Software Testing and Lines of Codes-A Study on Software Engineering Design Patterns

Suhaib Ahmed
Computer Science Engineering
Amity University
Dubai, UAE
suhaiba1@amitydubai.ae

Lipsa Sadath
Faculty, Software Engineering
Amity University
Dubai, UAE
lsadath@amityuniversity.ae

Jumana Nagaria
Computer Science Engineering
Amity University
Dubai, UAE
jumanan2@amitydubai.ae

***Abstract*- There has been n number of software development designs and test patterns in software engineering. Frameworks are available on methods to calculate the time, effort and human power required to develop a software. Our work is a novel study on the basics of how proportional is the time required to test a software based on the LOCs (Lines of Codes) and the design used. The work aims to interlink the understanding between two kinds of structural testing. This basic study experiment was conducted using 20 different programs. We argue and prove that there is an interconnection between the design, Lines of Code (LOC) and testing.**

Keywords— software testing, software design, structural test, top-down design, bottom-up design, LOC.

I. INTRODUCTION

Every Software is tested before it is used. There are different types of test methods. The technique to design a software matters so that it is convenient to be used by the user. This paper uses 3 types of Lines of Code (LOC) – less than 50, between 50-100 and above 100. The experimental data uses structural testing which is based on top down design and bottom up design. We argue that bottom up design is better for software development through a graph plotted between the test time and the Lines of Code (LOC) using 20 programs with different programming languages and two different program designs. Basic excel tools were utilized for calculating time and speed to generate the resultant graph. Thus we are using a UML design to summarize our results.

The paper follows the order as Section II includes details on software testing, importance of testing, structural and functional tests, Section III is a discussion on the importance of software design patterns such as bottom-up and top-down with also details on cohesion and coupling and lines of codes, Section IV includes literature review on various software tests studies on different platforms, Section V is our test experiment followed by results discussion and future works in Section VI and VII respectively.

II. SOFTWARE TESTING

Software test is a process of testing software with the intention of finding errors. The purpose of testing is to prove that software is free from any error. Software's design is very important in order to correctly test a program. Removal of bugs and errors in the earlier stages could cost less than those at the end[1].

Consider a software designed for a bank where every data is crucial. The bank holds the trust of the customers. Imagine a minor error in the program can result in major calculation problems leading to a big confusion. This often occurs due to faulty software. Therefore it becomes vital to test the software before using it officially.

A. Importance of Software Testing

Testing is important for the successful release of a software product. It is essential for gaining customer confidence and satisfaction. Testing helps to develop a high quality product hence requiring lower maintenance cost[1]. Then the longevity of the software is more and the chances of it to fail are lesser. This is important to develop an effective software application. In software testing it is advised to test the product earlier because the cost of the corrections are lower in the early stages. The process gets difficult to if the product is commercial.

Much problems have been discussed in the past due to technical failures in products due to lack of software testing [2]. October 2007 released Guitar Hero III: Legends of Rock a game that was popular in six days of its release[23]. This game allowed players to play music with a guitar-like controller. But within a month, the players discovered that it could not play a stereo sound which was the primary source for rock music.

It was difficult to correct the mistake as millions of customers were involved. To replace the error, the company offered to create a website that facilitated the process. The customers had to wait for the replacement disc. All of these, the company could have avoided by a simple software test. Therefore software testing is equally important as writing codes. If testing was overlooked, it led to disastrous results.

In September 1999, due to software error there was a loss of US$125 million. The martin engineers used English units for navigation where NASA [22] used metric units. This inconsistency resulted in losing a valuable spacecraft. Testing the spacecraft before releasing in a simulated environment would have helped to catch the error. One of the worst setbacks was the when the US missile system [21] couldn't detect an incoming missile from Iraqi scud. That resulted in killing of 28 soldiers injuring others in US army barracks. The software incorrectly calculated that the missile was out of range. To avoid this timing error, a simulated test for a long period could have been held. The top priority of testing is to avoid the errors in the final software version. Testing a program before release is worth the expense.

