNumericsSubtraction (0.008 s)	0.008	Pass
testNumericsSubtraction (0.012 s)	0.012	Pass
com.test.testng.TestNGTest	0.01	Pass
testAdditionInvalidInput (0.01 s)	0.01	Pass
testAddition (0.01 s)	0.01	Pass

The console output shows the following messages:

```

<terminated> testNG_testng.xml [TestNG] C:\Program Files\Jav
[RemoteTestNG] detected TestNG version 6.14.3
Test Case is being executed with Thread id: 1
Test Case is being executed with Thread id: 1
Test Case is being executed with Thread id: 1
Test Case is being executed with Thread id: 1
Test Case is being executed with Thread id: 1
Test Case is being executed with Thread id: 1

=====
Preseve Order of tests
Total tests run: 8, Failures: 0, Skips: 0
=====

```

As the test results have established, the test classes have been executed in the order that has been indicated in the `testng.xml` file.

It should be noted that by default (when the ‘preserve order attribute is not specified), the value of the attribute `preserve-order` is set to ‘False.’ When this value has not been configured, the tests are executed in an alphabetical order of the names of the test methods.

But, when the attribute ‘preserve-order’ is set to `True`, the test cases are executed in the order which has been specified in the `testng.xml` file.

- Example 5: Retrying Failed Test Cases

In most software projects, once an initial round of testing has been done, only the failed tests need to be executed again. This process, although not ideal, is usually implemented in most software projects, this helps validate the failed test cases in a faster way. The re-execution of only the test cases that failed has been simplified by TestNG as it provides support for re-running tests.

As we have seen from the previous examples, we make use of the `testng.xml` file to configure test suites.

On execution of the suite, in the test-output folder, a file is generated that contains all the failed tests. It is labeled as `testng-failed.xml`. So, if we wish to execute the failed test cases again, one must simply execute this suite in the way one executes the `testng.xml` file. Execution of this file executes only the failed test cases.

Retrying of test cases is important as sometimes; the true reason of failure remains unknown. It is possible that some tests failures are unrelated to the code in the project and are related to the network, environment, or other factors.

In case of projects that depend on multiple servers, there are chances of unavailability or other access issues. Hence, test cases need to be rerun so as to confirm that the failures are related to some bug in the code and no other factors. False negative test results can be eliminated by means of rerunning the tests.

To proceed with the configuration of rerunning the test cases after failure in an automatic manner, we need to make use of the `IRetryAnalyzer` interface in our projects that are based on TestNG.

TestNG provides the `IRetryAnalyzer` interface that has methods that help control and configure the test class for the purpose of retrying the tests. The interface has a method called `retry()` that we must override to specify a limit on the number of times a test should be retried (Beust and Sulemani, 2004).

To begin, we need to create a separate class that implements the `IRetryAnalyzer` interface as shown below:

```

Retry.java
package com.test.testing;

import org.testng.IRetryAnalyzer;
import org.testng.ITestResult;

public class Retry implements IRetryAnalyzer {

    int count = 0;
    int maximumTries = 4;

    @Override
    public boolean retry(ITestResult result) {

        if(count < maximumTries)
        {
            count++;
            return true;
        }
        return false;
    }
}

```

As observable from the above class, we have implemented the `IRetryAnalyzer` interface and overridden the `retry()` method. We have implemented a simple logic and set the maximum number of retries to 4. When the counter reaches 4, the test method is no longer executed.

We proceed to create a simple test class with a single test method that is programmed to fail by default as shown below:

```

RetryTest.java
package com.test.testing;

import static org.testng.Assert.assertEquals;

import org.testng.annotations.Test;

public class RetryTest {

    @Test(retryAnalyzer = Retry.class)
    public void testAddition() {

        assertEquals(true, false);
    }
}

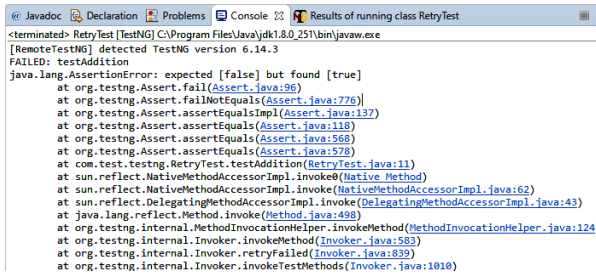
```

The class `RetryTest` has a test method containing an assert statement that will always fail. We make use of the ‘`retryAnalyzer`’ attribute to specify the class to be used for the purpose of retrying the test case. This attribute denotes that the test method would be retried in case of a failure based on the configuration provided in the class named ‘`Retry`.’

We proceed to execute this class as a TestNG test and receive the following results:

The screenshot displays the TestNG results for the class `RetryTest`. The top bar indicates 0 Passed, 1 Failed, and 3 Skipped tests. The 'Failed Tests' tab is selected, showing a single failed test: 'Default test (0.053 s)'. The test is part of the 'com.test.testing.RetryTest' suite. The failure is detailed in the 'Failure Exception' pane, showing a `java.lang.AssertionError: expected [false] but found [true]` at `org.testng.Assert.fail(Assert.java:96)`. The stack trace includes the following frames:

- `org.testng.Assert.fail(Assert.java:96)`
- `org.testng.Assert.failNotEquals(Assert.java:776)`
- `org.testng.Assert.assertEqualsImpl(Assert.java:137)`
- `org.testng.Assert.assertEquals(Assert.java:118)`
- `org.testng.Assert.assertEquals(Assert.java:568)`
- `org.testng.Assert.assertEquals(Assert.java:578)`
- `com.test.testing.RetryTest.testAddition(RetryTest.java:11)`
- `org.testng.remote.AbstractRemoteTestNG.run(AbstractRemoteTestNG.java:114)`



```

@ Javadoc Declaration Problems Console Results of running class RetryTest
-terminated: RetryTest [TestNG] C:\Program Files\Java\jdk1.8.0_251\bin\javaw.exe
[RemoteTestNG] detected TestNG version 6.14.3
FAILED: testAddition
java.lang.AssertionError: expected [false] but found [true]
    at org.testng.Assert.fail(Assert.java:96)
    at org.testng.Assert.failNotEquals(Assert.java:779)
    at org.testng.Assert.assertEqualsImpl(Assert.java:132)
    at org.testng.Assert.assertEquals(Assert.java:118)
    at org.testng.Assert.assertEquals(Assert.java:568)
    at org.testng.Assert.assertEquals(Assert.java:578)
    at com.test.testing.RetryTest.testAddition(RetryTest.java:11)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at org.testng.internal.MethodInvocationHelper.invokeMethod(MethodInvocationHelper.java:124)
    at org.testng.internal.Invoker.invokeMethod(Invoker.java:583)
    at org.testng.internal.Invoker.retryFailed(Invoker.java:839)
    at org.testng.internal.Invoker.invokeTestMethods(Invoker.java:1018)

```

As observed from the results, the test case has been executed 4 times and only after the fourth failure, it is labeled as failed. The console output gives as an assertion exception as expected.

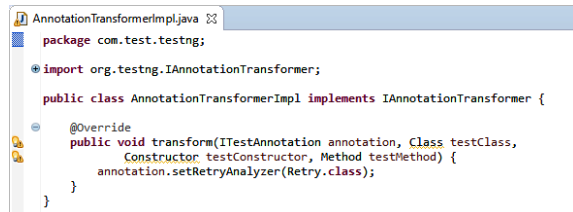
This is a straightforward method of implementing the retry logic for test cases. But, as annotations are static, when the values need to be updated or modified in some way, a recompilation of the code is needed.

But, TestNG provides us with another way of overriding the `retry()` method at runtime by means of a listener class. The default behavior can be overridden at runtime by using the `IAnnotationTransformer` listener. `IAnnotationTransformer` is a listener provided by TestNG that permits us to modify TestNG annotations and also configure them during runtime (TestNG, 2004).

The listener has a `transform` method that is called for every test method during execution. This listener can be used to configure it to set a `RetryAnalyzer` that is custom.

We can implement this listener in our class and provide an overridden value for the retry listener.

This is done as follows:



```

AnnotationTransformerImpl.java
package com.test.testing;

import org.testng.IAnnotationTransformer;

public class AnnotationTransformerImpl implements IAnnotationTransformer {

    @Override
    public void transform(ITestAnnotation annotation, Class testClass,
        Constructor testConstructor, Method testMethod) {
        annotation.setRetryAnalyzer(Retry.class);
    }
}

```

We have created a new class named `AnnotationTransformerImpl` which implements the `IAnnotationTransformer` listener and overrides the `transform()` method where we provide a value for the retry analyzer that should be used by the test suite/class.

We modify the test class to remove the `retryAnalyzer` attribute shown below:

```

RetryTest.java
package com.test.testing;

import static org.testng.Assert.assertEquals;

import org.testng.annotations.Test;

public class RetryTest {
    @Test
    public void testAddition() {

        assertEquals(true, false);
    }
}

```

As the retry analyzer is set at runtime, we do not need to use the attribute in the test class.

The details of the listener must be provided in the testng.xml file as well.

This is done as follows:

```

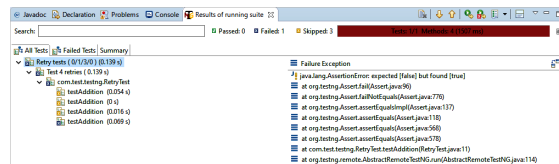
testng.xml
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="Retry tests">
    <listeners>
        <listener class-name="com.test.testing.AnnotationTransformerImpl"/>
    </listeners>
    <test name="Test 4 retries">
        <classes>
            <class name="com.test.testing.RetryTest"/>
        </classes>
    </test>
</suite>

```

The testng.xml file has been modified to add a listener for the test suite. We add the newly created listener AnnotationTransformerImpl to the testng.xml file.

The name of the test class to be executed is provided as always.

We proceed to execute it as a test suite. The results are:



As illustrated by the results, the test case has been executed for a total of 4 times before it is marked as failed. The console output remains the same as before.

- Example 6: Disabling or Ignoring Tests

Like JUnit, TestNG permits us to configure a test to be ignored.

This can be done by means of the ‘enabled’ attribute of the Test annotation. This attribute is used to set a Boolean value that defines the state of the test method as active or inactive respectively.

A straightforward and concise example demonstrating the use of this attribute is shown below:

```
IgnoreTestExample.java
package com.test.testng;
import org.testng.Assert;
import org.testng.annotations.Test;

public class IgnoreTestExample {

    @Test
    public void testAdditionEnabledByDefault() {
        Assert.assertEquals(2, 1 + 1);
    }

    @Test(enabled = true)
    public void testSubtraction() {
        Assert.assertEquals(5, 7 - 2);
    }

    @Test(enabled = false)
    public void testStringsNotEqual() {
        Assert.assertNotEquals("A", "B");
    }

}
```

The test class `IgnoreTestExample` consists of three methods, two of which make use of the `enabled` annotation. If it is set to `true`, the test case is active, if it is set to `false`, the test case is inactive.

It should be noted that if the `enabled` attribute is not configured, by default, the test is active and will execute.

To test this, we proceed to execute the test class as a TestNG test. The results are as follows:

```
<terminated> IgnoreTestExample [TestNG] C:\Program Files\Jav
[RemoteTestNG] detected TestNG version 6.14.3
PASSED: testAdditionEnabledByDefault
PASSED: testSubtraction

=====
Default test
Tests run: 2, Failures: 0, Skips: 0
=====

Default suite
Total tests run: 2, Failures: 0, Skips: 0
=====
```

As established by the results, the two test methods `testAdditionEnabledByDefault()` for which the `'enabled'` attribute has not been set and `testSubtraction()` for which the `'enabled'` attribute has been set

to true, have been executed whereas the method `testStringsNotEqual()` for which the 'enabled' attribute is set to false, has not been executed.

It is clear from the results that if the 'enabled' attribute is not configured, the test is executed by default.

Now that we have covered the TestNG framework, we proceed to study the Selenium framework in the next section.

6.3. SELENIUM FRAMEWORK

Selenium is a popular framework for automated testing that happens to be open source. It is a set of tools that is mostly used to validate web applications across a varying range of browsers and platforms (Gojare, Joshi, and Gaigaware, 2015). This tool helps automate tests of different type such as functional, system, regression, and acceptance. It is a set of tools and libraries that are implemented to support automated testing of web-based applications. One can test web applications developed using different programming languages using this framework. This useful framework helps control web browsers and emulates the actions a user performs when using such a browser. Additionally, it permits cross-browser testing which is extremely helpful for the purpose of validation of applications. Several programming languages such as Java, C#, Python, Ruby, Perl, etc., can be used to generate or create test scripts using Selenium (Razak and Fahrurazi, 2011).

Selenium is blanket of applications or a suite of applications with the following individual tools:

1. **Selenium IDE:** or integrated development environment is a simple tool provided by the selenium framework. It is used to develop simple test cases in selenium. The best part of this tool is that it is a browser extension. It is a browser extension for the browsers Firefox and Chrome and is an efficient way to develop or create selenium test cases. It can be installed easily like any browser plug-in. This makes it easy to use and install. What the IDE does is that it records the actions performed by a user of the web-based system by making use of selenium commands and certain parameters that have been defined in the context of the tool (Razak and Fahrurazi, 2011).

This tool helps save a lot of time as programming of any kind is not required. It also helps familiarize oneself with the syntax of scripts in Selenium. This

tool is generally used when no prior programming knowledge is needed for the purpose of testing. A good feature provided by this tool is that the scripts can further be exported to be used and manipulated in Selenium Grid or selenium web driver.

2. **Selenium WebDriver:** It is tool in the Selenium umbrella that allows a developer to develop tests that are responsive in nature in a scripting language of his/her choice. This is a tool of the selenium universe that allows the development and execution of tests against different web browsers. The added advantage is that a programming language can be used to write and develop test scripts (Bruns, Kornstadt, and Wichmann, 2009).

The Web Driver component is useful for testing websites, web applications, mobile websites, etc. Different APIs are provided by the WebDriver component that helps control the web browser and execute tests easily. Different operators such as conditional operators, if-then-else operators, etc., are provided by this driver. The WebDriver controls the web browser at the OS level.

The Selenium Web Driver supports the following programming languages:

- Java;
 - .Net;
 - Python;
 - PHP;
 - Perl;
 - Ruby.
3. **Selenium Grid:** It is a component that falls under the set of tools provided by Selenium and permits you to execute tests on distinct platforms. This means that tests can be run on different machines and different browsers at the same time. It permits the execution of multiple tests at the same time. Cross platform, testing can be done parallelly. Additionally, remote testing across platforms is possible using Selenium Grid which makes it a powerful tool. The ability of parallel test execution compiled with cross platform testing features help save a lot of time that is spent on testing. Multiple test suites can be run on different browsers across varied platforms simultaneously which helps improve the quality of the software project as considerable amount of time is saved if

Selenium is used (Bruns, Kornstadt, and Wichmann, 2009).

The Grid is implemented based on the hub-and-nodes concept. Here, the hub is the center and the machines are the nodes. The hub functions as the essential source or headquarters that gives different Selenium commands to all the nodes that are connected to it.

The nodes, or the machines that possess the browsers, receive the selenium commands remotely. The control that triggers the launching of the test cases is on the local machine and the execution is done on the remote machines. Tests can be run on a combination of multiple web browsers and operating systems combined which helps improve the overall testing coverage of web applications.

In the upcoming section, we will delve into the practical implementations of the Selenium WebDriver tool.

6.3.1. Selenium WebDriver

In this section, we shall see how the Selenium WebDriver can be implemented in our java web projects. The latest stable version of Selenium WebDriver is 3.141.59 which we will be installing and using in our examples.

Before we study some practical implementations of the Selenium WebDriver, we need to complete the setup for our web-based projects in Java.

6.3.1.1. *Selenium WebDriver Setup*

Selenium can be configured into a java project in two ways; by adding the jar files directly to our project or by adding the WebDriver dependencies using maven.

We shall see both methods.

1. **Addition of the Jar Files Directly:** A straightforward way the required selenium jar files can be added is download the jar files from the official selenium website and adding them to our project.

Let us see how this can be done.

First, we need to download the required jar files. For this task, we navigate to the following link:

<https://www.selenium.dev/downloads/>

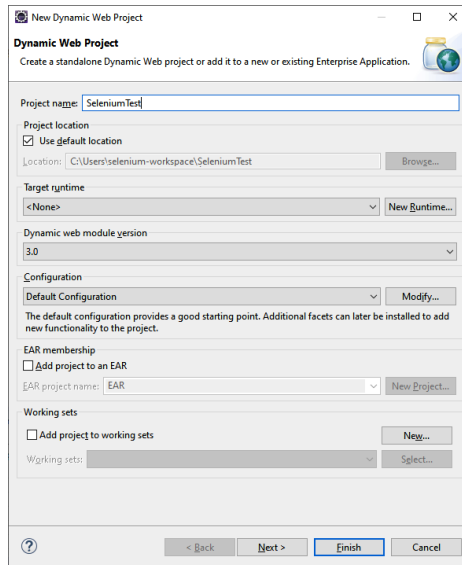
We click on the link and on the webpage; we navigate to the section for downloading the selenium webDriver and download the jar files from there.

Once the files have been downloaded, we proceed to installing selenium WebDriver in our project in Eclipse.

We first create a simple dynamic web project in Eclipse.

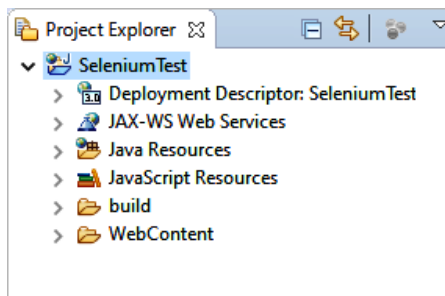
This is shown below:

We navigate to the new tab and navigate to New -> Dynamic Web Project. We select this option and a new project creation wizard is launched as shown below:

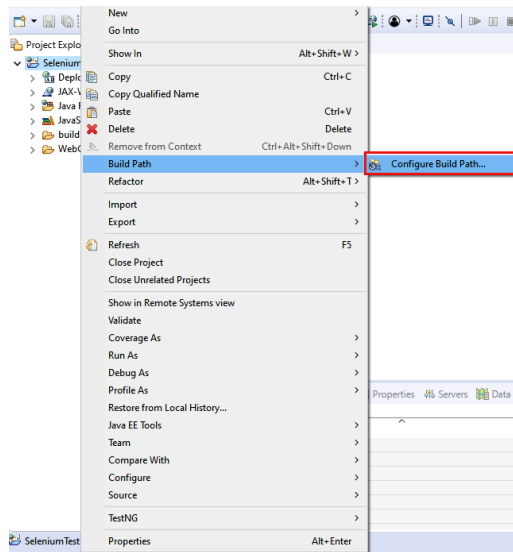


We provide the details needed and click on finish.

