**Failure Analysis Paper**

Introduction

Software engineering is a growing field that numerous organizations and businesses are becoming increasingly reliant on. Due to this increased reliance, it is crucial for appropriate and standardized testing regulations in place to ensure that system failures do not occur. According to the National Institute of Standards and Technology, the U.S. Economy approximately loses $59.5 billion annually due to inadequate software testing infrastructure [1].

Failure in software engineering [software engineering? software products and applications] could be an application crash or not producing the proper output [intended/expected output], or any variety of outcomes that deviate from the intended output. Failure analysis in software engineering pertains to analyzing how certain errors in a program [software] were overlooked and how this could be prevented in the future.

Testing Methods

Failures in computer science [computer science? software products] are generally minor and lead to simple logic errors, such as the program [software] not producing the correct output [intended/expected output and/or under performance from response time and resource consumption perspective, not meeting requirements put forth]. However, there are different levels of severity for the failures associated with software since a small error in code may lead to catastrophic events such as a plane crash or inaccurate military systems. Such catastrophic events usually stem from inadequate testing measures or lack of proper documentation to ensure each step is accurate.

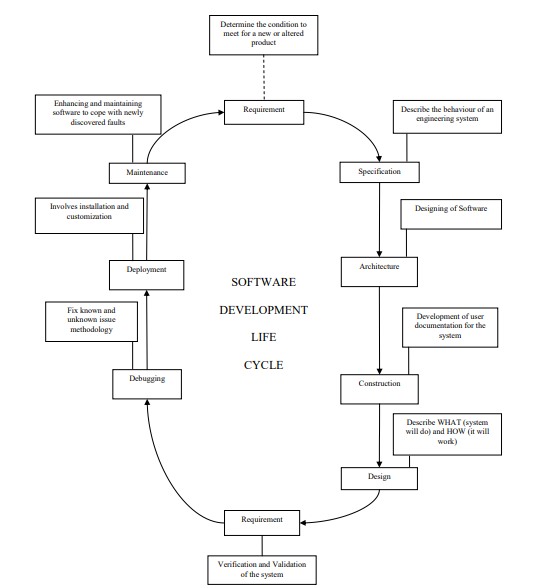


Figure 1 - Different phases of the Software Development Cycle [2]

As seen in Figure 1, there are 9 stages in the software development cycle, and it is crucial that there is software testing in each stage [2]. Furthermore, each stage is interlinked. Although a problem an error in the requirement or specification of the program [software] may seem minor, these errors will build onto each other and by deployment the error may be unnoticeable or irreversible [rearrange wording?]. Thus, starting testing early and often provides the opportunity to review requirements, ask questions, and resolve issues [2]. Thus, most failure analysis in software engineering is preventative [preventable].

A pie chart with text on it

Description automatically generated

Figure 2 - Sources of Failure in Computer Science[software products/applications] [3]

Figure 2 displays the commons [common] sources of failure in computer science [software] [3]. Requirements fault amount [does not sound right?] to 33.65% of the errors, which is the first stage of the software development cycle as seen in Figure 1 [2]. Therefore, most errors occur before the software has been designed [due to lack of requirements and specifications]. Also, there are many other errors that also cause failure such as compiler error and integration fault, which all occur due to bugs in the code that have been missed [during design and development].

Testing Methods

Most testing methods in software engineering are non-destructive and preventative [is preventative correct usage?]. The aim of these methods is to identify bugs and discrepancies in the program before the software is run on a large scale. [before the software is released to production and is generally made available for end users]

Forensic - Destructive testing

Destructive testing sees [is to find out] if a software product exhibits proper behavior when subject to improper usage or improper output [4]. This type of testing is useful for scenarios where a software user uses the software product in an improper manner [4]. One of examples of this type of testing is acceptance testing [4]. Acceptance testing occurs when the customer writes certain tests to determine if the system is doing the right things [5] [is performing the way that is designed for]. Destructive testing occurs after the code has been written and is ready to be used by customers, thus acceptance testing allows customers to feel confident that the application has the required features and [that it] behaves correctly [5]. This type of testing adheres to section 4.2.4.2 in the standard ISO/IEC/IEEE 29119 by ensuring there are appropriate test levels for each software [6]

Forensic – Non-Destructive Testing

Non-destructive testing checks whether the software product exhibits proper behavior when subject to the proper usage [the documented use-cases]. This type of testing occurs numerous times in the software development cycle to ensure each addition or change to the code does not incorrectly affect the functionality of the code [software]. An example of this type of testing is unit testing. Unit testing validates whether each software unit performs as expected when given the expected inputs [7]. These tests need to be exhaustive to ensure each scenario [use case] is accounted for. This type of testing adheres to section 3.4 in the standard IEE 1008-1987 by ensuring that all the proper [use-case] inputs for the program [software] are considered [8].

Preventative – Non-Destructive Testing

Software Failure Modes and Effects Analysis (SFMEA) is a preventative non-destructive testing method employed in software engineering. This method tries to determine all possible types of failure for each component of a software [9]. SFMEA focuses on the beginning stages of the software development cycle, such as the requirement[s] and specification[s] stage as seen in Figure 1. This method allows for the creation of a thorough plan and a detailed list of all the requirements for the program [software], essentially removing the errors that may occur due to inadequate planning [9]. This testing method adheres to section 3.128 in the standard IEEE 829, which emphasizes the need for a planning process that includes designing the test strategy and test plans [10].