The purpose of testing is to assure that the design is implemented properly with the help of the code written by the program. The system must be according to the customer needs[3].

B. *Functional Testing*

Functional Testing is based on the specification requirements. In this type of testing each function is confirmed with the requirements[4]. This test involves black box testing technique which is not concerned about the source code of the application but only the internal logic. During functional testing each function are tested by giving suitable inputs, and comparing them to the anticipated results. Checking of Interfaces, APIs, Client server applications and other functionalities of software are involved in functional testing.

C. *Structural Testing*

Structural Testing also called White-Box or glass-box tests are done to find bugs for various operations that occur in the lines of code of a program. It encompasses complete knowledge of the system[5]. The structural tests help the testers to find out problem based on the operation of the program. For example, the structural test exposes that the code has been written for a 40 character username but the application is only allows 20 characters to be entered.

Various techniques of Structural testing are code coverage, statement coverage and branch coverage. Code coverage eliminates any empty spaces in the test case given. It removes areas of the code that are not suitable. Hence the quality of the software application increases. Statement coverage refers to test each statement at least once during the testing process. Finally, Branch coverage checks the code with every path possible according to if loops and conditional loops in the source code.

Structural Testing helps in revealing errors in hidden code and identifying dead code or major issues in the program. It is better to do a structural test after the software application has been developed and after each modification. It is also necessary for the testers to have a deep knowledge on the language that is being used. Structural testing can take days and weeks or even longer for large programming applications to fully test. On the other end, it can be expensive to expend time and money due to the complexity involved.

Structural testing involves many testing types that evaluate the application source code. Unit testing is primary test that is done to any application and it helps to find most of the major bugs early in the development. These bugs are cheap and easy to fix in the starting phases. Memory Leak testing is done to eliminate any memory leaks which make the application run slower. Mutation testing is done to discover a new efficient coding technique that should be followed for expanding the software solution.

III. SOFTWARE DESIGN

Software design is used to convert the system specification into an implementable system. It describes the model, the structure, the algorithms used and the interfaces between the different system components. It deals with the size and complexity of programs[6]. The developers create the software design in iterative steps. They do not create the design at once but they keep adding the details to the previous design. Object Oriented program design such as C++ and Java as an unit of software design because it is widely approved by software industry[7]. The designs may be sequential and interleaved.

The information of the operating system, the database and other application system is very essential for a designer. These are the speciation requirements from the customer which may vary depending on the software being developed.

Example: A real-time system does not include a database so therefore there is no design for database but it includes a timing design.

There are controversies that the software design is not controlled by the software engineers. It is believed that software design gets evolved with culture. Example: Google Kubernetes[8].

A. *Importance of Software Design*

Software Design helps to build a model for the required product or the system. The quality of the model is evaluated and tested before coding where a large number of end users are involved[9]. Software design acts as a blueprint or a plan for the system. The man objective of the design phase is to identify the major modules of the system and there interaction with each other[10]. Software must have a good design that makes it easier for the customer to understand, interact and reliable. The design should provide accurate, complete detailed working of software[1]. A good design allows making small changes in any phase. The requirements of the software are fulfilled by designing according to the need of the customer.

B. *Top-Down Design*

Top down design is another strategy approach to identify the major modules and decompose them into lower modules iteratively until the desired system is achieved. Most of the design methodologies are based on this approach as the specification are very clear and the development starts from scratch[1]. Testing is not done when coding starts immediately after designing unless all the subsidiary modules are coded.