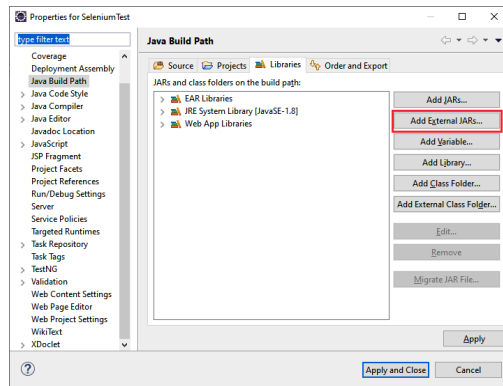
A simple web project is created as shown below:



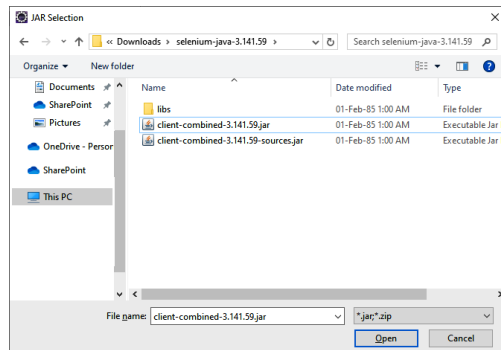
To add the jar files, we right click on the project and navigate to Build Path -> Configure Build Path and select it.



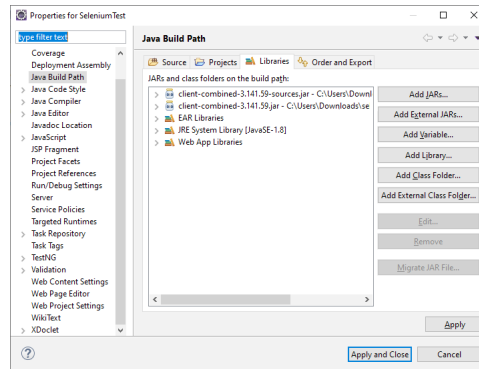
The following window opens up:



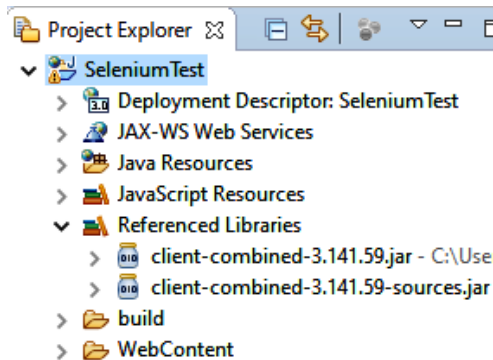
We click on the Add External JARs button. It prompts us to choose the jar files from the computer as shown below:



We select the downloaded jar file and click on open. They are now visible in the build path as seen below:



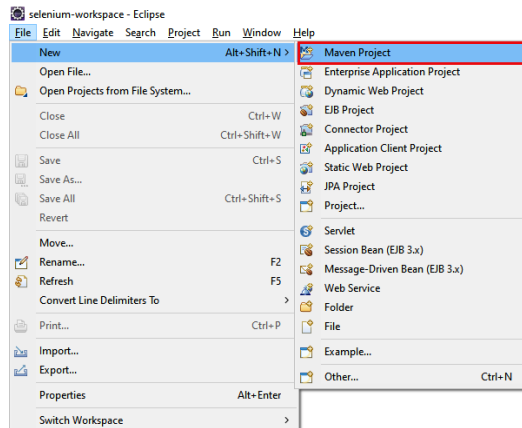
We click on the Apply and Close button. The jar files have now been successfully added to our project as seen in the image below:



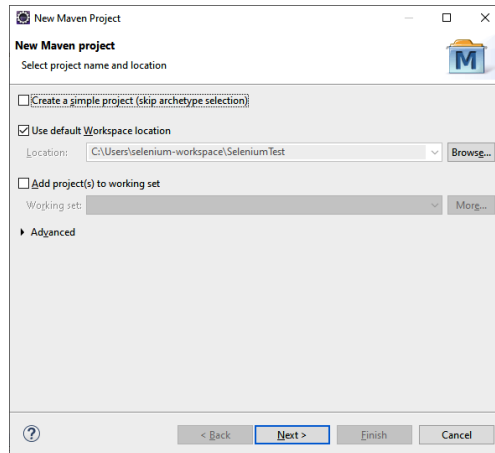
2. **Using Maven Dependency:** Maven is an impressive tool that aides in automation of the build process for Java projects. It can be used to inject the required artifact's dependencies in the project effortlessly. The setup of the selenium WebDriver using maven is uncomplicated and facile to implement. The required dependencies for the WebDriver need to be simply added to the pom.xml file in the maven project and maven automatically downloads the required jar files and adds it to our web project.

The steps followed are as follows:

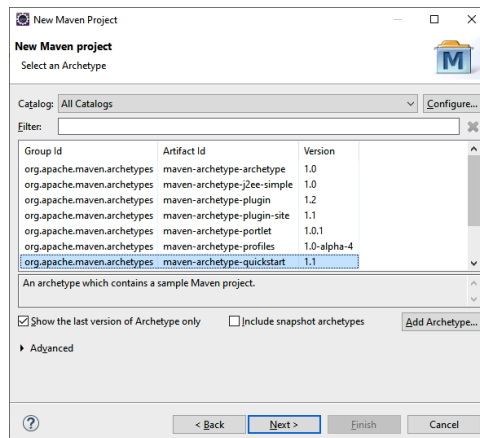
First, we create a new Maven project. To do this, we navigate to New -> Maven Project and click on Maven Project.

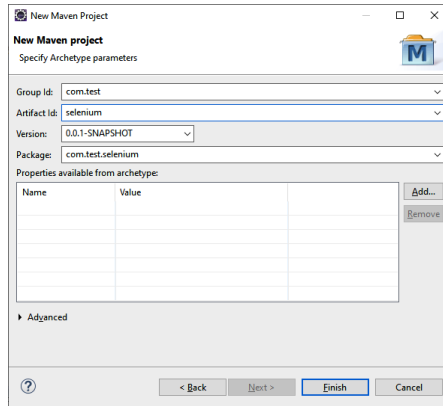


The project creation Wizard is launched as follows:

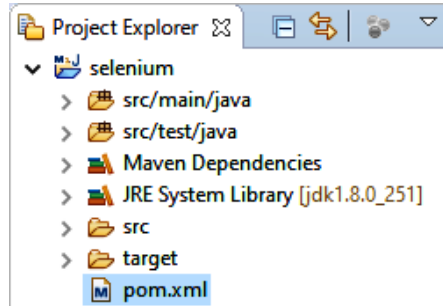


We select a quick-start project and provide the required details:





Once the details have been provided, we click on finish and complete the project creation.



A simple maven project has been created as seen.

For the purpose of adding the Selenium WebDriver dependency, we edit the pom.xml file and add the required dependencies as shown below:

```
selenium/pom.xml 12
<?xml version="1.0" encoding="UTF-8" ?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.test</groupId>
  <artifactId>selenium</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

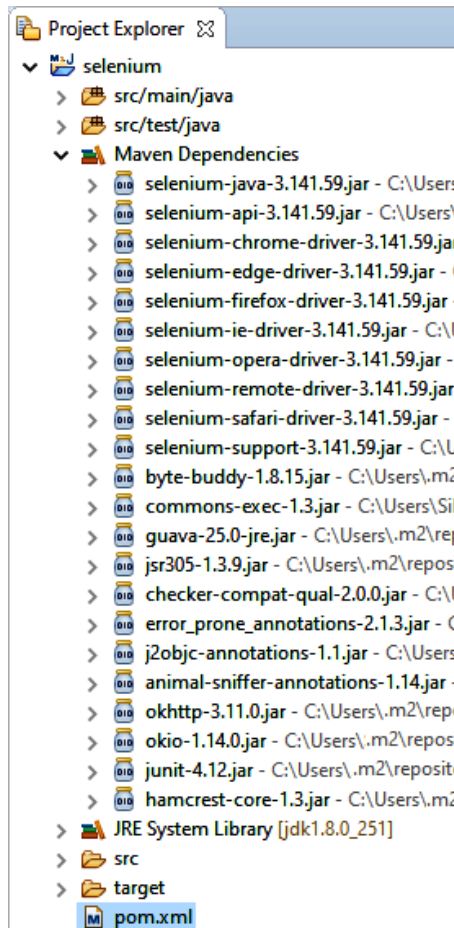
  <name>selenium</name>
  <url>http://maven.apache.org/</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>3.141.59</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit4</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

As observable from the pom.xml file, we have added the selenium-java maven dependency along with the JUnit dependency for the versions 3.141.59 and 4.12 respectively.

We build the project as a maven build and once this is done, the required jar files are added to our project as seen below:



It should be noted that the drivers for all the supported browsers have been added to our project as we have not specified a distinct driver. This can be done easily by providing the driver-specific dependency and excluding the rest.

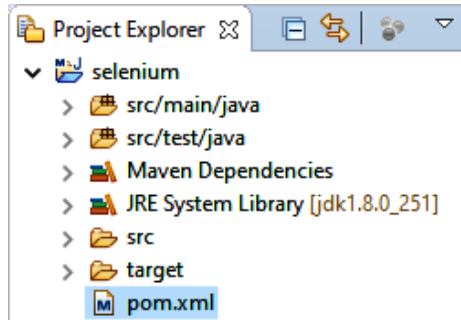
Having seen how the WebDriver can be installed in our projects, we proceed to studying some practical implementations of the same.

6.3.1.2. Selenium WebDriver Examples

- Example 1: Simple Selenium WebDriver Example to Open a Website

In this example, we shall make use of the WebDriver API to open a website via programming automation.

To do this, we consider the simple maven project we created during the setup of Selenium WebDriver.



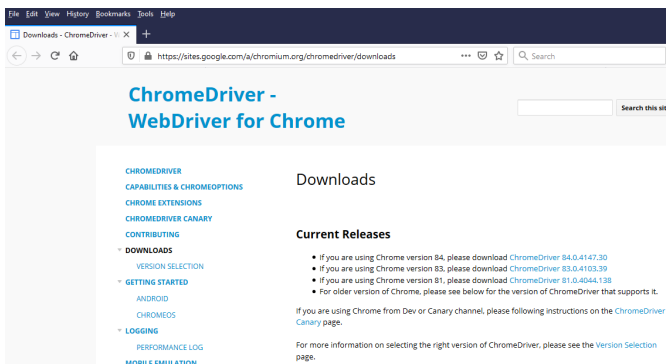
In this example, we will use the WebDriver API's to open the Google homepage in the Chrome browser.

To be able to use the Chrome browser using Selenium, we need to download the executable driver for Chrome. This is mandatory for implementing a script that opens a webpage using the Chrome browser. Selenium cannot do all the work of opening the webpage and interacting with it on its own. It needs help from the browser as well. This is where the Chrome driver comes into place as it helps Selenium to perform any actions on the Chrome browser. The Chrome driver is technically a standalone server implementing the connecting/wire mechanisms of WebDriver.

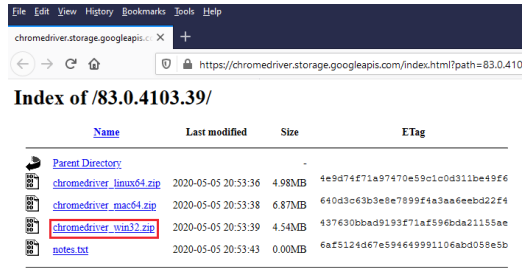
To download the Chrome WebDriver, we go to the official download page of ChromeDriver. The link is:

<https://sites.google.com/a/chromium.org/chromedriver/>

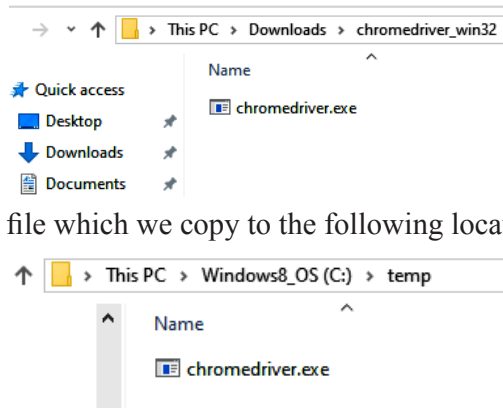
On opening the link, we obtain the following page:



As the version of Chrome that we are using is 83.0.4103, we proceed to download the ChromeDriver for Chrome version 83.



We proceed to download the driver for windows and unzip the compressed file.

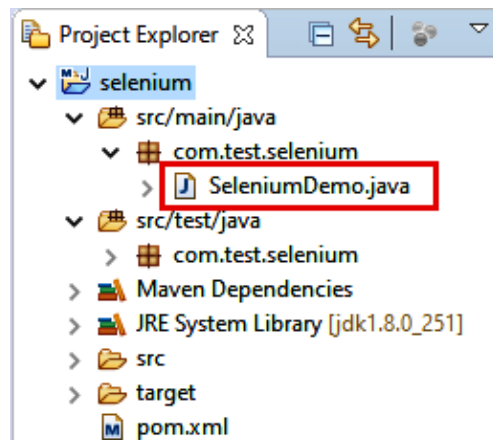


It has an exe file which we copy to the following location:

Now that we downloaded the required driver, we proceed to writing a simple script in Java.

We write a simple class that opens the link-<https://www.google.com>

We create a new class in the src/main folder of our project as follows:



The contents of our class are as follows:

```

SeleniumDemo.java
package com.test.selenium;

import java.util.concurrent.TimeUnit;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class SeleniumDemo {
    public static void main( String[] args )
    {
        //set the driver executable
        System.setProperty("webdriver.chrome.driver", "C:/temp/chromedriver.exe");

        //Initialize the Chrome driver
        WebDriver driver=new ChromeDriver();

        //apply a wait time
        driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);

        //maximize the window
        driver.manage().window().maximize();

        //open the Chrome browser with google URL
        driver.get("https://www.google.com");

        //wait for 5 seconds
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //close the browser
        driver.close();
    }
}

```

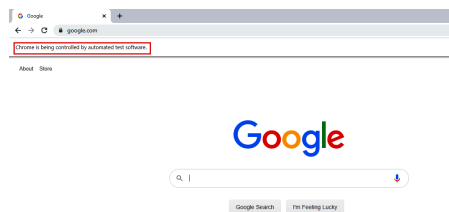
In this class, we first provide selenium with the location of the ChromeDriver using the webdriver property set as a variable. We Initialize the driver for the Chrome browser and proceed to open the Google homepage website. We make use of the `implicitlyWait()` function so as to ensure that we do not receive timeout errors before the site has opened up.

We maximize the window and proceed to wait for 5 seconds before closing the window.

To test this, we run this class as a standard Java application.

On execution, the Chrome browser is launched and then maximized.

The window is as follows:



As seen from the diagram, we have successfully opened the Google search homepage in the Chrome Browser.

The output on the console is as shown below:

```

Markers Properties Servers Data Source Explorer Snippets Console
<terminated> SeleniumDemo [Java Application] C:\Program Files\Java\jdk1.8.0_251\bin\javaw.exe
Starting ChromeDriver (35.0.4103.39 (c3b9f1c220219558064440843861342c28cd-refs/branch-heads/4103@/4416)) on port 8087
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggestions on keeping ChromeDriver safe.
ChromeDriver was started successfully.
[1595895908.721][WARNING]: This version of ChromeDriver has not been tested with Chrome version 84.
Jul 15, 2020 5:40:03 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: WC

```

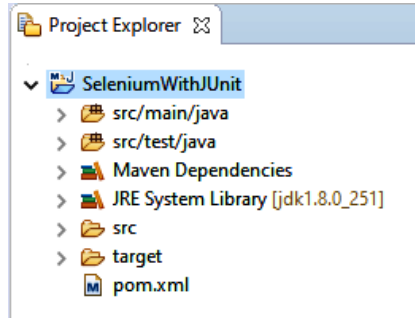
The console output shows that the Chrome browser has been opened by selenium.

We have created a simple program that automatically opens a website. We can take this further and program it to perform a host of actions. This is groundbreaking technology as testing can be immensely simplified by using this framework.

- Example 2: Writing a Test Case Using Selenium and JUnit

In this example, we explore selenium WebDriver in combination with JUnit. We write a test case in JUnit using the WebDriver.

First, we create a simple maven project named 'SeleniumWithJUnit' as shown below:



In this example, we will be using JUnit and Selenium. So, we add the required dependencies for JUnit and Selenium to our pom.xml file. We will be using JUnit version 5.6.2 and Selenium version 3.141.59, respectively.

This is shown below:

```

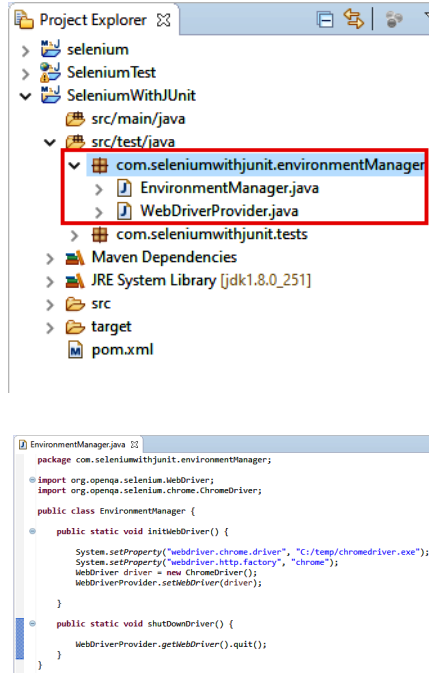
SeleniumWithJUnitpom.xml 11
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.xsi.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
  modelVersion="4.0.0">
  <groupId>com.seleniumwithjunit</groupId>
  <artifactId>seleniumwithjunit</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>3.141.59</version>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.6.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest</artifactId>
      <version>1.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-commons</artifactId>
      <version>1.6.2</version>
    </dependency>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-api</artifactId>
      <version>3.141.59</version>
    </dependency>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-engine</artifactId>
      <version>1.6.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.6.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-launcher</artifactId>
      <version>1.6.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-report-plugin</artifactId>
        <version>2.20.1</version>
      </plugin>
    </plugins>
  </reporting>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.19.1</version>
      </plugin>
      <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.2</version>
      </dependency>
      <dependency>
        <groupId>org.junit.platform</groupId>
        <artifactId>junit-platform-engine</artifactId>
        <version>1.6.1</version>
      </dependency>
    </plugins>
  </build>
</project>

```

Once we add the dependencies, we build the project via maven build so that all the required jar files are added to our project.

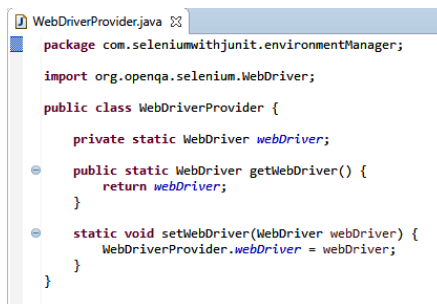
We proceed to create a package with classes that handle the test environment. This package will provide and initialize the drivers required for our JUnit test cases.

We create two classes in this package as follows:



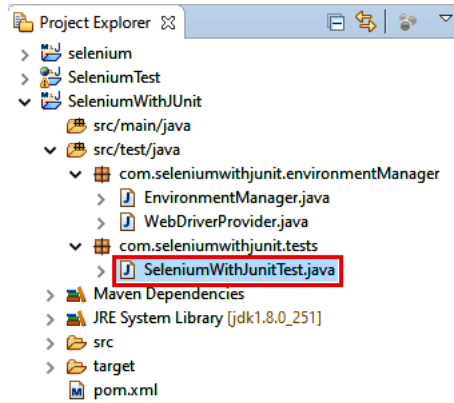
The EnvironmentManager class initializes the Chrome WebDriver and performs the shutdown activities. The location of the ChromeDriver is provided and it is used to initialize the WebDriver.

This class can be updated to provide initialization methods for other browsers as well.



The WebDriverProvider class provides setter and getter methods for a WebDriver instance.

Once the environment requirements are met, we proceed to write a JUnit Jupiter Test case in the src/test folder as shown below:



```

package com.seleniumwithjunit.tests;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;

import com.seleniumwithjunit.environmentManager.EnvironmentManager;
import com.seleniumwithjunit.environmentManager.WebDriverProvider;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.concurrent.TimeUnit;

public class SeleniumWithJUnitTest {

    @BeforeEach
    public void initializeDriver() {
        EnvironmentManager.initWebDriver();
    }

    @Test
    public void testURL() {
        //obtain the WebDriver
        WebDriver driver = WebDriverProvider.getWebDriver();

        //apply a wait time
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

        //open the Chrome browser with google URL
        driver.get("https://www.google.com");

        //retrieve the URL from the driver
        String driverURL = driver.getCurrentUrl();

        //check if the values match using JUnit assert
        assertEquals(driverURL, "https://www.google.com/");
    }

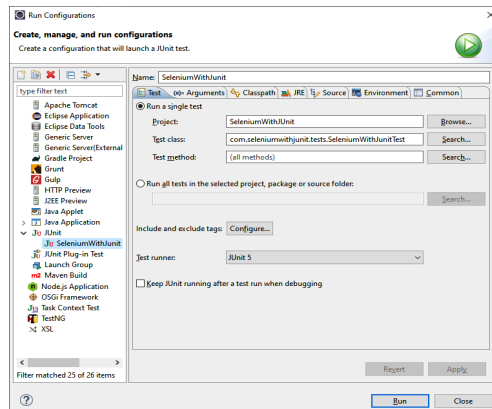
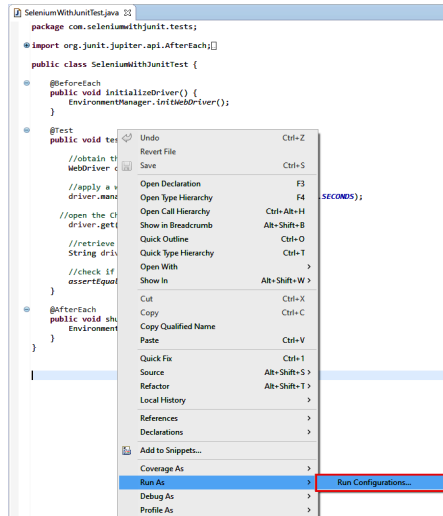
    @AfterEach
    public void shutDownDriver() {
        EnvironmentManager.shutDownDriver();
    }
}

```

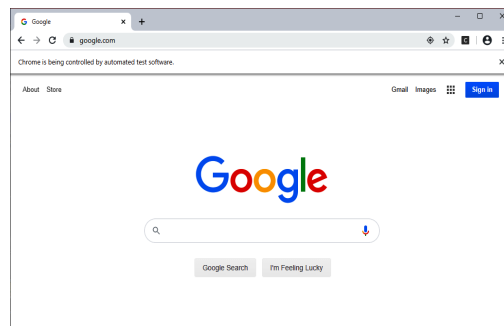
In our test case, we use the BeforeEach and AfterEach to call the EnvironmentManager that initializes and shuts down the web driver respectively.

We write a test method testURL() that opens up the Google search homepage and then checks the value of the URL retrieved from the driver against the expected value of the URL to verify that it matches.

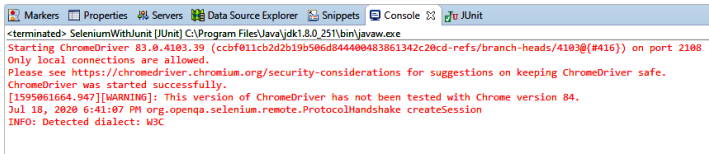
We run this test case by creating a JUnit5 run configuration by right clicking on the class and navigating to Run As -> Run Configurations



We create a new Run configuration and run the test case.
The Google search web page is opened as shown below:



The console output is as follows:

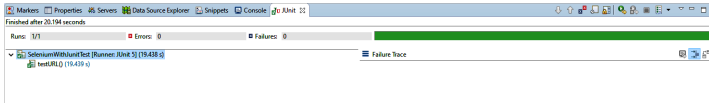


```

<terminated> SeleniumWithJUnit [JUnit] C:\Program Files\Java\jdk1.8.0_251\bin\java.exe
Starting ChromeDriver 83.0.4103.39 (ccb0f11cb2d2b19b506d844408483861342c20cd-refs/branch-heads/41030@{#416}) on port 2108
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggestions on keeping ChromeDriver safe.
ChromeDriver was started successfully.
[1595861664.947][WARNING]: This version of ChromeDriver has not been tested with Chrome version 84.
Jul 18, 2020 6:41:07 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C

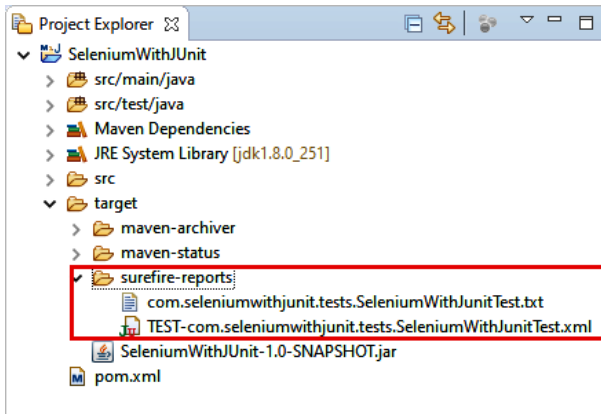
```

The result of execution of the test case is shown below:

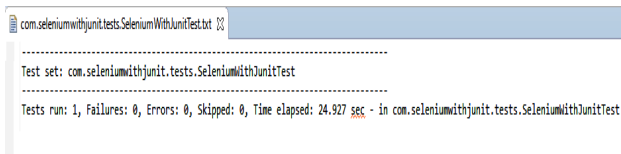


Additionally, as we have added the dependency for surefire reports in our project, we can see the reports of execution of the test case.

The surefire reports are generated on each maven build. They are located in the target folder at the following location:



The report is as follows:



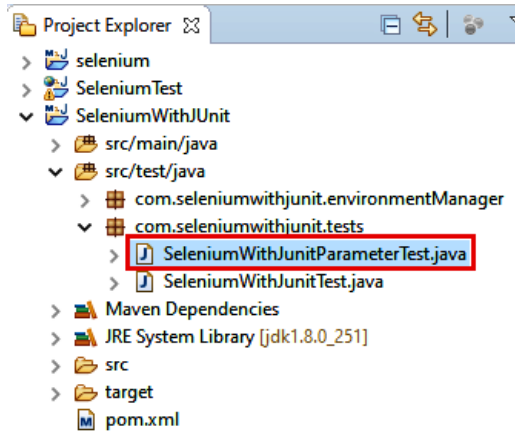
• Example 3: Selenium with JUnit with Parameter Passing

In this example, we take the previous example and configure the test case to search something on opening the Google search page.

The project configuration in the pom.xml stays the same as in the previous example. The EnvironmentManager package stays the same.

We add a new test case to the 'com.seleniumwithJUnit.tests' that opens up the Google search page and types in a search parameter.

This is shown below:



The contents of the test file are shown below:

```

SeleniumWithJUnitParameterTest.java
package com.seleniumwithjunit.tests;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

import com.seleniumwithjunit.environmentManager.EnvironmentManager;
import com.seleniumwithjunit.environmentManager.WebDriverProvider;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.concurrent.TimeUnit;

public class SeleniumWithJUnitParameterTest {

    @BeforeEach
    public void initializeDriver() {
        EnvironmentManager.initWebDriver();
    }

    @Test
    public void testURL() throws InterruptedException {
        //obtain the WebDriver
        WebDriver driver = WebDriverProvider.getWebDriver();

        //apply a wait time
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

        //open the Chrome browser with google URL
        driver.get("https://www.google.com");

        //finding the web element using name locator
        WebElement elem = driver.findElement(By.name("q"));
        elem.sendKeys(new String[]{"Logging Frameworks in Java By Neha Kaul"});
        elem.submit();

        Thread.sleep(5000);

        //retrieve the title from the driver
        String title= driver.getTitle();
        //check if the values match using JUnit assert
        assertEquals(title, "Logging Frameworks in Java By Neha Kaul - Google Search");
        //check if the title contains the search parameter
        assertTrue(title.contains("Neha Kaul"));
    }

    @AfterEach
    public void shutDownDriver() {
        EnvironmentManager.shutDownDriver();
    }
}

```

In this test class, we first use the WebDriver API provided by selenium to open the Google search homepage. Then, we make use of the WebElement API to type in some keywords into the search box. The WebElement is another API interface provided by Selenium that helps interact with the web browser.

Generally, whenever we open a web page, the first thing one does is to interact with the page in some way. This is done via the use of buttons, click on a link, providing some inputs to the page, selecting options from a dropdown list, etc. The primary action in test automation following the opening of any web page is to interact with it. This is where the WebElement API comes into place (Bruns, Kornstadt, and Wichmann, 2009).

The WebElement can be thought of as the Selenium representation of various HTML elements that can be found on any web page. Like the WebDriver interface, the WebElement is also an interface that is provided by Selenium to aid in the process of automating tests. We make use of the WebDriver to get the web element representing the search text box on the Google homepage. Once the web element is recovered, it can be manipulated and different operations can be performed on the element. This is what we have done.

In the test method, we pass the following string as a search parameter 'Logging Frameworks in Java by Neha Kaul.' This value is submitted to the search function of Google which will proceed to search for results matching the input.

We recover the title of the web page that has been currently opened by the driver. We then proceed to confirm if the results displayed has the title that matches the search parameters. Additionally, we check if the title contains some of the keywords from our original search parameter.

This way, we can check if the test is successful.

An important piece of code in the test method is the part where we recover the WebElement from the driver.

The line of code written is: `WebElement elem = driver.findElement(By.name("q"));`

Here, we make use of the `findElement()` method of the web driver api.

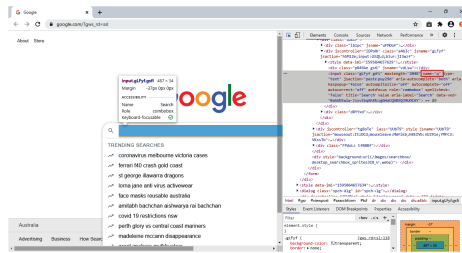
We use the 'By' class of the Selenium framework to locate an html element from the web page with the name 'q.' The By class is a class that consists of methods that can be used to find elements within a web document or page. In this example, we have searched for a web element on the Google search home page with the 'name' attribute 'q.' This attribute is used to specify a name for the element on the web page. The class 'By' consists of different locators; one of them is the 'name' locator. We make use of this locator in the test method to find the textbox where the input parameters can be entered.

The next important part hidden inside this line of code is the value ‘q’ that has been passed to ‘name’ locator. Let us see how we arrived at this value.

The goal of the test method was to type in search parameters automatically into the search box via selenium. Hence, it was necessary to identify the web element that corresponds to the search box.

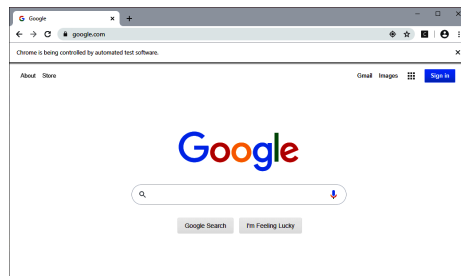
To identify the corresponding element, we open the Google homepage. We click on F12 to inspect the elements. We use the mouse to scroll over the search box which directs us to its source code where we find the name of the element corresponding to the search text box.

This is shown below:

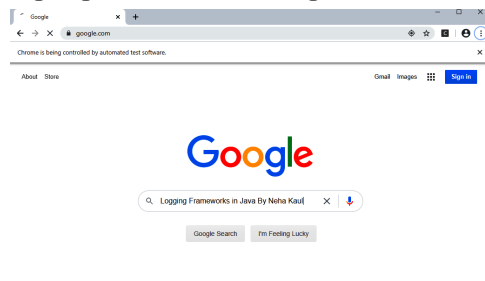


As we can observe, the value of the ‘name’ attribute of the textbox that accepts search parameters is ‘q’ which is what we use in the line of code mentioned above.

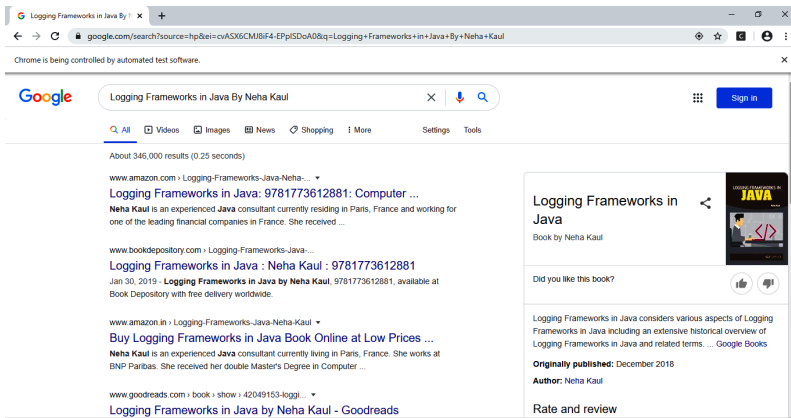
We now proceed to execute this test. It opens the Google home page.



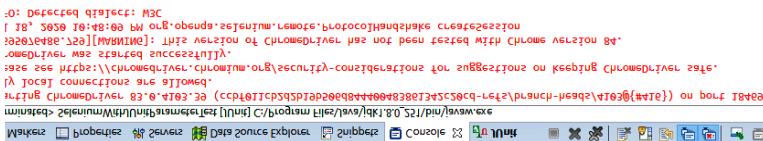
It types in the input parameter that we provided in the test method.



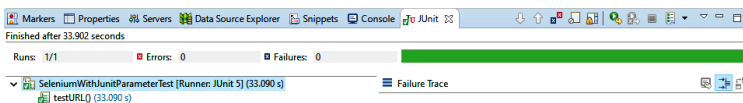
It displays the results of the search as follows:



It closes the window automatically. The console output is shown below:



The JUnit result is as shown below:



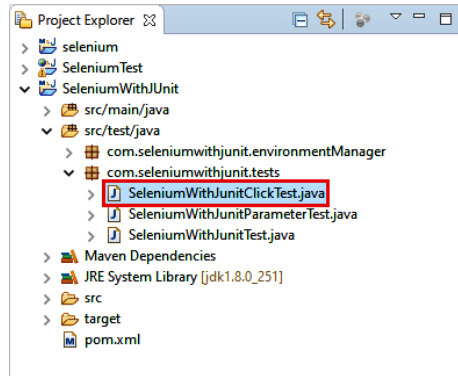
As we can see, the test executes successfully.

This example shows how efficient Selenium is and how effective it is in terms of automation. We were able to automatically open a website, type in a search string, observe the results, and then verify the contents of the title. This is still a very simple example, but one can imagine all the things that can be automated in case of a complex web application.

- Example 4: Selenium-JUnit Test Case Implementing Click Functionality

In this example, we write a test case that clicks on a link present in the Web page that is opened via the web driver API.

To demonstrate this functionality, we create a new test case in the 'tests' package shown below:



The class is as follows:

```

package com.seleniumwithjunit.tests;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

import com.seleniumwithjunit.environmentManager.EnvironmentManager;
import com.seleniumwithjunit.environmentManager.WebDriverProvider;

import static org.junit.jupiter.api.Assertions.assertTrue;
import java.util.concurrent.TimeUnit;

public class SeleniumWithJUnitClickTest {

    @BeforeEach
    public void initializeDriver() {
        EnvironmentManager.initializeDriver();
    }

    @Test
    public void testLinkClick() throws InterruptedException {
        //obtain the WebDriver
        WebDriver driver = WebDriverProvider.getWebDriver();

        //apply a wait time
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

        //maximize the window
        driver.manage().window().maximize();

        //open the Chrome browser with google URL
        driver.get("https://www.google.com");

        //finding the web element using name locator
        WebElement elem = driver.findElement(By.name("q"));
        elem.sendKeys(new String[]{"Logging Frameworks in Java By Neha Kaul"});
        elem.submit();

        Thread.sleep(5000);
        driver.findElement(By.linkText("Neha Kaul")).click();
        Thread.sleep(5000);

        //retrieve the title from the driver
        String title= driver.getTitle();
        //check if the title contains the search parameter
        assertTrue(title.contains("Neha Kaul"));
    }