**NASA Mars Climate Orbiter Failure Case Study**

Case Description

The Mars Climate Orbiter (MCO) was NASA’s first interplanetary weather satellite and a communications relay for the next lander mission to explore Mars [11]. The orbiter carried two science instruments: The Pressure Modulator Infrared Radiometer used to measure temperatures, dust, and water vapor, as well as the Mars Color Imager [11]. The orbiter burned in space in 1999 and the affiliated companies lost over $327 million [11]. The reason for this failure, according to findings by NASA’s Jet Prolusion Laboratory internal peer review, was an error in a transfer of information between the spacecraft team in Colorado and the mission navigation team in California. One team used English units while the other used metric units [12]. The main reason for this failure was the inability of NASA’s systems and the checks and balances in place to detect errors in the performance of the software [11]. Small errors that were introduced in sequential trajectory estimates over the course of the 9-month journey to Mars resulted in the closest distance the orbiter should be in relation to Mars being much lower than the expected descent seen in Figure 3 [12]. Thus, as the orbiter got too close to the Martian atmosphere, [and] it burned.

A diagram of a flight

Description automatically generated

Figure 3 - Expected Entry/Descent/Landing Phase [12]

Case Investigation and Recommendations

Error 1

Although the bug in the software appeared to be the root cause for the MCO failure, further analysis displayed that communication between two crucial teams of the MCO was severely lacking, which lead to the bug. The operations navigation team had a lack of knowledge regarding the MCO altitude control and related subsystem parameters [13]. The operations navigation team were able to detect an error by doing additional analysis, which used more resources, but they were not able to understand the significance of this error and thus did not strive to fix it [13]. The team came onboard shortly before the launch and were not present for any of the preliminary software testing, which contributed to their inadequate knowledge [13]. This error could have been prevented if this operations navigation team had been properly informed of all the previous software development and testing that had occurred. Also, the spacecraft team and operations team should have had meetings where they discussed [unforeseen or anticipated] any problems or updates. This error could have been prevented by following the standard NASA standard 8379.8, which emphasizes the need for management support of software assurance [14]. The management support needs to be familiar with the software’s required functionalities and track the actual results of the software’s performance [14]. If this standard was applied during the development of MCO, then the management support would help bridge the gap in communication between the operations and navigation team to ensure both teams knew the requirements for the software. Also, the employment of SFMEA would have enabled both teams to understand the requirements and specifications for the product before they began developing the software [9].

Error 2

Further issues were uncovered in the verification and validation process of the MCO. End-to end testing to validate the software performance and its applicability to the required specification did not appear to be accomplished [13]. The interface control process and the verification of specific system interfaces was not done at the expected level and rigor [13]. This error could have been mitigated through the employment of the standard IEEE 829, which highlights the importance of a master test plan [4]. This standard clearly breaks down exactly what should be a part of a test plan for a software product including the scope, approach, tools, resources, and levels of testing needed to adequately test the functionality of the software [4]. The standard would have helped prevent the lack of sufficient testing and ensured that there was proper testing and that each level of the testing plan was accomplished before moving on to the next step. Also, the use of unit testing would have ensured that each unit of code performed accurately, and exhaustive test cases would have been used to confirm that when given the proper inputs, the software produces the proper respective outputs [7].

Error 3

Another minor error that had a catastrophic effect for the MCO was the margins needed to ensure a successful orbit capture eroded over time [13]. Inadequate statistical analysis was utilized to fully understand the trajectory of the MCO and there were no adequate contingency plans in place in case of an anomaly [13]. Thus, if a proper contingency plan was in plan with adequate planning, tests, and commitments as well as adequate analysis of the software, then the MCO’s trajectory could have been changed once the teams recognized that the MCO was not following its intended path [13]. A recommendation for this error can be found in standard ISO/IEC/IEEE 29119, which details test plans and test strategies required for risk-based testing [6]. The operations and navigation teams should have employed a risk-based testing approach and ensured the software passed multiple levels of testing including, unit testing, integration testing, system testing, etc. as well as numerous types of testing and design techniques before allowing the MCO to go to the next phase [6]. Acceptance testing would have been a great method to confirm when given the wrong inputs there were instructions in the software for how to handle improper input, like a contingency plan specifically for software [5].

Conclusion

Software engineering is a field that requires thorough and exhaustive testing, especially to catch small bugs or logical errors [integration failures, boundary conditions, performance impacts, exceptions handling, load tolerance]. Although the errors in a program [software] may seem minor, these errors often build on top of each other or result in an output that is far off from the expected output. The time, money, and resources required to have adequate testing and planning may be high, however, adequate testing and planning is crucial to avoid huge failures that result in an unnecessary loss of money and resources [reputation, morale].

<https://www.nist.gov/system/files/documents/2021/03/24/econImpactSumm.v23.pdf>

[1] M. E. Khan and F. Khan, "Importance of Software Testing in Software Development Life Cycle," *International Journal of Computer Science Issues (IJCSI),*vol. 11, *(2),*pp. 120-123, 2014.

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4760152>

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1652020>

http://www.dsc.ufcg.edu.br/~jacques/cursos/map/recursos/Testing05.pdf

[3] ISO/IEC/IEEE. “IEEE 29119-1: Software and systems engineering – Software testing” IEEE, 2022.

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1383101>

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5983353>

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4914732>

[4] IEEE. “IEEE 829-2008: Standard for Software and System Documentation”, IEEE, 2008.

<https://www.jpl.nasa.gov/news/nasas-mars-climate-orbiter-first-martian-weather-satellite>

[11] https://www.jpl.nasa.gov/news/nasas-mars-climate-orbiter-first-martian-weather-satellite

[12] Jack Fisher, “The Information ByWay”, The Mars Program Failures, *INSIGHT*, vol. 3, no. 2, pp. 52–54, July 2000.

[13]<https://www.dcs.gla.ac.uk/~johnson/Mars/MCO_report.pdf>

[14] NASA. “NASA-STD-8379: Software Assurance and Software Safety Standard”, NASA, 2022.