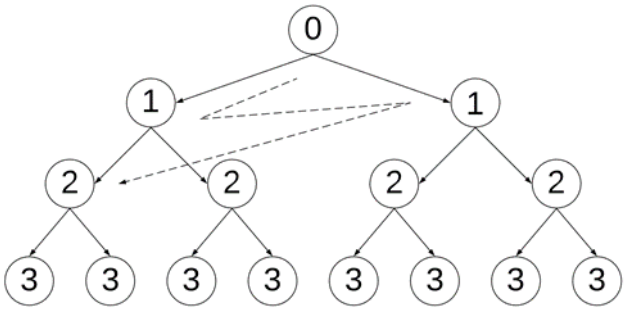

Fig. 1. Top down Program Design

C. *Bottom-up Design*

One of the strategies to design is Bottom-up design. It finds the modules required by the different programs. These modules can be input and out functions, graphical functions which are combined together in the form of library[1]. Combining these modules into a big module until the desired program is achieved

but it is sometimes difficult to identify the modules and initiate the design[11].

As the name suggests, the designs are processed from the bottom layer. Bottom up design is suitable for designing software from an existing system.

Example: A module for a student result system includes the details of the students as well as the subjects for entering the marks respectively.

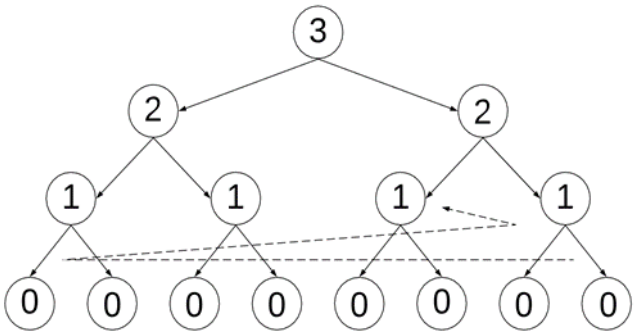

Fig. 2. Bottom up Program Design

D. Cohesion and Coupling

Cohesion is a software metric used in software engineering to measure the unity within a software module. In other words, cohesion is the functional strength of a module. A module with maximum cohesion is functionally independent[1]. Cohesion emphases on the internal interactions inside a class of a program[12]. Functional Independence refers to the ability to perform a single task or a function. This type of modules has minimal interaction with other modules in the software. In bottom up design the dynamic cohesion is the interdependency between the objects during runtime[13]. Good software will have high cohesion. Cohesion is identified by different types based on the functionality of the given module. In feature cohesion the different roles of various interfaces are directly related[14].

Coupling is a metric used to measure the inter-relationship of various modules in a software structure. In terms of bottom-up approach, software modules are divided into smaller modules that are interdependent[1]. Coupling emphases the interaction between different classes in a program[12]. Modularization helps in achieving data independence, where even if the data is changed in one module, it won't affect the data in other modules[15]. Therefore low coupling is preferred. The complexity and the abstraction of two or more modules affect the degree of coupling.

E. Lines of Code

Lines of Code (LOC), is a metric used to measure the program size by counting the number of lines in the source code. The lines of code include the header files, functions, declarations and other executable and non-executable codes. There was a predicament whether to include blank lines and comments in LOC. Later, it was certain that comments in the code help the testers to debug the program without difficulty and also reducing the cost during maintenance phase. The Lines of Code (LOC) is always dependent on the language used for development[1]. It is inadequate for calculating other tasks like functionality, complexity, efficiency of the code.

The effort and the duration of the software project are calculated with the help of LOC and other constraints affecting the development. The cost of a software project planning is estimated by using Single Variable Models or Multi Variable Models. The common equation is,

$$C = {}_a L^b \qquad \text{…….. (1)}$$

where C is the cost of effort in person-months, L is the lines of code and a, b are the constants derived from past projects of the organization.