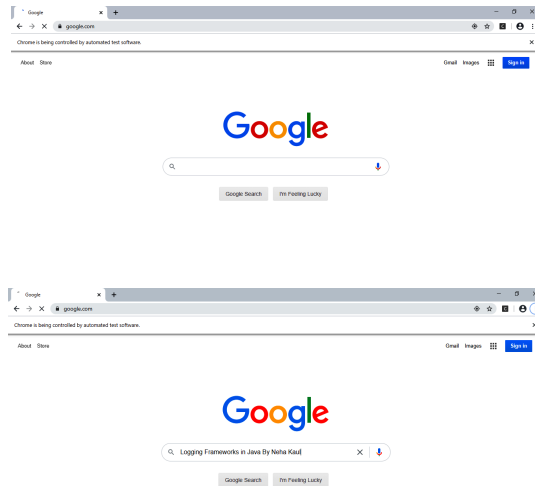
    @AfterEach
    public void shutdownDriver() {
        EnvironmentManager.shutdownDriver();
    }
}

```

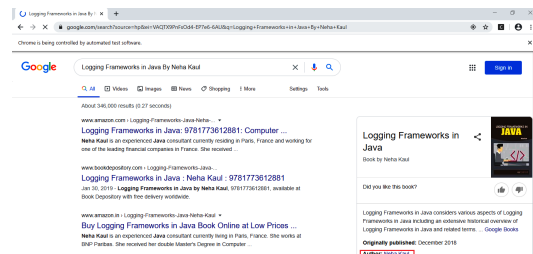
In the test method `testLinkClick()` of the test class `SeleniumWithJUnitClickTest`, we first obtain the web driver and open the Google homepage. We maximize the opened window and type in the search parameters that we used in the previous example.

Now, we proceed to make use of the `linkText()` API provided by the class 'By' in the selenium framework. This locator helps determine the hyperlinks that are present on the web page. This method accesses the link using the exact text that is present in the link. In case of our test method, we look for a link with the exact text 'Neha Kaul' in it.

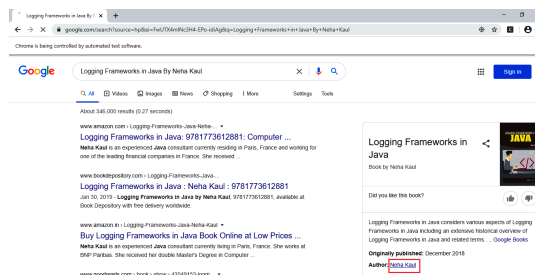
We run the test class as a JUnit Jupiter test case. On execution, the Chrome browser is opened, and the search parameters are entered in the browser.



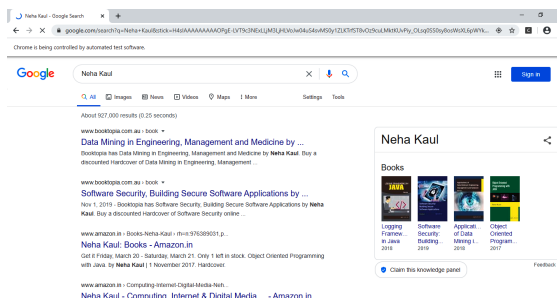
The results of the search are displayed. We have programmed the test case to click on a link with the exact text ‘Neha Kaul.’ This is highlighted in red below:



As we have added a sleep statement in the code, we are able to see the moment when the link is identified and then subsequently clicked.

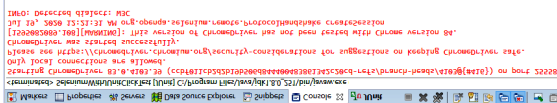


As seen from the image above, the link with the text ‘Neha Kaul’ is underlined right before it is clicked. This means that it has been identified as the link configured in the script and it will be clicked.

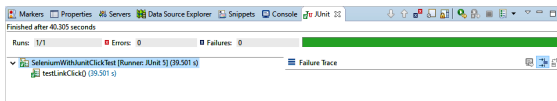


The link is clicked, and the results of clicking on the hyperlink text are displayed as shown in the screenshot above.

The console output is as shown below:



The result of the JUnit test case is as follows:



The test has passed successfully.

This example effectively demonstrates that selenium can be used to perform several activities on web browsers with ease. Links on a web page can be easily accessed by means of a simple locator.

It should be noted that if the locator locates several links on the web page with the same text, it will automatically select the first link that it finds. The API will choose the very first result and this cannot be changed unless more specific search parameters are provided to the locator.

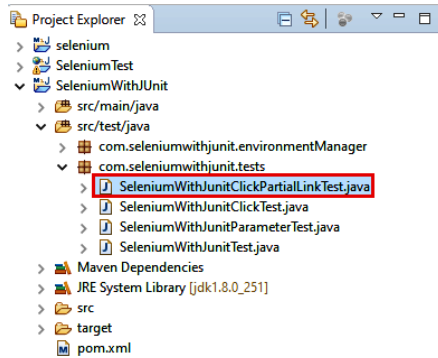
Sometimes, there could be a situation where the exact text of the link to be clicked is not known. If we do not know the exact text of the link that we need to click on, we can make use of the `partialLinkText()` method.

This is demonstrated in the next example.

- Example 5: Selenium-JUnit Test Case Partial Link Click Functionality

In this example, we will cover the use of the API `partialLinkText()`. As we have seen in the previous example, there is a possibility of configuring the test script to click on a hyperlink text. But, often the text of a hyperlink is long, and it is not possible to know the entire text of the link correctly. This is where the `partialLinkText()` API is useful.

It helps us match parts of the hyperlink text. To test it, we proceed to create a new test class in our src/test folder as shown below:



The contents of our test class are:

```

SeleniumWithJUnitClickPartialLinkTest.java
package com.seleniumwithunit.tests;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

import com.seleniumwithunit.environmentManager.EnvironmentManager;
import com.seleniumwithunit.environmentManager.WebDriverProvider;

import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.concurrent.TimeUnit;

public class SeleniumWithJUnitClickPartialLinkTest {

    @BeforeEach
    public void initializeDriver() {
        EnvironmentManager.initWebDriver();
    }

    @Test
    public void testPartialLinkClick() throws InterruptedException {

        //obtain the WebDriver
        WebDriver driver = WebDriverProvider.getWebDriver();

        //apply a wait time
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

        //maximize the window
        driver.manage().window().maximize();

        //open the Chrome browser with google URL
        driver.get("https://www.google.com");

        //finding the web element using name locator
        WebElement elem = driver.findElement(By.name("q"));
        elem.sendKeys(new String[]{"Logging Frameworks in Java By Neha Kaul"});
        elem.submit();

        Thread.sleep(5000);

        JavascriptExecutor js = (JavascriptExecutor)driver;
        js.executeScript("scrollBy(0, 100)");

        Thread.sleep(4000);

        driver.findElement(By.partialLinkText("www.goodreads.com")).click();

        Thread.sleep(5000);

        //retrieve the title from the driver
        String title = driver.getTitle();
        //check if the title contains the search parameter
        assertTrue(title.contains("Neha Kaul"));
    }

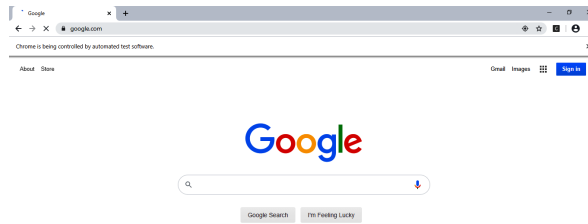
    @AfterEach
    public void shutDownDriver() {
        EnvironmentManager.shutDownDriver();
    }
}

```

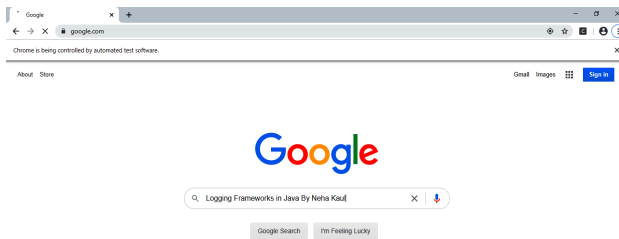
As we can see, the code different only slightly from the previous example. Here, we make use of the By class's partialLinkText() API to provide a partial link.

We execute the test class:

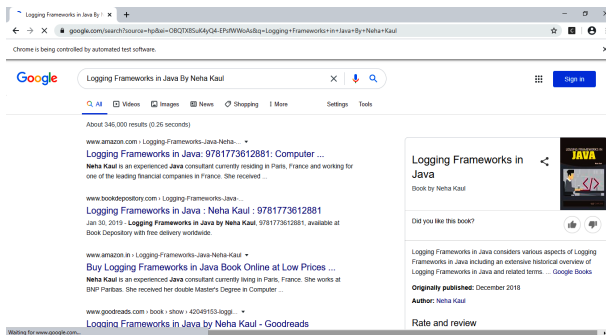
The Google search page is opened.



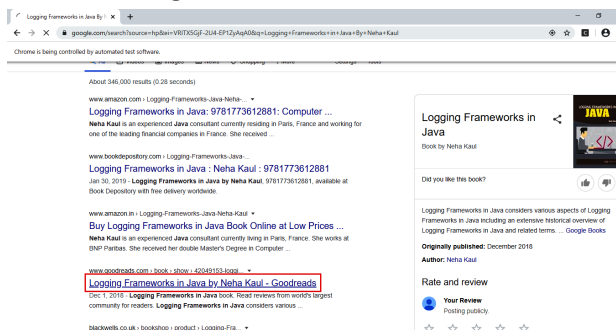
The search text is entered automatically:



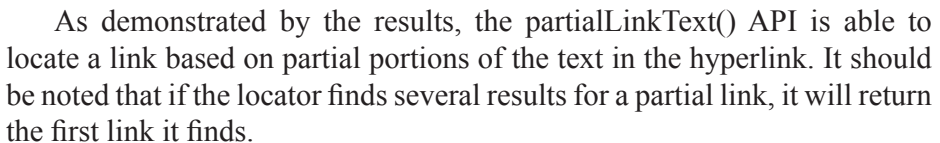
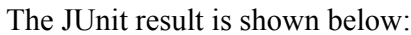
The search results are displayed as follows:



The text provided in the partialLinkText invocation is searched and highlighted before clicking on it.



The desired web page is opened as shown below:



6.4. MOCKITO FRAMEWORK

6.4.1. What Is Mocking?

As we have seen, testing of any type is to ensure that the object/module performs the way it is designed to function. One of the commonplace methods of testing is unit testing which seeks to check small units/modules of a system for its correctness in behavior and functioning. As unit, testing

evolved, different frameworks that simplified this process started coming up in the open source market. Developers and innovators created unit testing and automation testing frameworks. Such automation frameworks are designed to be used in a way that exercises all the expected behaviors of a system and then verifies that the system behaves as expected. But, with complex systems, the individual components may be dependent on other components which impact the tests directly. The tests may need to be dependent on more than one unit which tends to make the tests slightly disorganized. It is preferred that the test cases be dependent on just one module rather than a combination of several units or modules. This is where the concept of ‘mocking’ or ‘mock objects’ comes in place.

Mocking is a technique of testing in which the real objects are replaced with objects that are called mock objects which behave the same way as the object being replaced. The replacement objects have a predefined comportment which is the same as the original object. These objects are created only for use in the test cases that they are meant for. In simpler terms, a mock object is a programming object that is programmed to return a specific value/output for a specific input without performing any tangible action. A mock object appears to behave and act as the real object and perform all the tasks meant to be done by the real object. The concept of ‘Mocking’ signifies the imitation or mimicking the behavior of the original object.

The objects came into existence to mimic the behavior of the real object in a controlled manner. These objects are created programmatically as a simulation of the real object to write a test that verifies the behavior of another object. This is because when the context of a test is a complex unit, there is a chance that the individual unit may have to be dependent or interact with other modules.

There can be association between modules which complicates things when writing test cases. Hence, when we test a distinct component, we may actually test the component and its relationship with other modules or units as well. It is important to test just the unit under consideration so that one can stay focused on the unit. This is remedied by use of mocks as it helps simulate or mimic the units that the unit under test depends on. The mock objects fake the external dependencies of the module and maintain the consistency of the object’s behavior. The mock object is designed in a way that it returns dummy information/results back to the test which helps writing the tests easily (Grzejszczak, 2014).

6.4.2. Why and When Should We Use Mocks?

Mock objects offer a stand-in for the real object. Mocking should be used to simplify writing test cases and to focus on testing one unit rather than an interaction of different units. Mocking can be used in several scenarios. This concept is especially handy when we have a component that depends on another component, but it has not been developed yet. This scenario occurs often. In software projects, there is a chance that the other component is still in the process of being built. In such cases, a developer remains blocked in his testing until the other component is ready. This happens quite often when working in teams, where there is a high probability that one component's development is divided among several developers. Without the concept of mocking, the developer would have to wait until the other developer/developers are done. But mocking the object would help the developer finish his module and then move on to something else. Mocking the necessary dependency would give him the required objects to complete his tests. It helps develop unit tests rapidly and help test a component in isolation. This would not only save time but also improve the quality of the project as coupling would be reduced.

Interdependency on other modules is a major cause of concern for developers. A developer usually wishes to verify his own code and move on to the next piece of code. He executes his unit test on a regular basis as the code keeps developing over a period which is the case with most projects implementing continuous integration. But sometimes, the test can fail due to a bug in a dependent module. Now, the developer's test case fails due to another component and he is stuck waiting for the issue to be resolved. It could also be that the developer would think that there is an issue in his code and some confusion could arise. In large teams implementing complex projects, where the developers are separated geographically, it sometimes becomes difficult to ensure smooth communication. This could prevent the developer from identifying the issue in the dependent module and he may waste a lot of time investigating the issue. Mocking the dependent module would help him significantly reduce the time and effort spent on the test cases.

Another scenario when one can use mock objects is when I/O operations are needed. When a unit has an I/O operation such as a disk read/write, the time taken to execute the tests is high. It is a known fact that operations that access the disk for read/write purposes tend to take up a lot of time. Similarly, when a unit connects to any type of database and queries it, the

time needed for the connection followed by the time needed for the execution/query is a lot. If we imagine a test suite with over 10 classes where each test class/package connects to a database instance and queries it for each test case, we can calculate that the tests would take a significant amount of time to execute. It is a known fact that in pre-production and production environments, database connections, and queries takes up more than 10–20 seconds for read/write operations. Therefore, if the component under test is performing an operation that is bound to be slow, it is desirable to reduce the amount of time spent on executing the test case. It would be a waste of time to wait for our test to execute for several minutes while the database is being accessed. This time could be spent on other important activities such as writing more tests, adding ‘nice-to-have’ features, and improving the product quality. This is where mocking can be of great help. It would help reduce the complexity of the test case and execute it faster.

Similarly, when servers such as an application server for instance, are being used by the system, the time of execution is higher. This situation can benefit from the use of mocking as well.

Another situation where mocking can be helpful is when the configuration or setup needed for the unit is not available for some reason. For example, if a unit/component queries a remote service hosted on a remote server, and if this server is non-functional for some reason, the test case would fail. The remote server may not have been acquired yet as well. This happens in software projects at times. Sometimes, the development has been started without acquiring the entire environment infrastructure. What happens is, the environment for production is given higher priority than an environment meant for developer testing. This means that there can be delays in acquiring the servers for the development environments. In such cases, mocking is of substantial assistance to the developers who are working on such unit. It saves them a lot of time and effort that would have been spent in waiting for a server, configuring it, and then executing the test.

The same applies for a database/server that has not been configured yet. There can be times that although the infrastructure is in place; the configuration of the infrastructure has not been done yet for some reason. Such situations can benefit from the use of mocks. The use of mocking makes the developer independent from other modules and external infrastructural/configurational issues. Mocking should be used when its use simplifies the process of testing. Mockito is one of the many frameworks available for mocking objects in Java.

Now that we have seen how mocks can help in the process of testing, we will study API's and annotations and useful features provided by the Mockito framework.

6.4.3. Mockito Framework Details

The Mockito Framework is a favored mocking framework among the several mocking frameworks that are available in the market. This framework provides helpful APIs that are used to create mock objects. The API provided by this framework help create mock objects or 'mocks' for the classes and interfaces that a class/method under test uses or interacts with in a simple fashion. The API provided by this framework is simple and easy to use and are intuitive in nature. The syntax of the API is simple and straightforward making it easy to learn and implement. It is uncomplicated to create mock objects and assigning them behavior using this framework. The learning curve is very short and linear when it comes to this framework. This framework offers a lot more than simple mocking and has features that prove to be useful in case of complex scenarios. The ability to rapidly create mocks, specify their behavior and then verify the interactions makes Mockito a powerful framework for mocking.

This framework uses the Java Reflection API internally and permits the creation of objects that serve as mocks. This mock object generated by the framework by use of reflection returns data or a result that is dummy data and is independent from any kind of external module dependencies. The use of the Reflection API help generate mocks are technically sound, thereby simplifying the development of unit tests by mocking exterior dependencies and applying the mock objects to the code/unit under test.

In general, when we make use of mocking in the development of our unit test cases, two phases are followed:

- **Stubbing:** This is a process where one specifies the behavior of the mock object. The act of stubbing is the way me inform the framework what to do when one interacts with the mock object. This helps define the behavior and response of the mock objects and methods. It is a way of handling how the mocks work. It helps the user control and oversee the functioning of the mock objects. We can program and manage the responses of the mock objects. The mock objects can be configured to return the value we desire, to throw an exception or throw an error. Stubbing permits us to configure different types of behaviors and controls in our mocks

and also add different conditions to it. In brief, this process helps control the functioning of the mock objects.

- **Verification:** The process of verification is checking the interactions of other objects and classes (usually test classes and objects) with our mocks. This process helps us understand how the mock objects were referenced and called and used. It helps calculate the number of times they were called and referenced. This process aims to examine the arguments of the implemented mock objects and ensure that they are as designed and expected. This operation helps in ensuring the functionality of the mock objects. One can ensure that the values passed on or returned by the mock objects are as expected and that no error is present in the functioning of the mock. This procedure is meant to help us determine what happened to the mock objects and how it was used.

The combination of the two phases of stubbing and verification helps build simple and flexible unit test cases by means of using the API provided by Mockito. Encoding of complex behavior into mock objects and verification of interaction with the complex mock objects is made possible by combining these two phases.

Before moving on to practical implementations of this framework, we cover some basic terminology and annotations that anyone implementing mocks should be aware of.

6.4.3.1. Terminologies and Mockito Annotations

In this section, we cover some basic terminology used when working with mocking frameworks and annotations that are used in Mockito:

1. **Terminologies:** As we have covered, a unit test generally indicates the testing of a single module. This means that the test should just check the functionality expected of the unit in complete isolation from other modules if possible. Any potential byproducts or negative impacts originating from other classes or methods or other systems should be eliminated when writing unit tests.

This elimination of unwanted dependencies and variables can be done by use of replacement objects that mimic the behavior of the original external dependencies. There exist different types of test doubles, one of which is a mock object. Other test doubles include-dummy objects, fake objects and

stub objects respectively. With the goal of better understand mock objects, we must also understand these other objects that are akin to mocks but differ from them in some way.

- A dummy object is an object that is referenced and passed here and there within the code but is never actually used. Such an object usually does not have a concrete implementation. The methods of such objects are never called. For instance, an example of a dummy object is an object that helps fill the list of parameters of a method. These parameters are never going to be used, just accessed as a part of the method.
- A fake object is slightly more developed than a dummy object. Such an object has some sort of concrete implementation. But even though such an object has an implementation, the implementations are very simple and basic in nature. A simple example of a fake object is a fake database (usually memory database H2) which is used for the unit tests rather than the real database such as SQL or Dynamo, etc. (Acharya, 2014).
- A stub object or class, like a fake object has a corresponding implementation associated to it. But this implementation is a partial implementation of the class/methods/interface. This partial implementation serves as a piece of code whose instance will be used during the test process. A stub does only what it is programmed to do. A stub's implementation context is limited to the tests it is needed for. Stubs can also be configured to record or note information regarding the amount of times it was referenced and the object that referenced it.
- A mock object is nothing but a dummy realization of a class or a method or an interface. A mock object permits you to define the behavior of certain methods including their output. Objects of this type are programmed to function in a particular manner for the purpose of the test. Such objects can be configured in any way that we want. Generally, mock objects record and possibly log their interaction with the software system under test and are also able to verify such interactions (Acharya, 2014).

These different types of test doubles can be instrumental in the process of testing. Such objects can be passed on to other objects which can then be tested in isolation. The test class or method can perform the task of verification and validation of the behavior of these objects. The tests can

check if the mock objects are functioning correctly. Additionally, one can check if distinct methods belonging to the mock objects have been referenced or not. Such practices help ensure that the test method tests the unit correctly along with the mock objects. The test classes/method ensures that the mock objects behave as they were supposed to and verify that no undesirable results are observed.

2. **Annotations:** In this section, we cover some basic Mockito annotations.
 - **@Mock:** This annotation is used to denote creation of a mock object. Any variable (local or member) that is annotated using this annotation will be mocked. This annotation allows us to create a mock using shorthand and makes the test class more readable. The process of verification is easier when using this annotation as the name of the field is used for identification of the mock object (Mock-mockito-core 3.4.4 Javadoc, 2020). This annotation can be used instead of the `mock()` API method.
 - **@Spy:** This annotation is used to denote the creation of a spy object. A spy object in Mockito is a mock, but it is a partial mock object (Spy (Mockito 3.4.4 API), 2020). A spy object basically wraps around an existing/real instance of the object. It then manipulates this new instance with the addition that it can now track the interactions it has. In simple words, it will do and behave as the real object with the ability to spy on it. We can invoke all the regular methods of the object and track the interactions which are efficient. This annotation is the shorthand and concise alternative to using the `spy()` API method.
 - **@InjectMocks:** This annotation is used to be used to mark a particular field/object and direct it to instantiate the object with all the fields that have been annotated with either the `@Mock` or `@Spy` annotations. A field denoted with this annotation will inject dependencies marked with the `@Spy` and/or `@Mock` annotations into itself. This annotation permits injection of mocks and spies using the shorthanded annotation. It helps reduce repetitive instantiation of mock and spy objects.
 - **@Captor:** This annotation is used to create an instance of the

type `ArgumentCaptor`. The `ArgumentCaptor` class is used to capture the values of arguments of the objects passed to it so that they can be used for further assertions. Mockito makes use of the `equals()` method for its assertions, which is the recommended way of matching values in assertions as the code is rendered clean, simple, and easy to understand. This is also the recommended way of matching arguments because it makes tests clean and simple. The `@Captor` annotation is the shorthand for creation of an instance of this type (`Captor` (Mockito 3.4.4 API), 2020).

It should be noted that although Mockito is an exceptional tool, it does have a few limitations when it comes to implementations.

Some of the limitations of Mockito are:

- Inability to mock methods that are static;
- Inability to mock methods that are final;
- Inability to mock the `equals()` method;
- Inability to mock the `hashCode()` method;
- Inability to mock classes that are declared as final.

Now that we have covered some basic terminology and annotations in Mockito, we shall delve in the practical implementations of the Mockito framework in the next section.

6.4.4. Practical Implementations

In this section, we will be looking at the setup for installation and integration of Mockito in your Java project followed by examples that demonstrate the use of the framework.

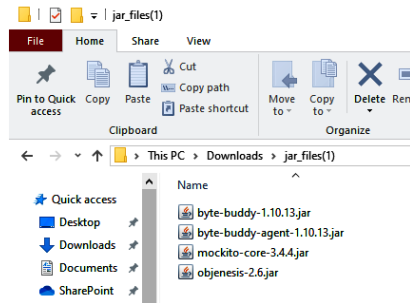
6.4.4.1. *Mockito Setup*

The Mockito framework can be added to any project in several ways. We shall be covering the following couple of standard ways of doing this.

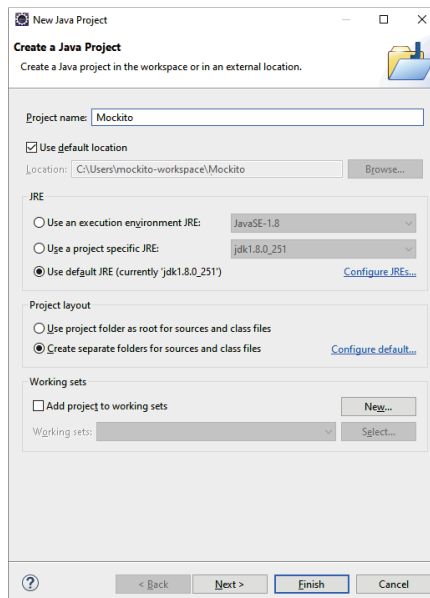
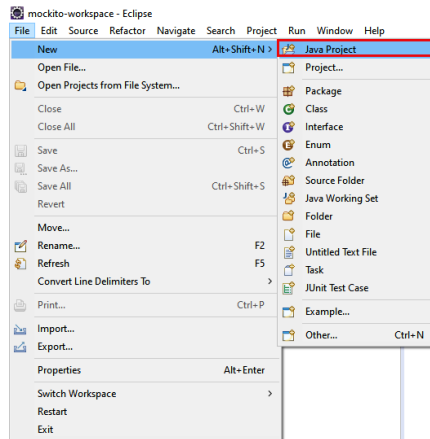
1. **Adding the Jars Directly to the Project:** The latest stable version of Mockito is 3.4.4. This is the version that we will be using in our examples. First, we navigate to the following site to download the required jar files:

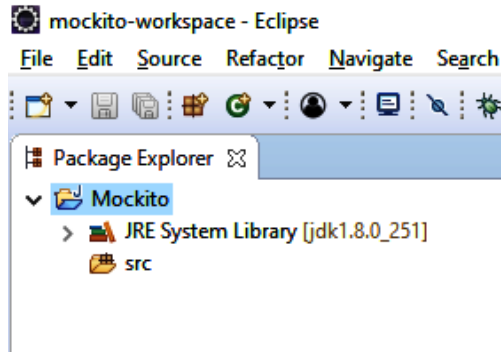
<https://jar-download.com/artifacts/org.mockito>

We unzip the downloaded compressed folder.

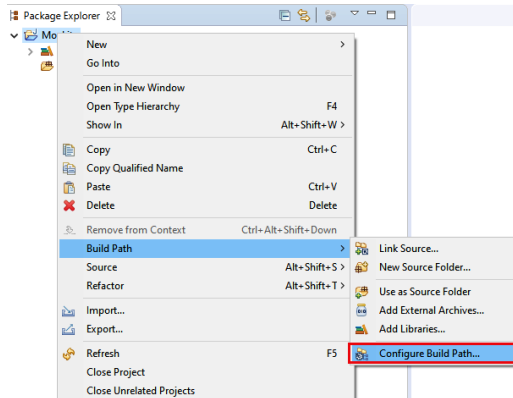


We then proceed to create a new Java Project as shown:

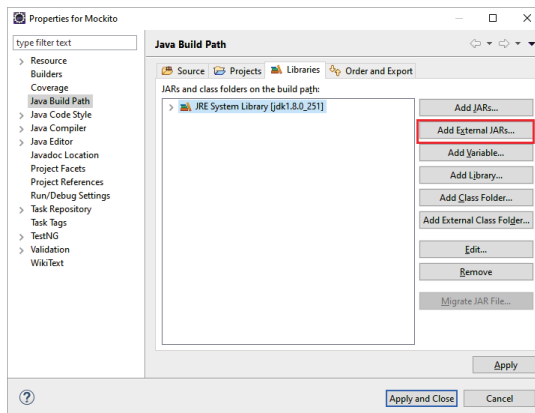




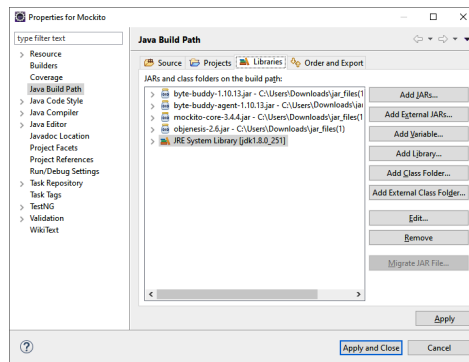
Once the project has been created successfully, we right click on the project and navigate to Build Path -> Configure Build Path as shown below:



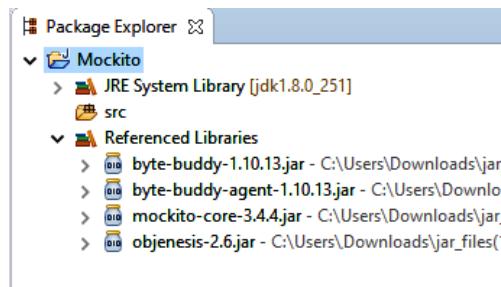
The following window opens:



To add the downloaded jar files, we click on the 'Add External JARs' button and navigate to the folder where we extracted the jar files and add them.



The jar files have been added to the libraries of the project as seen. We click on the 'Apply and Close' button to finalize the addition of the jar files to our project. This is shown below:

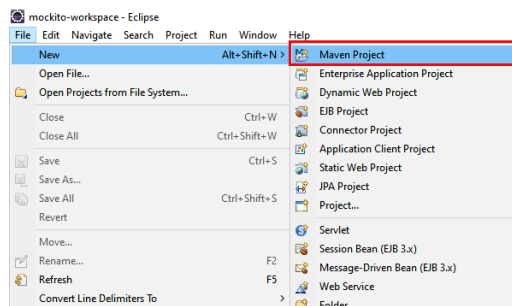


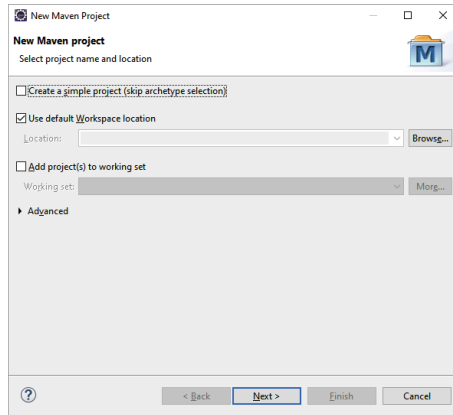
The required dependencies for Mockito have been successfully added to our project as seen from the image above.

2. **Use of Maven to Inject Dependencies:** Maven is a powerful tool that helps in automation of the build process for Java projects. We can make use of maven to add any required dependencies to our project.

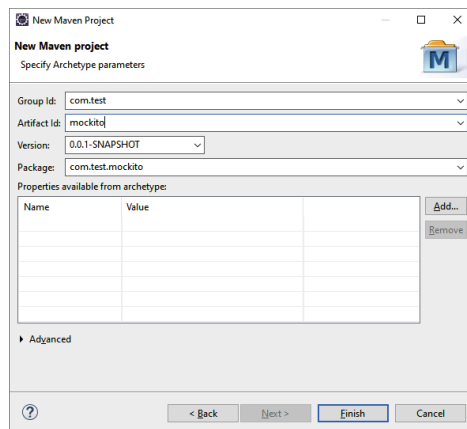
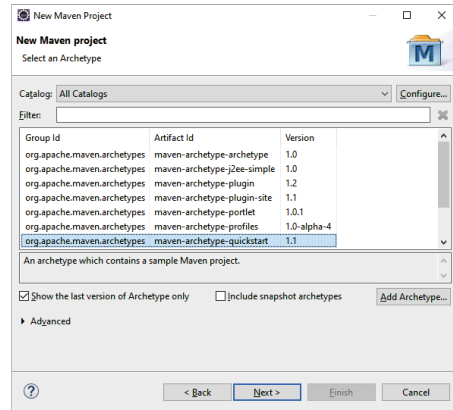
The steps to setup Mockito using maven are as follows:

We first create a maven project as shown below:

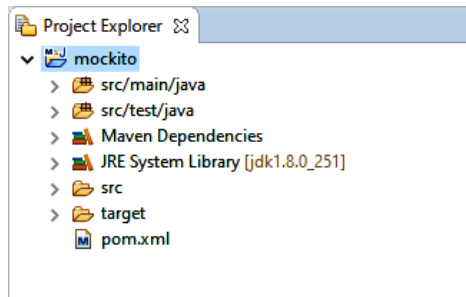




We opt to create a simple quick start maven project.



Once we finish the creation, a simple maven project with a src folder, test folder, target folder, and pom.xml is created as follows:



To add the Mockito dependencies, we proceed to edit the pom.xml file and add the dependency for Mockito as shown below:

```

mockito/pom.xml
<?xml version="1.0" encoding="UTF-8" ?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.test</groupId>
  <artifactId>mockito</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

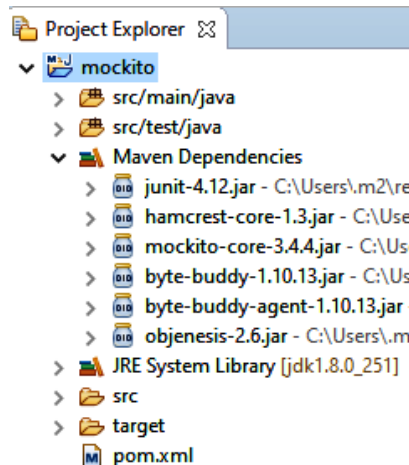
  <name>mockito</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>3.4.4</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

For the purpose of the setup, we have added JUnit4 to our pom.xml as well. Once the pom.xml file is edited, we build the project. The dependencies we specified in the pom.xml file have been added automatically to the project as shown below:



As seen from the screenshot, we have 4 jar files for Mockito and 2 jar files for JUnit4. Now that we have seen how Mockito can be set up in our project, we delve into some practical examples of its use in the next section.

6.4.4.2. Practical Examples of Mockito

In this section, we will explore the API and annotations provided by the Mockito Framework.

- Example 1: Simple Mockito-JUnit Test

In this example, we create a simple test using Mockito and JUnit5. We create a maven project with the required dependencies for Mockito and JUnit5.

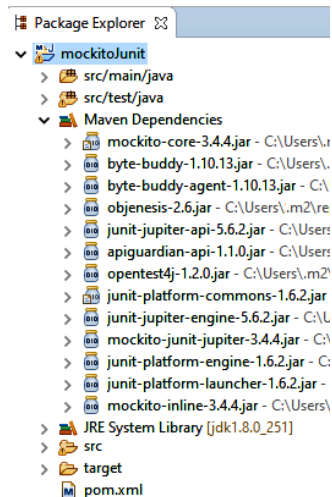
The pom file of this project is as shown below:

```

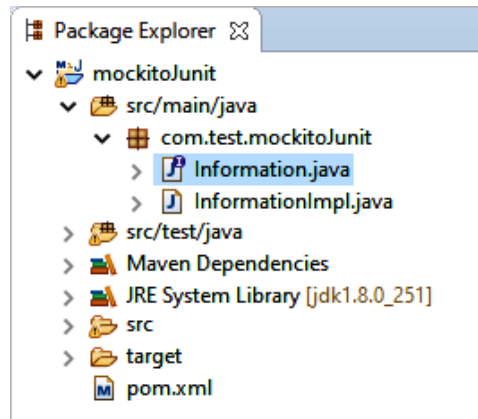
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.test</groupId>
  <artifactId>mockitoJUnit</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>mockitoJUnit</name>
  <url>http://www.mockito.org/</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <mockito.version>3.4.4</mockito.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>${mockito.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.6.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.6.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-junit-jupiter</artifactId>
      <version>${mockito.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-engine</artifactId>
      <version>1.6.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-launcher</artifactId>
      <version>1.6.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

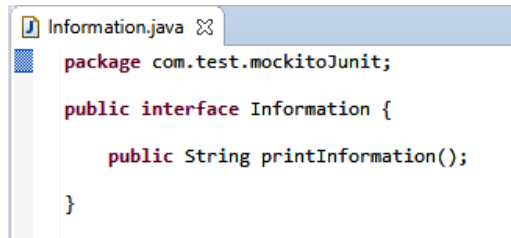
Once we build this project, the required dependencies for JUnit5 and Mockito version 3.4.4 are added to the project as shown below:



To demonstrate the use of mocking, we consider a simple interface and its implementation as follows:



The interface has a single method that returns a string value.



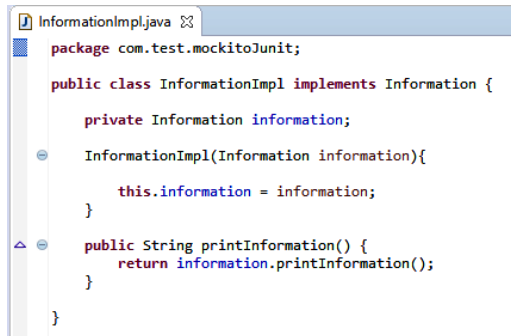
```
package com.test.mockitoJUnit;

public interface Information {

    public String printInformation();

}
```

The implementation of the interface is shown below:



```
package com.test.mockitoJUnit;

public class InformationImpl implements Information {

    private Information information;

    InformationImpl(Information information){

        this.information = information;

    }

    public String printInformation() {

        return information.printInformation();

    }

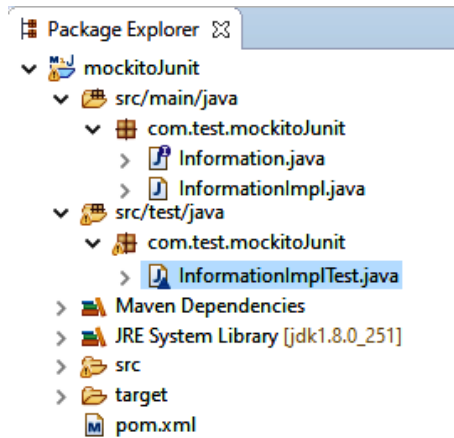
}
```

As we can see, the implementation class has a constructor and the interface method implementation.

It should be noted that no concrete implementation of the `printInformation()` method is present.

We are going to mock/simulate the behavior of this method.

We proceed to create a test case to test this implementation. We create a new test case named `InformationImplTest` in the `src/test` folder as shown below:



The contents of this test class are:

```

package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

class InformationImplTest {

    private Information information;
    private InformationImpl informationImpl;

    @BeforeEach
    void setUp() {
        information = Mockito.mock(Information.class);
        informationImpl = new InformationImpl(information);
    }

    @Test
    void testPrintInformation() {
        Mockito.when(information.printInformation()).thenReturn("How are you?");
        assertEquals("How are you?", informationImpl.printInformation());
    }
}

```

The test class has 2 methods: one method is the setUp() method. In this method, we mock the interface Information. We pass this value to the default constructor of the InformationImpl class. We make use of the mock() API provided by Mockito to mock the behavior of the interface.

The second method is the test method testPrintInformation() method where we write our test using the mock object. We make use of the 'when-thenReturn' construct which is the standard way to use mocks in Mockito. These two methods are used in combination to define the behavior of the mock object.

The when() API permits us to stub methods. We make use of it to use the mock object. It is used to define a condition/scenario when the mock is invoked, fulfilled by the mock method.

The thenReturn() methods allows us to define and configure the return value when a particular method of the object that is mocked has been called.

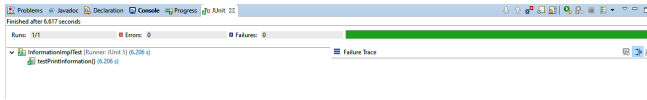
In simpler terms, the when-then Return construct translates to:

“When the x method on the mock is called then return y value” (Mockito (Mockito 3.4.4 API), 2020).

In our example, we configure our mock in such a way that when the `printInformation()` is called, the value “How are you?” is returned.

We make use of an assertion to check if the value returned by the mock matches the value configured.

We execute the test as a JUnit test. The results are:



As observed from the test results, the test case has passed successfully.

In the next example, we will look at the use of the ‘Mock’ annotation.

- Example 2: Mockito-JUnit Test With @Mock Annotation

In this example, we make use of the ‘Mock’ annotation instead of the `mock()` API method.

The test class is the same that we use before, but with minor changes.