IV. LITERATURE REVIEW

Software's were widely integrated into many products. The concept of search based software testing influenced the domain specialists to build test cases with very little software testing knowledge. According to Marculescu et al., there was an interaction between the testing system and the domain specialists during the iterative developments to maintain quality that determined the fitness of possible solutions[16]. Lower quality software was built due to cost or unavailability of resources and requirement of expertise to test software. The running example included the mechanical arm that made use of the joystick. ISBSE model developed consisted of two cycles: inner cycle and outer cycle. The interactions between these cycles were attained through feedbacks, evaluations from domain expert. The outer cycle arbitrated the interface between the domain specialist and interaction handler. The primary role of interaction handler was to show the possible solution to the domain expert and take necessary feedback. If fitness function was replaced by a human domain specialist that resulted in low potential solution but additional knowledge from the human added up to improve the selection mechanism. Feedbacks were mostly related to solution candidate, display feedback that were related to the number of candidates exhibited, memory requirements, response times and quality focus feedback. The domain expertise was enabled to modify the criteria that implemented the preliminary selection. The main objective of the inner cycle was to provide best solution for the human specialist. Search component and intermediate fitness function were the two components of the inner cycle. The purpose of intermediate fitness function was to provide early screening of solutions so that the successful solutions were studied by the domain expertise. The search component encapsulated the algorithm to generate potential solution. The significance arises for systems where domain knowledge was the key factor in the success of the testing process.

Mutation testing is a software testing method which helped to identify artificial defects in a system under test[17]. This method was conducted by generating mutants in the original program and by running the mutant program and the original program through the test suites. If the test suits does not identifies fault in the original program and all the artificial faults in the mutants, it was determine that the original program is free from any error. Due to the storage requirement, high computation and human effort many cost reduction techniques had been developed. The use of good mutation tools was very

important for a mutation testing. A good mutation testing tool consisted of many cost reduction technique. It was suggested to choose any tool with correct operator according to the test requirement. The test cases were tried to keep independent and built a re-executable test with well-suited for the testing tools. Mutants were created for critical parts. If the system was too large, evaluation of these tests took a long time so it was advised to perform the tests at night. A perfect threshold for the mutation test was resulted in cost rather than in benefits. The study showed that each mutation tools had cost reduction technique.

Successful software's were built upon user's satisfaction and the usefulness. However faults, defects, misunderstanding in any project were leaded to high cost which in turn affected the quality of the product[18]. The aim of software review process was to reduce the cost as well as to produce good quality products. The necessity of taking preventive measure in the early stage of development of the project helped to fix the errors. During the requirement analysis and development many case study reflected the benefits of software review. The case study conducted by Petunova.O and Berzisa.S was based on nine European countries that involved test planning, test case design, business requirement. The case study collected the statistics of 4 release testing based on the level of complexity: critical, major and minor defect before implementation of test case review. The data indicated that 206 defects were found out of which 62% - major defects, 25 %-critical defects and others were minor defects. The test case review were implemented and checked if they were according to the well-defined standards and necessities. Another tester reviewed the test case which had equivalent experiences as an author but could not write the test. To sum up the report, the statistics were collected after the test review that provided the end result of 145 defects, out of which major defect were 65%,minor defects were 11% and others as critical defect. As observed the percentage of defects did not change but the total numbers of defects were reduced by 30%.

In software testing, communication played a big role that helped to fix the potential defects. There were communication problems between the testers and the developers who were the external service provider. Sometimes due to emails, more time was required to explain the developer about the defects and mistakes. Software quality depended on information exchanged as it made it easier to find the errors and take the significant step. Information availability reduced the confusion and incorrect decision. Sometimes important information regarding the test was not passed to the tester. Often tester used outdated description version which resulted in making mistake. It was believed that the software review reduced the total testing time if the defects were fixed at early stages.

Biological modeling systems were important where simulated data on the computer was used to predict the behavior which was resilient by nature. This leaded to limited experiments in laboratories which were expensive and slow. As a result, the usages of computers were becoming necessary because it was less expensive and faster. Further analyzing biological systems, software approaches were flexible by means of replication, isolation and delegation. This ensures that the systems failed and recovered without compromising the whole system. Thus, biological systems and software systems were more flexible, scalable and welcomed changes.

The author's work in the paper observed that biological immunity and software resilience were considered as two sides of the same coin[19]. For example, immune systems under biological systems were resilient system. Thus, it helped software systems to be inspired from the elements, relations and behaviors of immune systems.

The Akka Actor model was an architecture model which was used for structuring resilient systems which supported scalable and concurrent computation. It was inspired by the immune system elements which were the first step towards making a bio-inspired paradigm for developing software systems. The Actor Model was characterized by (i) inherent concurrency of computation within and among Actors, (ii) dynamic creation/replication of Actors, (iii) inclusion of Actor addresses in messages, (iv)interaction only through direct asynchronous message passing.

The lightweight approach for a specification testing included planning, performing test and finding faults in a specification to execute the desired output. A simple file system on a SIM model for mobile application with Global system for mobile communication (GSM) was used as a framework[20]. The aim of the model was to calculate the effective cost and to detect faults. The framework proposed indicated how different roles are responsible for the specification testing. There were 4 roles where the developer implemented, the requirement engineer provided the specifications and requirements for the development and the testers were responsible for generating a test graph. The test graph was used to compare with the specification that was derived from the requirement engineers. Finally the role of the reviewer was required for rechecking the test graph and to complete the testing process.

A control flow graph enhanced easy readability because it was created based on effect predicate. There was a start node and each conjunction was a new node that was attached with the sub graphs of its arguments. To make the testing easier, a telegraph editor was used and the traversal was automated. An operation test was performed to check the precondition and input for any violation, the post-state and outputs were related to the specified inputs. Animator was needed to verify if the operation test was corresponding with the specifications.

There were various risk factor associated with the framework such as test graph which was derived from incorrect requirements or was incomplete. The test graph was a partial model that doesn't contain important behaviors to be tested. There were greater risks when specification and implementation model displayed the same coincidental incorrectness. It was very important to analyze that all the specifications were executed. The data collected from this framework was tested with BZ Testing Tool framework. The result displayed a fail in effort.

V. TEST EXPERIMENT

The experiment was performed on 20 different programs which range in number of lines of code and these programs were coded by the students. The lines of code were noticed to be less than 50 and ranging from 50 to above 100. These programs were classified according to the program design. The main idea was to identify the design from the programming language used. For example, Java is an object oriented approach; hence it was

categorized into Bottom-up design whereas C is Top-down design due to the procedure oriented approach. The testing was done to measure the time taken to fix the bugs and errors.

The Table 1 contains the list of various programs that were tested. Different programming languages had LOC between 0-50, 50-100 and above 100. For less than 50 lines of code, there were six programs which were tested. Out of 20, 6 programs were considered for lines of code somewhere in the range of 50 and 100 and the rest of the 4 programs were tested for over 100 Lines of code.

TABLE I EXPERIMENTAL DATA

Table No. 1						
Pattern	LOC (<50)	Language	LOC (50-100)	Language	LOC (100+)	Language
Top Down	Multiplication of Matrices	C	Implementation of Queue	C	Shift Reduce Parser	C
	Merge Sort	C	Implementation of Stack	C	Implementing Linked List	C
	Quick Sort	C	Program to construct Flag	C		
			Lexical Analysis	C		
			Recursive Descent Calculator	C		
Bottom Up	Thread Priorities	Java	Constructor and Destructor	C++	Booking_List - (Reservation Application)	Java
	Friend Functions	C++	Multilevel Inheritence	C++	Ticket_Enquiry - (Reservation Application)	Java
	Single Inheritence	C++	Arithmetic Operation in Applets	Java		
			Operations using Text field and Combo Box	Java		
			Program on different DataStreams	Java		
Total	6		10		4	

Based on the above table 1, a graph was interpreted with the help of Microsoft Excel by taking Time in (mins) and LOC as units. The average time taken by the testers to debug is plotted in this graph Fig. 4.