```

package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;

class InformationImplTest {

    @Mock
    private Information information;
    private InformationImpl informationImpl;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
        informationImpl = new InformationImpl(information);
    }

    @Test
    void testPrintInformation() {
        Mockito.when(information.printInformation()).thenReturn("How are you?");
        assertEquals("How are you?", informationImpl.printInformation());
    }
}

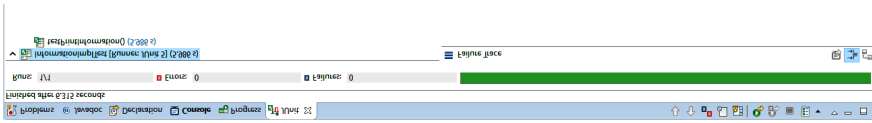
```

As seen from the above example, we have replaced the statement calling the `mock()` API and simply added the mock annotation to the declaration of the field ‘information.’

But, simply using the ‘Mock’ annotation is not enough. The object needs to be initialized too. We can achieve this by using the `openMocks()` api present in the `MockitoAnnotations` class.

This API method is used to initialize any objects that have been annotated with any of the following annotations for the given test class: `@Mock`, `@Spy`, `@Captor`, `@InjectMocks` (MockitoAnnotations-mockito-core 3.4.4 Javadoc, 2020).

We initialize the mock object in the setup method and execute the test class:
The results of execution are:



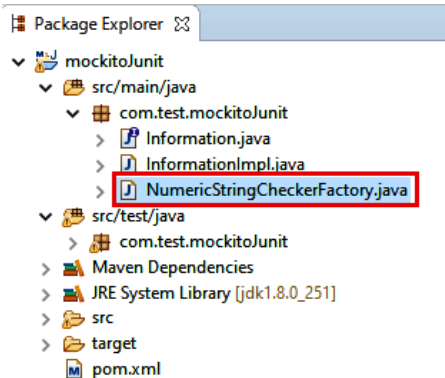
The test is executed correctly as seen above.

- Example 3: Mockito-JUnit Test with a Parameter

In this example, we test a method that expects a parameter and mock its behavior using Mockito.

We consider the following class that checks whether a string contains an integer or not.

The class is present in the src/main folder as shown below:



The contents of the class are:

```

NumericStringCheckerFactory.java
package com.test.mockitoUnit;

public class NumericStringCheckerFactory {

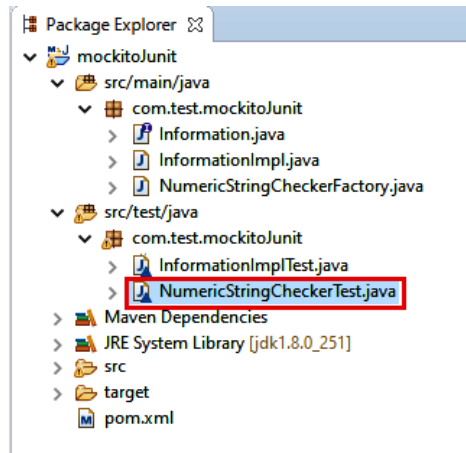
    public int isNumericString(final String str) {
        if(str == null || str.isEmpty()) return 1;

        for(char c : str.toCharArray()){
            if(Character.isDigit(c)){
                return 0;
            }
        }
        return 1;
    }
}

```

The method `isNumericString()` returns 1 if there is an issue or if the string does not contain an integer. Else, it returns 0.

We want to mock this behavior. To do this, we create a test class in the `src/test` folder as shown below:



The contents of this class are:

```

package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import static org.mockito.ArgumentMatchers.anyString;

class NumericStringCheckerTest {

    @Test
    void testNumericString() {
        NumericStringCheckerFactory checkerFactory = Mockito.mock(NumericStringCheckerFactory.class);
        // define return value for method isNumericString()
        Mockito.when(checkerFactory.isNumericString(anyString())).thenReturn(0);

        // use assertion to test the functioning
        assertEquals(checkerFactory.isNumericString(anyString()), 0);

        //verify the mock
        Mockito.verify(checkerFactory).isNumericString(anyString());
    }
}

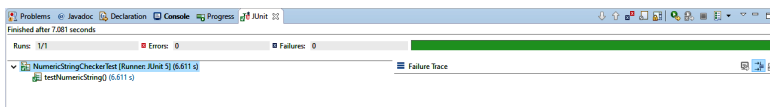
```

In the test class, first, we create a mock object of the class that we wish to Mock. We then define the behavior when the `isNumericString()` method is called using the `when-thenReturn` construct.

We use an `Argument Matcher` to pass a parameter to the `isNumericString()` method. The `anyString()` argument matcher is used to denote that any non-null string can be passed to the method.

We configure the method to return 0 when any string is passed. We proceed to use an assertion to check if 0 is returned. We use of the `verify()` method provided by Mockito to check if the mock object has been called or not.

We execute the test. The results are:



The results clearly show that the mock has been called and that it returns 0 when any string is passed to it.

- Example 4: Mockito-JUnit Test Invocation Count

In this example, we make use of the `times()` API to check the number of times a mock object has been called. We take the same test class as used in Example 3.

We modify the `verify` statement to use the `times()` API to check whether the `isNumeric()` method of the mock object has been called once.

This is shown below:

```

NumericStringCheckerTest.java
package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.Mockito.times;

class NumericStringCheckerTest {

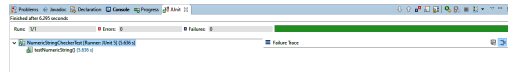
    @Test
    void testNumericString() {
        NumericStringCheckerFactory checkerFactory = Mockito.mock(NumericStringCheckerFactory.class);
        // define return value for method isNumericString()
        Mockito.when(checkerFactory.isNumericString(anyString())).thenReturn(0);

        // use assertion to test the functioning
        assertEquals(checkerFactory.isNumericString(anyString()), 0);

        // verify the mock
        Mockito.verify(checkerFactory, times(1)).isNumericString(anyString());
    }
}

```

We execute this test. The results are:



As we notice from the results, the mock object invokes the `isNumericString()` method exactly one time.

- Example 5: Using Spies in Mockito

In this example, we will study the use of spies via the `spy()` API and the `@Spy` annotation.

We consider a simple `ArrayList` in our test class. We proceed to mock this list using the `spy()` API. We then manipulate the spy object and add data to it. We then verify that the data has been added to it correctly. Lastly, we use an assertion to verify the new size of the `ArrayList`.

The test class is as follows:

```

MockitoSpyTest.java
package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;
import java.util.List;

import org.mockito.Mockito;
import org.junit.jupiter.api.Test;

class MockitoSpyTest {

    @Test
    void spyTest() {
        List<Integer> originalList = new ArrayList<>();
        List<Integer> spyList = Mockito.spy(originalList);

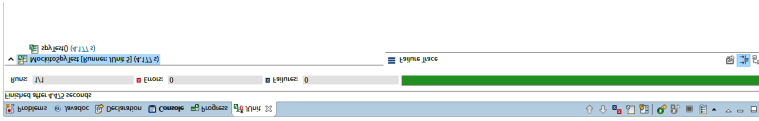
        spyList.add(1);
        spyList.add(2);
        spyList.add(3);

        Mockito.verify(spyList).add(1);
        Mockito.verify(spyList).add(2);
        Mockito.verify(spyList).add(3);

        assertEquals(3, spyList.size());
    }
}

```

We proceed to execute this test and obtain the following results:



As one can observe from the positive results, the spy object works correctly and test the functioning of the add() and size() method of ArrayList.

Now, we proceed to update the test class to use the ‘Spy’ annotation as follows:

```
MockitoSpyAnnotationTest.java
package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;
import java.util.List;

import org.mockito.Mockito;
import org.mockito.Spy;
import org.junit.jupiter.api.Test;

class MockitoSpyAnnotationTest {

    @Spy
    List<Integer> spyList = new ArrayList<Integer>();

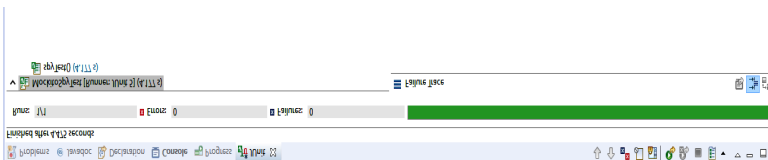
    @Test
    void spyTest() {

        spyList.add(1);
        spyList.add(2);
        spyList.add(3);

        Mockito.verify(spyList).add(1);
        Mockito.verify(spyList).add(2);
        Mockito.verify(spyList).add(3);

        assertEquals(3, spyList.size());
    }
}
```

The results of the execution are:



As we can see, both the spy() API and the ‘Spy’ annotation can be used to create spy objects with success.

- Example 6: Stubbing Spies

In this example, we take the use of spies a little further. We proceed to stub a spy object and define behavior for the spy.

We have the following test class:

```

MockitoSpyStubTest.java
package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.*;
import java.util.Map;
import java.util.TreeMap;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

class MockitoSpyStubTest {

    @Test
    public void testStubWithSpy() {
        Map<Integer, String> spyMap = Mockito.spy(new TreeMap<Integer, String>());

        //check size before stubbing
        assertEquals(0, spyMap.size());

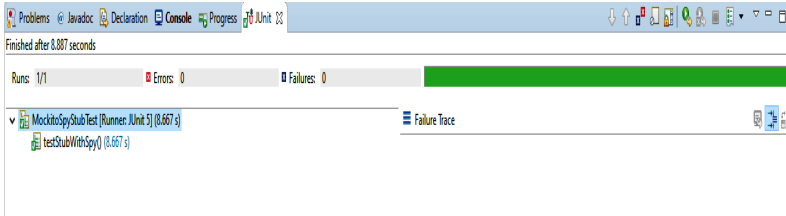
        //override the size() method of Map
        Mockito.doReturn(100).when(spyMap).size();

        //check the configuration
        assertEquals(100, spyMap.size());
    }
}

```

Here, we create a spy object of the type `TreeMap<Integer, String>`. We check its initial size which is zero. We then proceed to configure the behavior of the `size()` method. We mock it to return the value hundred upon its invocation. A simple assert statement is used to confirm that this has been done.

The results of execution of this test case are:



The test passes with success which shows that the spy object has been stubbed in a correct way.

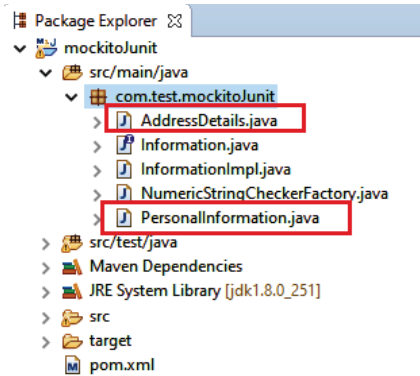
- Example 7: Mockito-JUnit Detailed Spy Example

In this example, we demonstrate the use of spies for a java class. We configure spies to spy on a method defined in a class.

We consider a class that manages the personal information of a person such as their name, age, and address. We have two classes namely `PersonalInformation` and `AddressDetails`.

`AddressDetails` is a composed variable of the `PersonalInformation` class.

The classes are present in the `src/main` folder as shown below:



The PersonalInformation class has three member variables for name, age, and address respectively.

```

package com.test.mockitoUnit;

public class PersonalInformation {
    String name;
    int age;
    AddressDetails address;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public AddressDetails getAddress() {
        return address;
    }

    public void setAddress(AddressDetails address) {
        this.address = address;
    }

    public String getPersonalInformation() {
        return "PersonalInformation [name=" + name + ", age=" + age + ", address=" + address.getAddressDetails() + "]";
    }
}

```

The AddressDetails class holds data about the address such as street name, the street number, and the apartment number. The class consists of a method getAddressDetails() which returns the address information. We will be using spies to mock the behavior of this method.

The contents of this class are:

```

package com.test.mockitoUnit;

public class AddressDetails {
    String streetName;
    int streetNumber;
    int apartmentNumber;

    public String getStreetName() {
        return streetName;
    }

    public void setStreetName(String streetName) {
        this.streetName = streetName;
    }

    public int getStreetNumber() {
        return streetNumber;
    }

    public void setStreetNumber(int streetNumber) {
        this.streetNumber = streetNumber;
    }

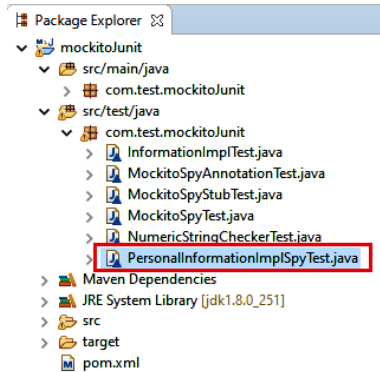
    public int getApartmentNumber() {
        return apartmentNumber;
    }

    public void setApartmentNumber(int apartmentNumber) {
        this.apartmentNumber = apartmentNumber;
    }

    public String getAddressDetails() {
        return "[" + streetNumber + "/" + streetName + " Apt No.: " + apartmentNumber + "]";
    }
}

```

We create a test class in the src/test folder as shown below:



The test class contents are:

```

package com.test.mockitoJUnit;

import org.junit.Before;
import org.junit.Test;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.Spy;

import com.test.mockitoJUnit.AddressDetails;
import com.test.mockitoJUnit.PersonalInformation;

import static org.junit.Assert.*;

public class PersonalInformationImplSpyTest {

    @Mock
    AddressDetails addressDetails = Mockito.mock(AddressDetails.class);

    @Spy
    PersonalInformation personalInformation;

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
        personalInformation = new PersonalInformation();
        personalInformation.setAddress(addressDetails);
        personalInformation.setAddress(addressDetails);
    }

    @Test
    public void testGetAddressDetails() {
        // Create an instance of the class of the class which
        // we want to mock
        PersonalInformation personalInformation = new PersonalInformation();
        personalInformation.setAddress(addressDetails);
        personalInformation.setAddress(addressDetails);

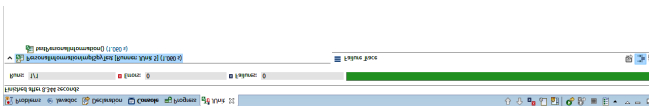
        // Verify the value is set as expected when the spy is created
        Mockito.verify(addressDetails, Mockito.times(2)).getAddressDetails();
    }
}

```

In the test class, we define a spy object for the AddressDetails class. In the setup method that will be executed before the test method, we create an object of the type PersonalInformation and use the spy to set the address. We configure the behavior of the spy object to return an address when the getAddressDetails() method is called.

In the test method, we proceed to use a simple assertion to verify the complete information about the newly added person. We also use the verify() method to check whether the getAddressDetails() method has been invoked once.

We execute this test case. The results are:

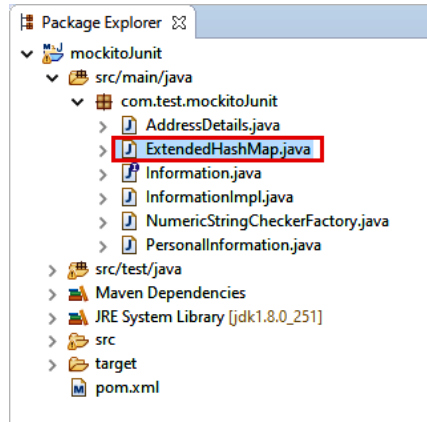


As clearly seen from the positive results, the spy object has been successfully used and stubbed for the method belonging to the AddressDetails class.

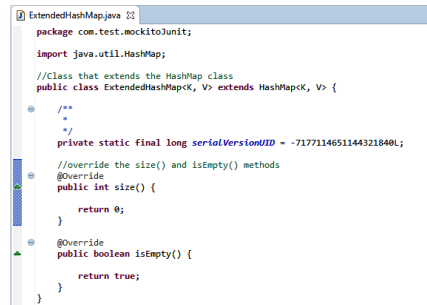
- Example 8: Mockito Verify Example

In this example, we demonstrate different ways in which the verify() API can be used in combination with Mockito API methods such as never(), atLeast(), atMost() and times() (Acharya, 2014).

To demonstrate this, we first create a simple class that extends the `HashMap` class under the `src/main` folder as shown below:

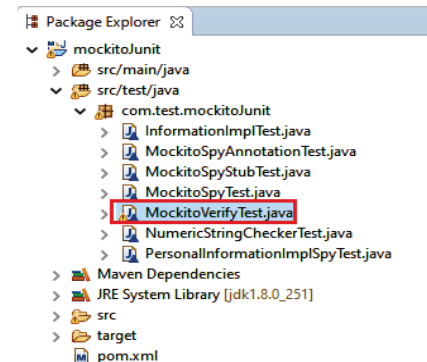


The class is as follows:



The `ExtendedHashMap` class extends the `HashMap` class and overrides two methods of the parent class: `size()` and `isEmpty()`.

We now create a new test class to test this extended class in the `src/test` folder.



The test class is as follows:

```

MockitoVerifyTest.java
package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.Mockito.atLeast;
import static org.mockito.Mockito.atMost;
import static org.mockito.Mockito.never;
import static org.mockito.Mockito.times;

import java.util.HashMap;

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

class MockitoVerifyTest {

    @Test
    void testVerify() {

        //create a mock object
        HashMap<String, String> mockedHashMap = Mockito.mock(ExtendedHashMap.class);

        //check that no interactions with the mock object have taken place yet
        Mockito.verifyNoInteractions(mockedHashMap);

        //verify that the method size() has been called exactly 1 time
        mockedHashMap.size();
        Mockito.verify(mockedHashMap, times(1)).size();

        //verify that the isEmpty() method has been called more than once and less than 10 times
        mockedHashMap.isEmpty();
        mockedHashMap.isEmpty();
        Mockito.verify(mockedHashMap, atLeast(1)).isEmpty();
        Mockito.verify(mockedHashMap, atMost(10)).isEmpty();

        //verify that the method clear() has never been invoked
        Mockito.verify(mockedHashMap, never()).clear();

        //verify that the a key-value pair has been added to the HashMap
        mockedHashMap.put("1", "A");
        Mockito.verify(mockedHashMap).put(anyString(), anyString());

        //check if the size returned is 0
        assertEquals(mockedHashMap.size(), 0);
    }
}

```

First, we create a mock object for the ExtendedHashMap class.

We then use the `verifyNoInteractions()` method to check whether the mock object has been invoked until this point or not. At this point, it has not been called which means that the verify will pass.

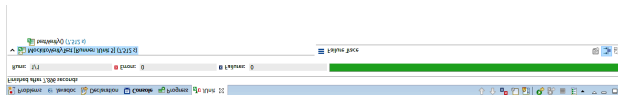
Then we invoke the `size()` method once and verify that it has been invoked exactly one time.

We proceed to invoke the `isEmpty()` method twice and then make use of the `atLeast()` method to check if it has been called a minimum of one time. We also use the `atMost()` method to check that the `isEmpty()` method has not been called more than 10 times.

We use the method `never()` to check if the `clear()` method has never been invoked.

We add data to the map and verify that it has been added using the `anyString()` method. Lastly, we make use of an assertion to check the size of the HashMap which is overridden to return zero.

We execute this test and obtain the following results.



As demonstrated by the results, the mocking of the object and the verification has passed successfully.

- Example 9: Argument Captors

In this example, we see in detail via a few examples, the use of Argument Captors.

We mock a simple `HashMap` and use `Argument Captors` to mock the values of the key and value pair of `HashMap`. We create a test class named `ArgumentCaptorTest` and add it to the `src/test` folder.

We make use of the '`ArgumentCaptor`' annotation to define the argument captor fields for the key and value.

We make use of `setUp()` method to initialize the mock methods.

We must note that calling `MockitoAnnotations.initMocks(this)` before any test method is important to get the field initialized by the framework.

In the test method, we add data (a key-value pair) to the mocked `HashMap`. We proceed to verify that the data has been added using the `capture()` method. Finally, we assert that the values of the key and value correspond to the values added to the mock object.

The test file is shown below:

```
ArgumentCaptorTest.java
package com.test.mockitoJunit;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.HashMap;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;

public class ArgumentCaptorTest {

    @Mock
    HashMap<String, String> hashMap;

    @Captor
    ArgumentCaptor<String> keyArgumentCaptor;

    @Captor
    ArgumentCaptor<String> valueArgumentCaptor;

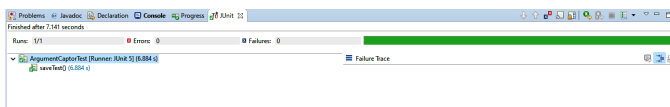
    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void argumentCaptorTest() {
        hashMap.put("A", "Apple");