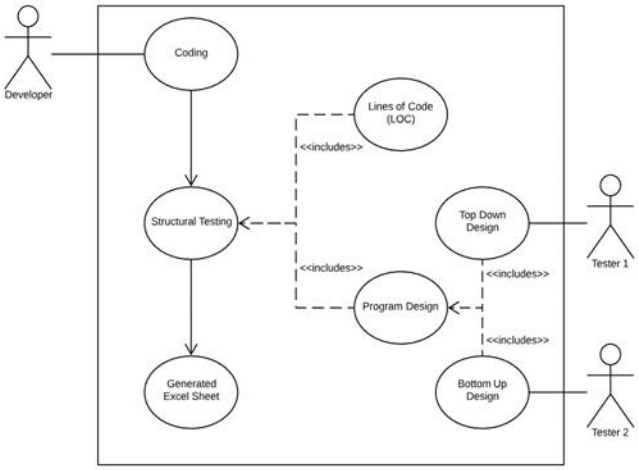

Fig. 3. UML Design

VI. RESULTS

The experimental result delivers that there is a relationship between testing and Lines of Code (LOC). The lesser the lines of code lesser is the time taken to test a program. Individual programs are integrated to make software and it is tested individually. Developer codes the software programs and testers are solely responsible to make the software error free. The Fig. 3 depicts a developer and two testers namely Tester 1 and Tester 2 as actors and their roles respectively. In Fig. 4 the Blue line indicates Top-Down Program Design and the Red line indicates Bottom-Up Program Design. The x-axis in the graph Fig. 4 represents the Lines of Code (LOC) and the y-axis represents the Time in minutes to test a program. Tester 1 performs the testing on Top-Down design, whereas Tester 2 simultaneously performs the testing on Bottom-Up design. Tester 1 proves through the Fig. 4 that top down design takes more time to perform the test as it needs to begin the testing from the starting node. The bottom up approach is far easier to test as every node is interconnected and easier to follow.

Hence the study concludes that testing is directly proportional to lines of code and bottom up approaches are far more accurate, reliable, convenient and faster than later design.

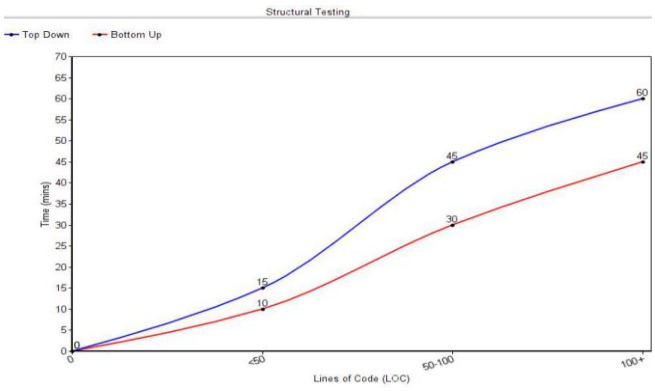

Fig. 4. Experimental Graph Result

VII. CONCLUSION AND FUTURE WORKS

Software testing guarantees the reliability and satisfaction of the customer through developing a high quality product with low maintenance cost. As many new technologies are emerging, the demand for testing is increasing. Hence this paper links how testing design and Lines of Code (LOC) can be related. Bottom-up approach is compared with Top- down approach in order to estimate the time and feasibility to find and solve a bug. It was proven that Bottom- up design is faster and easier than Top-down design.

In the upcoming world of A.I, IOT and machine learning, testing techniques will evolve. The requirement to change the old traditional method of testing is growing. Therefore testing is seen to have a better and a brighter future.

We use Lines of Code (LOC) for basic experimental data. Our future work will involve expanding the experiment in an industrial project where the size of Lines of Code will be in KLOCs. Furthermore, to prove bottom-up program design is best pattern for software designing.

VIII. REFERENCES

[1] Y. Aggarwal, K.K.; Singh, Software Engineering, 3rd ed. New Age International Publishers.

[2] E. Torres, "Inadequate Software Testing Can Be Disastrous [Essay]," IEEE Potentials, vol. 37, no. 1, pp. 9–47, 2018.

[3] Shari Lawrence Pfleeger; Joanne M.Atlee, Software Engineering: Theory and Practice, 3rd ed. Pearson Education Inc., 2006.

[4] B. Beizer, Black-box testing: Techniques for functional testing of software and systems, vol. 4, no. 8. John Wiley & Sons, Inc. (US), 1995.

[5] R. Black, Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing. John Wiley & Sons, Inc. (US), 2002.

[6] I. Sommerville, Software engineering, 9th ed. .

[7] J. Al Dallal and L. C. Briand, "An object-oriented high-level design-based class cohesion metric," Inf. Softw. Technol., vol. 52, no. 12, pp. 1346–1361, 2010.

[8] E. A. Lee, "Is software the result of top-down intelligent design or evolution?," Commun. ACM, vol. 61, no. 9, pp. 34–36, 2018.

[9] R. S. Pressman, Software Engineering: A Practitioner's Approach, 7th ed. McGraw Hill Education (India) Private Limited, 2014.

[10] P. Jalote, An Integrated Approach to Software Engineering, 3rd ed. Narosa Publishing House Pvt. Ltd., 2005.

[11] H. A. Kautz, B. Selman, M. Coen, and S. Ketchpel, "Bottom-up design of software agents," in Commun. ACM, 1994.

[12] M. English, T. Cahill, and J. Buckley, "Construct specific coupling measurement for C software," Comput. Lang. Syst. Struct., vol. 38, no. 4, pp. 300–319, 2012.

[13] V. Gupta and J. K. Chhabra, "Dynamic cohesion measures for object-oriented software," J. Syst. Archit., vol. 57, no. 4, pp. 452–462, 2011.

[14] S. Apel and D. Beyer, "Feature cohesion in software product lines," Proceeding 33rd Int. Conf. Softw. Eng. - ICSE '11, p. 421, 2011.

[15] H. Dhama, "Quantitative models of cohesion and coupling in software," J. Syst. Softw., vol. 29, no. 1, pp. 65–74, 1995.

[16] B. Marculescu, R. Feldt, and R. Torkar, "A concept for an interactive search-based software testing system," in International Symposium on Search Based Software Engineering, 2012, pp. 273–278.

[17] P. Reales, M. Polo, J. L. Fernandez-Aleman, A. Toval, and M. Piattini, "Mutation testing," IEEE Softw., vol. 31, no. 3, pp. 30–35, 2014.

[18] O. Petunova and S. Bērziša, "Test Case Review Processes in Software Testing," Inf. Technol. Manag. Sci., vol. 20, no. 1, pp. 48–53, 2017.

[19] M. Autili, A. Di Salle, F. Gallo, A. Perucci, and M. Tivoli, "Biological Immunity and Software Resilience: Two Faces of the Same Coin?," in International Workshop on Software Engineering for Resilient Systems, 2015, pp. 1–15.

[20] T. Miller and P. Strooper, "A case study in model-based testing of specifications and implementations," Softw. Testing, Verif. Reliab., vol. 22, no. 1, pp. 33–63, 2012.

[21] E. Schmitt. (1991, June 6). U.S. details flaw in Patriot missile. New York Times. [Online]. Available: https://www.nytimes.com/1991/06/06/world/us-details-flaw-in-patriot-missile.html?mcubz=0[Accessed on 15-02-2019].

[22] CNN. (1999, Sept. 30). NASA's metric confusion caused Mars orbiter loss. Available Online: http://edition.cnn.com/TECH/space/9909/30/mars.metric/ [Accessed on 15-02-2019].

[23] C. Kohler. (2007, Dec. 12). Activision sued for Guitar Hero sound problems. Wired. [Online Available]: https://www.wired.com/2007/12/activision-sued/. [Accessed on 15-02-2019].