        // use the argument captors
        Mockito.verify(hashMap).put(keyArgumentCaptor.capture(), valueArgumentCaptor.capture());

        // check if the values specified by the ArgumentCaptor match
        assertEquals("A", keyArgumentCaptor.getValue());
        assertEquals("Apple", valueArgumentCaptor.getValue());
    }
}
```

On execution, we obtain the following results:



As we can notice from the results, the argument captor works correctly.

We can also make use of the `ArgumentCaptor` class instead of the annotation. This is demonstrated by the class below:

```

@ArgumentCaptorTest.java
package com.test.mockito.junit;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.HashMap;

import org.junit.jupiter.api.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Mockito;

public class ArgumentCaptorTest {

    @Test
    public void argumentCaptorTest() {
        HashMap<String, String> hashMap = Mockito.mock(HashMap.class);

        Mockito.when(hashMap.get("Apple")).thenReturn("Apple");

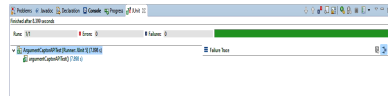
        ArgumentCaptor<String> valueArgumentCaptor = ArgumentCaptor.forClass(String.class);
        ArgumentCaptor<String> keyArgumentCaptor = ArgumentCaptor.forClass(String.class);
        // use the argument captors
        Mockito.verify(hashMap).put(keyArgumentCaptor.capture(), valueArgumentCaptor.capture());

        // check if the values specified by the argumentCaptor match
        assertEquals("Apple", valueArgumentCaptor.getValue());
        assertEquals("Apple", keyArgumentCaptor.getValue());
    }
}

```

In the test class, we have made use of the `mock()` method to mock a `HashMap`. We make use of the `ArgumentCaptor` class using the `forClass()` method to denote the type of value it should accept.

The rest of the verification and assertion remains unchanged.



As we can see from the result of the execution, the test case has executed successfully.

Argument captors can be used on all kinds of objects. In the next example, we use an Argument Captor on a `List` variable which accepts a single value rather than a key-value pair. In this example, we see the interaction of an argument captor with a spy object.

The test class is as follows:

```

SimpleArgumentCaptorTest.java
package com.test.mockito.junit;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.ArrayList;
import java.util.List;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;

public class SimpleArgumentCaptorTest {

    @Captor
    private ArgumentCaptor<String> stringCaptor;

    @BeforeEach
    public void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void testArgumentCaptor() {
        // create a spy
        List<String> stringList = Mockito.spy(ArrayList.class);
        stringList.add("Google");

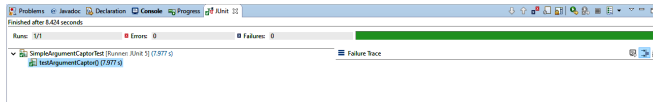
        // capture value in spy object
        Mockito.verify(stringList).add(stringCaptor.capture());

        // assert the argument value
        assertEquals(stringCaptor.getValue(), "Google");
    }
}

```

Here, we use the ‘Captor’ annotation to define an argument captor of type `String`. We create a simple spy object and add data to it. We make use of the `verify()` method to check and capture the argument added to the spy object. We then use a simple assertion to verify that the value captured is the same value that has been added to the spy object.

We execute the test to obtain the following results:



As seen from the results, the test passes successfully demonstrating the use of spies with argument captors. In the next example, we proceed to add multiple data to our list and perform the assertions. This is shown below:

```
SimpleArgumentCaptorTest.java
package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.times;

import java.util.ArrayList;
import java.util.List;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;

public class SimpleArgumentCaptorTest {

    @Captor
    private ArgumentCaptor<String> stringCaptor;

    @BeforeEach
    public void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void testArgumentCaptor() {

        // create a spy
        List<String> stringList = Mockito.spy(ArrayList.class);
        stringList.add("Google");
        stringList.add("Microsoft");

        // actual verification
        Mockito.verify(stringList, times(2)).add(stringCaptor.capture());

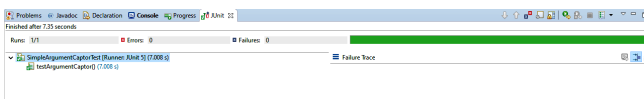
        // assert last capture
        assertEquals(stringCaptor.getValue(), "Microsoft");

        // assert all the values captured
        List<String> captorList = stringCaptor.getAllValues();
        assertEquals(captorList.get(0), "Google");
        assertEquals(captorList.get(1), "Microsoft");
    }
}
```

In the test class, we add two elements to our list. We then verify that an argument has been captured two times.

It should be noted that the capture() method of the ArgumentCaptor class will return the value that was added last. To check that all the added values have been added, we make use of the getAllValues() method and use assertions to test the values captured.

We execute the test:



The passed test shown by the results prove that the values have been captured correctly by the Argument Captor.

- Example 10: Exception Handling Using Mockito

In this example, we understand how exception handling can be mocked using the Mockito Framework.

In the src/test folder, we create a new test case as shown below:

```
ExceptionHandlerTest.java
package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import java.util.List;

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

class ExceptionHandlerTest {

    @Test
    void testException() {
        // create a mock object
        @SuppressWarnings("unchecked")
        List<Integer> mockedList = Mockito.mock(List.class);

        // define the behavior of the mock and configure it to throw an Exception
        // under a certain condition
        Mockito.when(mockedList.add(-1)).thenReturn(new RuntimeException("Negative value not allowed"));

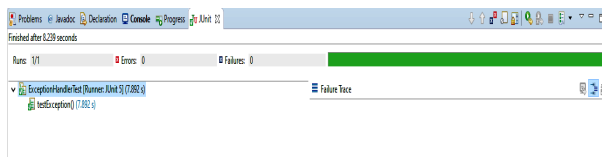
        // assert that an Exception is thrown
        Exception exception = assertThrows(RuntimeException.class, () -> mockedList.add(-1));

        // assert the exception message
        assertEquals("Negative value not allowed", exception.getMessage());
    }
}
```

In the test class, we first create a mock object which mocks the List class. We then define the mock's behavior. We program the mock object to throw an exception when we try to add -1 to the list.

We use the thenThrow() API which allows us to configure the object to throw an exception.

Lastly, we use an assert statement to check if the message thrown by the exception is as expected.



On execution, the test passes as shown above.

In the next example, we mock the NumericDivision class which throws an ArithmeticException when we try to divide any integer by zero.

The NumericDivision class is as follows:

```
NumericDivision.java
package com.test.mockitoJUnit;

public class NumericDivision {

    public int divideIntegers(int a, int b) {
        if (b == 0)
            throw new ArithmeticException("Illegal Operation: cannot divide by zero");
        else
            return a / b;
    }
}
```

The NumericDivision class consists of a single method that divides two integers and throws an ArithmeticException when we try to divide any integer by zero.

We create a test case to test the Exception thrown by this class. The test class is shown below:

```
ArithmeticExceptionHandlerTest.java
package com.test.mockitoJUnit;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.mockito.ArgumentMatchers.anyInt;
import static org.mockito.ArgumentMatchers.eq;

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

class ArithmeticExceptionHandlerTest {

    @Test
    void testArithmeticException() {
        // create a mock object of the NumericDivision class
        NumericDivision divide = Mockito.mock(NumericDivision.class);

        // define the behavior of the mock and configure it to throw an
        // ArithmeticException
        // when the value of the divisor is 0
        Mockito.doThrow(new ArithmeticException("Illegal Operation: cannot divide by zero")).when(divide)
            .divideIntegers(anyInt(), eq(0));

        // assert that an ArithmeticException is thrown
        Exception exception = assertThrows(ArithmeticException.class, () -> divide.divideIntegers(3, 0));

        // assert the exception message
        assertEquals("Illegal Operation: cannot divide by zero", exception.getMessage());
    }
}
```

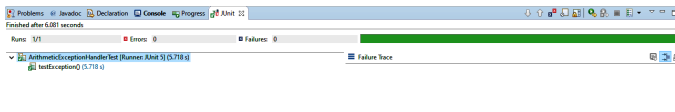
In the test class, we first mock the NumericDivision class. We define the behavior of the mock object to throw an exception when we try to divide any integer by zero. We make use of the doThrow() method to stub the method to return an exception under a certain condition.

We use the eq() method of the ArgumentMatchers class to pass the int value 0 to our mock object. We cannot pass the value 0 directly as a mock requires the use of ArgumentMatchers. If we provide the value 0 directly, a runtime mockito exception is thrown.

Then, we assert that an exception is thrown when we try to divide any integer by zero.

Lastly, we assert the message thrown by the exception.

We then execute the test case. The results are as follows:



As visible from the successful results, the exception has been successfully mocked using the framework.

- Example 11: Testing a Service Using Mockito Framework

In this example, we will study an example where we mock a service using Mockito.

We consider the PersonalInformation class that we used previously. We have updated this class to add two constructors-one default constructor and one parameterized constructor.

The class is shown below:


```

1 // PersonalInformation.java
2 package com.test.mockitoJUnit;
3
4 public class PersonalInformation {
5     String name;
6     int age;
7     AddressDetails address;
8
9     public PersonalInformation() {
10         this.name = null;
11         this.age = 0;
12         this.address = null;
13     }
14
15     public PersonalInformation(String name, int age, AddressDetails address) {
16         this.name = name;
17         this.age = age;
18         this.address = address;
19     }
20
21     public String getName() {
22         return name;
23     }
24
25     public void setName(String name) {
26         this.name = name;
27     }
28
29     public int getAge() {
30         return age;
31     }
32
33     public void setAge(int age) {
34         this.age = age;
35     }
36
37     public AddressDetails getAddress() {
38         return address;
39     }
40
41     public void setAddress(AddressDetails address) {
42         this.address = address;
43     }
44
45     public String toString() {
46         return "PersonalInformation [name: " + name + ", age: " + age + ", address: " + address.getAddress() + "]";
47     }
48 }

```

Similarly, we add two constructors to the AddressDetails class as well. This is shown below:

```

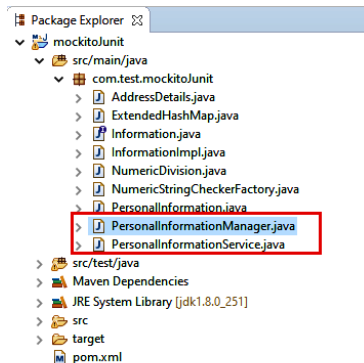
1 // AddressDetails.java
2 package com.test.mockitoJUnit;
3
4 public class AddressDetails {
5
6     public AddressDetails(String streetName, int streetNumber, int apartmentNumber) {
7         this.streetName = streetName;
8         this.streetNumber = streetNumber;
9         this.apartmentNumber = apartmentNumber;
10     }
11
12     public AddressDetails() {
13         this.streetName = null;
14         this.streetNumber = 0;
15         this.apartmentNumber = 0;
16     }
17
18     String streetName;
19     int streetNumber;
20     int apartmentNumber;
21
22     public String getStreetName() {
23         return streetName;
24     }
25
26     public void setStreetName(String streetName) {
27         this.streetName = streetName;
28     }
29
30     public int getStreetNumber() {
31         return streetNumber;
32     }
33
34     public void setStreetNumber(int streetNumber) {
35         this.streetNumber = streetNumber;
36     }
37
38     public int getApartmentNumber() {
39         return apartmentNumber;
40     }
41
42     public void setApartmentNumber(int apartmentNumber) {
43         this.apartmentNumber = apartmentNumber;
44     }
45
46     public String getAddressDetails() {
47         return "[" + streetNumber + "/" + streetName + " Apt No.: " + apartmentNumber + "]";
48     }
49 }

```

We have a Service that manipulates the data of the type PersonalInformation. We have a manager class that provides methods that query the database and update the information of people.

The service class calls upon the Manager class methods. The Manager class acts as a DAO.

These 2 classes are created under the src/main folder as shown below:



The `PersonalInformationManager` has two methods `getAge()` and `save()` which are yet to be implemented. This class, on completion, will eventually connect to the database to save the data of the person and retrieve details of the person.

```

PersonalInformationManager.java
package com.test.mockitoJUnit;

public class PersonalInformationManager {

    public int getAge() {

        //TODO: query the database for the age of the person
        return 0;
    }

    public void save(PersonalInformation personalDetails) {
        //TODO: save details of the Person to the database
    }
}

```

The service class which provides the personal information services is shown below:

```

PersonalInformationService.java
package com.test.mockitoJUnit;

public class PersonalInformationService {

    private final PersonalInformationManager personalInformationManager;

    public PersonalInformationService() {
        this(new PersonalInformationManager());
    }

    public PersonalInformationService(PersonalInformationManager informationManager) {
        this.personalInformationManager = informationManager;
    }

    public void saveDetails(PersonalInformation personalDetails) {
        personalInformationManager.save(personalDetails);
    }

    public int getAge() {
        try {
            return personalInformationManager.getAge();
        } catch (Exception e) {
            return -1;
        }
    }
}

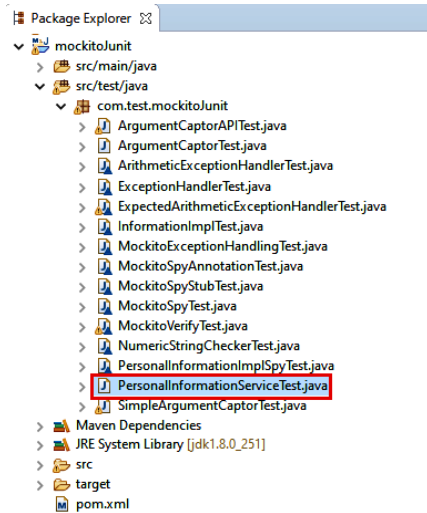
```

The class `PersonalInformationService` initializes a variable of type `PersonalInformationManager` and uses it to call its methods.

Since, the implementation of the `PersonalInformationManager` is not completed yet, but the service class has been completed, we are unable to test it using just a simple testing framework.

We need to mock the behavior in order to test it. Hence, we make use of mocks to test the service.

To test the service class, we create a new test class under the `src/test` folder as shown below:



The contents of the class are:

```

1 PersonalInformationServiceImplTest.java
2
3 package com.test.mockitoJUnit;
4
5 import org.junit.jupiter.api.Test;
6 import org.junit.jupiter.api.extension.ExtendWith;
7 import org.mockito.Mock;
8 import org.mockito.Mockito;
9 import org.mockito.junit.jupiter.MockitoExtension;
10 import org.mockito.quality.Strictness;
11
12 @ExtendWith(MockitoExtension.class)
13 @MockitoSettings(strictness = Strictness.STRICT)
14 public class PersonalInformationServiceImplTest {
15
16     @Mock
17     PersonalInformationManager personalInformationManager;
18
19     @Test
20     public void testPersonalInformationSave() throws Exception {
21         // create an object of type PersonalInformation
22         final String name = "John Wilson";
23         final int age = 46;
24         AddressDetails address = new AddressDetails("King's Street", 18, 22);
25         PersonalInformation personalInformation = new PersonalInformation(name, age, address);
26         // use the mock object to create an object of the service
27         PersonalInformationService personalInformationService = new PersonalInformationService(personalInformationManager);
28         // invoke the save method that calls the database
29         personalInformationService.saveDetails(personalInformation);
30         // verify that the method has been called and that a non-null object of type
31         // PersonalInformation has been saved
32         Mockito.verify(personalInformationManager, Mockito.times(1)).save(Mockito.isNotNull());
33         Mockito.verify(personalInformationManager, Mockito.times(1)).save(Mockito.is(PersonalInformation.class));
34         // verify that the getAge() method has never been invoked
35         Mockito.verify(personalInformationManager, Mockito.never()).getAge();
36     }
37 }

```

In the test class, we first create a mock object that mocks the `PersonalInformationManager` class.

We make use of the `@ExtendWith` annotation to which we pass a class instance of type `MockitoExtension`. `MockitoExtension` is a class provided by Mockito framework that is used to initialize the mock objects and handles stubs that are categorized as being ‘Strict.’ The `Strictness` value comes from the `Strictness` Enumeration.

The `Strictness` Enumeration is a newly added feature to the Mockito framework which is used to configure the “strictness” of Mockito and it affects the behavior of the stubs and the verifications. The `STRICT` enum provides different levels for the value of “strictness.”

The goal behind adding rules for stubbing is to improve the quality of tests.

Some of the main targets of the strictness enumeration are:

- Detection of stubs that are not used in the test classes and methods.
- Reduction in the duplication of code and elimination of test code that is not needed.
- Promoting the developer to write test that are clean by elimination of code that is ‘dead’ or unusable.
- Improve the ability to debug the tests and increase productivity of the testing process.

In the test class, we have used the value WARN. There are three possible values for the strictness level.

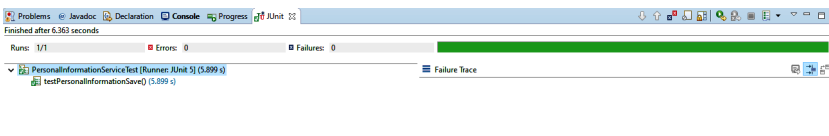
They are (Strictness-mockito-core 3.4.4 Javadoc, 2020):

1. **Lenient:** No strictness is added. This is the option by default.
2. **Warn:** Adds a level of strictness in the form of a console output. This ensures tests that are clean and improves the debugging ability of the tests. (Strictness-mockito-core 3.4.4 Javadoc, 2020).
3. **Strict-Stub:** This is the highest level of strictness. It ensures that the test cases are cleanly written, no duplication of code is present, and the debugging ability is vastly improved when using this value (Strictness-mockito-core 3.4.4 Javadoc, 2020).

Looking back at our test class, we first create an object of the base type `PersonInformation`. We make use of the mock object to create an instance of the service class. Using this instance, we call the `saveDetails()` method of the mock object.

We then proceed to verify that the `saveDetails()` method of the mock object has been called and that it has saved an object that is not null and is of type `PersonInformation`. Lastly, we verify that the `getAge()` method of the mock object has never been invoked.

We execute the test class. The results are as follows:



As observed from the results, the test has passed. This clearly demonstrates that we have successfully made use of mock objects to mock the behavior of a Service class.



Bibliography

BIBLIOGRAPHY

1. Christopher Jensen (2014). Nissan Recalls 990,000 Vehicles for Air Bag Malfunction. The New York Times. https://www.nytimes.com/2014/03/27/automobiles/nissan-recalls-990000-cars-and-trucks-for-air-bag-malfunction.html?hpw&rref=automobiles&_r=0 (accessed on 20 August 2020).
2. Acharya, S., (2014). *Mockito Essentials*. Packt Publishing Ltd.
3. Adrion, W. R., Branstad, M. A., & Cherniavsky, J. C., (1982). Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2), 159–192.
4. Afzal, W., Torkar, R., & Feldt, R., (2009). A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6), 957–976.
5. BA Baggage Failure. (2019). The Independent. <https://www.independent.co.uk/travel/news-and-advice/british-airways-baggage-lost-heathrow-luggage-ba-terminal-5-a9010306.html> (accessed on 20 August 2020).
6. Baker, C. L., (1957). Review of DD Mc-Cracken, Digital computer programming. *Math. Comput.*, 11(60), 298–305.

7. Bath, G., & McKay, J., (2014). *The Software test Engineer's Handbook: A Study Guide for the ISTQB test Analyst and Technical Test Analyst Advanced Level Certificates 2012*. Rocky Nook, Inc.
8. Bath, G., & Van, V. E., (2013). *Improving the Test Process: Implementing Improvement and Change: A Study Guide for the ISTQB Expert Level Module*. Rocky Nook, Inc.
9. Beizer, B., (2003). *Software Testing Techniques*. Dreamtech Press.
10. Beust, C., & Suleiman, H., (2007). *Next Generation Java Testing: Testng and Advanced Concepts*. Pearson Education.
11. Binet, A., & Simon, T., (1916). *The Development of Intelligence in Children (translated by Elizabeth S. Kite)*. Baltimore: Williams and Wilkins.
12. Bolger, R., (2012). *How Knight Capital Increases SEC Trade Certainty*. International Financial Law Review.
13. Borrás, C., (2006). *Overexposure of Radiation Therapy Patients in Panama: Problem Recognition and Follow-Up Measures* (Vol. 20, pp. 173–187). Revista Panamericana de Salud Pública.
14. Borrás, C., Rudder, D., & Amer, A., (2001). *Overexposure of Radiotherapy Patients in Panama: Dosimetric Aspects*. Sobreexposición de Pacientes de Radioterapia en Panama: Aspectos dosimétricos.
15. Bruns, A., Kornstadt, A., & Wichmann, D., (2009). Web application tests with selenium. *IEEE Software*, 26(5), 88–91.
16. Buchanan, M., (2015). Physics in finance: Trading at the speed of light. *Nature*, 518(7538), 161–163.
17. Captor (Mockito 3.4.4 API), (2020). javadoc.io. <https://javadoc.io/static/org.mockito/mockito-core/3.4.4/org/mockito/Captor.html> (accessed on 20 August 2020).
18. Carzaniga, A., Fuggetta, A., Hall, R. S., Heimbigner, D., Van, D. H. A., & Wolf, A. L., (1998). *A Characterization Framework for Software Deployment Technologies*. Colorado State Univ Fort Collins Dept of Computer Science.
19. Chapman, R. J., (2011). *Simple Tools and Techniques for Enterprise Risk Management* (Vol. 553). John Wiley & Sons.
20. *Cheating Frenchman Sues Uber for Unmasking Affair*, (2017). The Guardian. <https://www.theguardian.com/technology/2017/feb/13/cheating-frenchman-sues-uber-for-unmasking-affair> (accessed on 20

August 2020).

21. Co., N. M., & Ltd. (2001). *A Brief History of Nissan Motors*. Nissan motor company global website. <https://www.nissan-global.com/GCC/Japan/History/history/index-e.html> (accessed on 20 August 2020).
22. Deutsch, M. S., (1981). *Software Verification and Validation: Realistic Project Approaches*. Prentice Hall PTR.
23. Devi, T. R., (2012). Importance of testing in software development life cycle. *International Journal of Scientific and Engineering Research*, 3(5), 1–5.
24. Dowson, M., (1997). The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2), 84.
25. Fairley, R. E., (1978). Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4), 14–23.
26. Fleischman, W., (2010). Electronic voting systems and the therac-25: What have we learned? *The “Backwards, Forwards, and Sideways Changes” of ICT, Proceedings of ETHICOMP*, 170–179.
27. Furrer, F. J., (2019). *Future-Proof Software-Systems: A Sustainable Evolution Strategy*. Springer.
28. Gamma, E., & Beck, K., (2006). JUnit.
29. Gelperin, D., & Hetzel, B., (1988). The growth of software testing. *Communications of the ACM*, 31(6), 687–695.
30. Gojare, S., Joshi, R., & Gaigaware, D., (2015). Analysis and design of selenium web driver automation testing framework. *Procedia Computer Science*, 50, 341–346.
31. Graham, D., Van, V. E., & Evans, I., (2008). *Foundations of Software Testing: ISTQB Certification*. Cengage Learning EMEA.
32. Grzejszczak, M., (2014). *Mockito Cookbook*. Packt Publishing Ltd.
33. Gupta, M. M., (1987). New and recent IEEE publications. *IEEE Spectrum*, 22, 44–00.
34. Hodges, A., (2009). Alan Turing and the Turing test. In: *Parsing the Turing Test* (pp. 13–22). Springer, Dordrecht.
35. Homès, B., (2013). *Fundamentals of Software Testing*. John Wiley & Sons.
36. Howden, W. E., (1982). Life-cycle software validation. *Computer*, 2, 71–78.
37. Howden, W. E., (1987). *Functional Program Testing and Analysis*

- (Vol. 2). New York: McGraw-Hill.
38. Isidore, C., (2014). *Nissan Recalls 1 Million Vehicles due to Airbag Flaw*. CNNMoney. <https://money.cnn.com/2014/03/26/autos/nissan-recall/index.html> (accessed on 20 August 2020).
 39. JUnit 5, (2020). JUnit 5. <https://JUnit.org/JUnit5/> (accessed on 20 August 2020).
 40. Kaur, M., & Singh, R., (2014). A review of software testing techniques. *International Journal of Electronic and Electrical Engineering*, 7(5), 463–474.
 41. Khan, M. A., & Sadiq, M., (2011). Analysis of black box software testing techniques: A case study. In: *The 2011 International Conference and Workshop on Current Trends in Information Technology (CTIT 11)* (pp. 1–5). IEEE.
 42. Khan, M. E., (2011). Different approaches to white box testing technique for finding errors. *International Journal of Software Engineering and Its Applications*, 5(3), 1–14.
 43. Kubde, P., & Sable, D., (2014). The role of verification and validation in system development life cycle. *International Journal of Research in Advent Technology*, 2(2).
 44. Layt, S., (2019). *More Trouble for Queensland Hospital Software after Statewide Issues*. Brisbane Times. <https://www.brisbanetimes.com.au/national/queensland/more-trouble-for-queensland-hospital-software-after-statewide-issues-20190911-p52q46.html> (accessed on 20 August 2020).
 45. Leveson, N. G., & Clark, S. T., (1993). An investigation of the therac-25 accidents. *Computer*, 18–41.
 46. Lions, J. L., Luebeck, L., Fauquembergue, J. L., Kahn, G., Kubbat, W., Levedag, S., & O'Halloran, C., (1996). *Ariane 5 Flight 501 Failure Report by the Inquiry Board*.
 47. Man Sues Uber for Revealing Affair, (2017). BBC News. <https://www.bbc.com/news/world-europe-38948281> (accessed on 20 August 2020).
 48. Mars Climate Orbiter Failure Board Releases Report, (1999). *NASA's Mars Exploration Program*. <https://mars.nasa.gov/msp98/news/mco991110.html> (accessed on 20 August 2020).
 49. Mockito (Mockito 3.4.4 API), (2020). javadoc.io. <https://javadoc.io/static/org.mockito/mockito-core/3.4.4/org/mockito/Mockito.html> (accessed on 20 August 2020).

50. Mockito Annotations-Mockito-Core 3.4.4 Javadoc, (2020). javadoc.io. <https://www.javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/MockitoAnnotations.html> (accessed on 20 August 2020).
51. Mock-Mockito-Core 3.4.4 Javadoc, (2020). javadoc.io. <https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mock.html> (accessed on 20 August 2020).
52. Morris, F. L., & Jones, C. B., (1984). An early program proof by Alan Turing. *IEEE Annals of the History of Computing*, 2, 139–143.
53. NASA Safety Center, (2009). *System Failure Case Studies*. https://sma.nasa.gov/docs/default-source/safety-messages/safetymessage-2009-08-01-themarsclimateorbitertermishap.pdf?sfvrsn=eaa1ef8_4 (accessed on 20 August 2020).
54. Nuseibeh, B., (1997). Ariane 5: Whodunnit?. *IEEE Software*, 14(3), 15.
55. Oza, N. V., Hall, T., Rainer, A., & Grey, S., (2006). Trust in software outsourcing relationships: An empirical investigation of Indian software companies. *Information and Software Technology*, 48(5), 345–354.
56. Porrello, A. M., (2012). *Death and Denial: The Failure of the THERAC-25, A Medical Linear Accelerator*.
57. Razak, R. A., & Fahrurazi, F. R., (2011). Agile testing with Selenium. In: *2011 Malaysian Conference in Software Engineering* (pp. 217–219). IEEE.
58. Repetition Info (JUnit 5.0.0 API), (2017). JUnit 5. <https://JUnit.org/JUnit5/docs/5.0.0/api/org/JUnit/jupiter/api/RepetitionInfo.html> (accessed on 20 August 2020).
59. Richardson, D. J., & Wolf, A. L., (1996). Software testing at the architectural level. In: *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints' 96) on SIGSOFT'96 Workshops* (pp. 68–71).
60. Robillard, M. P., Coelho, W., & Murphy, G. C., (2004). How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12), 889–903.
61. Shand, R. M., (1994). User manuals as project management tools: I. theoretical background. *IEEE Transactions on Professional Communication*, 37(2), 75–80.
62. Singh, S. K., & Singh, A., (2012). *Software Testing*. Vandana

Publications.

63. Spillner, A., Linz, T., Rossner, T., & Winter, M., (2007). *Software Testing Practice: Test Management: A Study Guide for the Certified Tester Exam ISTQB Advanced Level*. Rocky Nook, Inc.
64. Spy (Mockito 3.4.4 API), (2020). javadoc.io. <https://javadoc.io/static/org.mockito/mockito-core/3.4.4/org/mockito/Spy.html> (accessed on 20 August 2020).
65. Strictnes-Mockito-Core 3.4.4 Javadoc, (2020). javadoc.io. <https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/quality/Strictness.html> (accessed on 20 August 2020).
66. Tabbaa, B., (2019). *The Rise and Fall of Knight Capital: Buy High, Sell Low, Rinse and Repeat*. Medium. <https://medium.com/dataseries/the-rise-and-fall-of-knight-capital-buy-high-sell-low-rinse-and-repeat-aef17fae780f6> (accessed on 20 August 2020).
67. Teli, S. N., Majali, V. S., Bhushi, U. M., & Surange, V. G., (2014). Impact of poor quality cost in automobile industry. *International Journal of Quality Engineering and Technology*, 4(1), 21–41.
68. Terman, L. M., & Merrill, M. A., (1960). *Stanford-Binet Intelligence Scale: Manual for the Third Revision*. Form LM.
69. Test, (2020). [www.dictionary.com. https://www.dictionary.com/browse/testing](http://www.dictionary.com/browse/testing) (accessed on 20 August 2020).
70. TestNG, (2004). *TestNG*. Retrieved from: <https://testng.org/doc/documentation-main.html> (accessed on 20 August 2020).
71. Turing, A. M., (1936). On computable numbers, with an application to the entscheidungs problem. *J. of Math*, 58(345–363), 5.
72. Turner, C. S., (1993). An investigation of the therac-25 accidents. *Computer*, 18(9I62/93), 0700-001830300.
73. View, T. O. C., (1990). IEEE standard glossary of software engineering terminology. *IEEE Std.*, 610–612.
74. Wirth, N., (2008). A brief history of software engineering. *IEEE Annals of the History of Computing*, 30(3), 32–39.
75. World War II Fitness Test, (2020). The art of manliness. <https://www.artofmanliness.com/articles/are-you-as-fit-as-a-world-war-ii-gi/> (accessed on 20 August 2020).

INDEX

A

Acceptance testing 144, 148, 150
Acceptance testing level 156
ADA programming language 3
Airbag system 57
Alpha and beta testing 149
Angular momentum desaturation
(AMD) 49
Annotation declaration 188
API interface 250
Assembly language 3
Automated testing framework 207
Automobile manufacturing compa-
ny 55

B

Baggage system 109
Banking application 21
Banking software application 24

BeforeEach annotation 191
Behavior driven development
(BDD) 259
Black box testing 131, 137, 138,
154, 161
Boolean expression 137
Boolean value 198, 199, 231
Boundary value analysis 139

C

Climate orbiter's system 52
Code based testing 132
Commonplace methods 259
Communication systems 50
Complex systems 74
Complex web application 253
Computer input system 79
Computerized system 62
Computerized treatment system 62
Computer programming 8

Computer science 7
 Configuration file 219
 Controlled test environment 37
 Cost-effective process 22
 Critical software 76
 Cross-browser testing 233
 CSVSource annotation 181, 182
 Customer satisfaction 22, 25

D

Data provider method 222
 Debugging ability 294
 Detection systems 59
 Development phase 19, 22, 23
 Development team 19, 26
 Digital computer programming 8
 Digital computer systems 77
 Dynamic testing 126, 131, 132, 137, 156

E

Effective programming skills 97
 Electronic implementation 111
 Electronic trading group (ETG) 33
 Error detection models 10

F

Federal information processing systems 11
 Financial industry 43
 Financial investors 35
 Financial losses 23
 Formal communication 55
 Functional testing 151

G

Groundbreaking technology 244

H

Hardware components 6
 High-quality software 121
 Horizontal bias (BH) 31
 Hyperlink text 256, 257

I

Independent subsystem 156
 Industry coding standards 98
 Integrated electronic medical record (ieMR) system 111
 Integrated software system 146
 Intelligent behavior 7
 International standards 125
 IRetryAnalyzer interface 228, 229
 IRS systems 31

J

Jar files 170
 Java programming language 166
 Java project 168, 210, 267
 Java virtual machine 166
 Java web projects 235
 JUnit dependencies 168, 172
 JUnit framework 168, 189, 190, 191, 195
 JUnit-vintage project 205

L

Legal liabilities 116
 Life altering systems 69

M

Machine hardware 89
 Maven 172, 238, 270
 Memory management 126
 Metric system 50
 Monitoring system 44

Multidata systems 66, 68, 75, 76

N

Navigation software 50, 51

Navigation system 30, 49, 50, 51, 52, 55

Non-functional testing 152

O

Occupant identification system 56

Online brokerage firms 34

Operating system 93

P

Power Peg' algorithm 37, 38, 45

Private markets 34

Probabilistic risk assessment 100

Programmatical errors 21

Programmer 21

Programming certification 97

Programming languages 3

Q

Quality assurance 69

R

Radiation therapists 65

Radiation therapy system 72, 103

Radiation treatment therapy 103

Reference system 7

Regression testing 154, 155

Retail liquidity program (RLP) 35

retryAnalyzer attribute 230

Risk management controls 39

RLP component 46

Runtime environment 126

S

SDLC model 158

Security testing 153

Selenium-java maven dependency 240

Selenium WebDriver dependency 240

Self-destructed protocol 30

Service class 294

SMARS system 36, 37, 38, 43

Smart market access routing system (SMARS) 36

Software analysis 10

Software application 3, 10, 18, 19, 25

Software community 10

Software complexity 3

Software deformity 58

Software design 60

Software development life cycle 6, 144

Software development process 126

Software development project 11

Software engineering 10

Software error 33

Software evolution 6

Software inclusive approach 77

Software interlocking mechanism 91

Software issues 6

Software languages 3

Software product 18

Software reliability 6

Software review 10

Software systems 9

Software testing 3

Software testing process 125

Software testing qualification board 120

State transition diagram 142

Static testing 126, 127, 128, 129,

130

STEP methodology 12, 14

Strictness enumeration 293

System level testing 147

T

Test annotation 178, 191, 208, 209,
221, 226, 231Test class 167, 168, 179, 181, 182,
183, 184, 185, 186, 187, 189,
190, 191, 193, 194, 195, 196,
197, 198, 199, 200, 201, 203,
204, 205, 206, 208, 209, 215,
218, 219, 221, 222, 223, 224,
225, 226, 228, 229, 230, 231,
232, 250, 254, 255, 257, 262,
265, 266, 275, 276, 277, 278,
279, 280, 282, 283, 284, 286,
287, 288, 289, 290, 292, 293,
294

Test driven development (TDD) 15

Test environment 246

Testing 1, 2

Testing methodologies 18

Testing process 121

Testing software 15

Testing techniques 123, 126, 138,

150, 151, 153, 154, 155, 157,
159, 160, 161

Test method 198

Therac-25 system 82, 84, 93, 98

Thermoluminescent dosimetry
(TLD) 63

Timing analysis 96

Trading algorithms 37, 38

Trading systems 36

Traditional unit testing frameworks
224

Training program 2

Trajectory calculations 55

Trajectory change maneuver (TCM-
5) 53

Turing machine 3

V

Validation testing 126

Vehicle industry 56

W

Web applications 233, 234, 235

Web element 251, 252

White box testing 131, 132, 157

White box testing methods 157

Analytic Methods of Systems and Software Testing

Software is an indispensable and integral part of our quotidian lives. We rely a great deal on software for the completion of our routine activities and tasks. We do not usually ponder over the real-life impact of software in the world. The magnitude of involvement of software in our lives is overwhelming. Software systems and applications are used on a massive scale in several crucial and non-critical fields such as healthcare, education, medicine, research, engineering, business, finance, banking, etc. Software systems and applications help us perform a significant number of tasks in a simpler and cost-effective way, thereby revolutionizing the world and improving the quality of life for most. It can be said that software has evolved into one of man's basic necessities.

It is inevitable that the role and relevance of software will continue to rise in the near and distant future. Software systems and applications will surely become more complex and like today continue to be indispensable to solving the world's biggest problems. Consequently, with the inevitable rise of complex software, the challenges associated with testing such software are bound to surge. Effective testing is linked directly to the efficiency and quality of the software. Proper and detailed testing is a crucial part in the life cycle of any software. It is a way to ensure the validity and the quality of any software system. Furthermore, in depth testing helps instill confidence in the quality of the software to its stakeholders. As software systems and applications are privy to a pool of possible issues ranging from defects in the software to potential malicious attacks, it is prudent to evaluate the software as thoroughly as possible depending on its criticality. Comprehensive testing and analysis of any software solution is paramount in order to ensure that no harmful and risky vulnerabilities in the software can be exploited which could have adverse effects on the software and its users.

Ergo, it is imperative that the construction of software systems is robust and secure. One of the chief reasons for software failure is the lack of quality testing. Inadequate testing of software, in most cases results in varying types of losses such as monetary, reputation, clients, etc. Developing software applications of good quality is the precursor to ensuring efficient testing of its quality. A combination of quality and efficient testing is the key to strengthening software systems and making it less susceptible to potential internal issues and external threats. Naturally, it is safe to assume that testing takes up a significantly large percentage in the determination of the success or effectiveness of a software.

In this book, we will look deeply into the history of testing in the field of software. The risks of inadequate testing mechanisms will be explored in detail. Different mechanisms of software testing will be examined and guidelines for implementation of test strategies in software will be presented in this book. The book covers software testing patterns and software design that helps build robust software applications. Practical implementations of code that explains different test scenarios have been provided in this book.



Neha Kaul is an experienced software consultant and technical author currently residing in Paris, France. She is the author of four technical books: Object Oriented Programming with Java, Logging Frameworks in Java, Applications of Data Mining in Engineering, Management and Medicine and Software Security: Building Secure Software Applications. She received her double Master's Degree in Computer and Communication Networks and Information Technology from Telecom SudParis and University Paris-Saclay in 2016. She is a recipient of the prestigious Telecom Scholarship for Excellence provided by Fondation Telecom, France. She received the Bachelor of Engineering degree in Computer Engineering from the University of Pune, India in 2011. From 2011 to 2014, she was employed as a Software Engineer with Geometric Ltd, Pune, India. Her major avenues of research include Advanced Java/J2EE frameworks, Automation Testing, Software Quality, Software Security, Logging Frameworks, Project Management and Leadership.