The Y2K Bug: A Failure Analysis Case Study in Software Engineering

Ashley Maurer

University of Florida

ashley.maurer@ufl.edu

1 TABLE OF CONTENTS

2 COMMON TYPES OF FAILURE IN SOFTWARE ENGINEERING

2.1 INTRODUCTION

Software engineering is the "application of engineering to software," according to IEEE Standard 610.12-1990 [1]. Software engineers develop, build, test, run, and maintain software. They utilize skills like coding (in many programming languages), debugging, and design to create software products and systems that fulfill the needs of clients and users.

A common type of failure in software engineering is a system failure, which happens due to a software issue that causes the system to stop functioning for various amounts of time or to restart [2]. An unexpected output failure is when the software stops producing the user's desired or expected result [3]. This may be in the form of the program returning an incorrect value, or in the program ceasing to function altogether. Additionally, input failures are a type of failure that occurs when the user tries to interact with the program. These failures can occur only with certain inputs (transient), or for all inputs (permanent) [4]. Input failures can cause the program to return the wrong output or cause the program to cease functioning.

2.2 CAUSES OF UNEXPECTED OUTPUT FAILURES

Many causes of software failure are classified as bugs, which are coding errors that make a program produce unpredicted results that may cause it to unexpectedly stop working or produce incorrect outputs [5]. Some common types of bugs are syntax, logic, and design errors.

Syntax errors are numerous but are usually caught early on by debugging tools. A program usually will not run without fixing syntax errors, so software engineers fix these bugs quickly after every few lines of code they write. These bugs usually occur because of missing characters or incorrect indentation [6].

Logic errors occur when the software engineer incorrectly writes or calculates code, resulting in an incorrect output or a system failure [4, 6]. These bugs can occur when the programmer tries to evaluate an impossible mathematical equation, like dividing by zero, or writes possible mathematical equations, but gets the wrong result. Another logic error is an infinite loop, where the software engineer must "crash" the program to stop the program and break out of the loop [6]. Logic errors can be difficult to find since many don't show visible error messages when the program is run.

Design errors are errors in the design of the program. A software engineer may design the program to hold data in a list containing multiple types, which may make it difficult to perform calculations using the list. They also may design the program to take in a broad range of user inputs, which could make the program vulnerable to cyber-attacks. There are many flawed designs that make a program unusable for the client's demands.

The root cause of design, logic, and syntax errors is human error. A software engineer must be careful to design a program while thinking of potential user input and the calculations it will need to perform. They must predict the outputs produced by expected inputs in order to test for logic errors. It is important to start with a well thought out design when building a program in order to create a cohesive and useful product.

3 TESTING TYPES AND METHODS

3.1 PREVENTATIVE METHODS

Preventative testing methods are widely used in software engineering throughout the software development process. Testing can be conducted for all or part of a system.

Non-destructive testing techniques allow the product to remain intact, while destructive testing techniques break the product to examine its qualities. Destructive testing generally doesn't apply to software because when a program fails, the source code isn't destroyed.

To test the security of a system, testers will simulate an attack by inputting unexpected entries that may cause the system to fail. Black box testing is a method where the tester acts as a malicious user and enters inputs without access to the internal code of the program [7, 8]. White box testing is similar, except the tester can see the internal code of the program. The tester checks that inputs produce the expected output [7, 8]. These tests allow software engineers to analyze potential security failures that may be caused by the users of a program and fix them before the product is made public.

3.2 FORENSIC METHODS

Forensic testing methods are used in software engineering to collect and analyze data from previous systems in order to avoid injecting known errors into new systems [9]. It is difficult to preventatively test every outcome of a software program because they are all designed with slight variations and there is no standard test that covers every error possibility. Given these issues, it is important to test a system or program after its release.

Producing reliable results using forensic testing is difficult because the development of software building tools is very fast and unstandardized [9]. There are no standard or quality control requirements for programming languages or operating systems, so the failures of different systems are difficult to predict or control [9]. Figure 1 shows how forensic testing uses measurements and analysis to build better products [9].

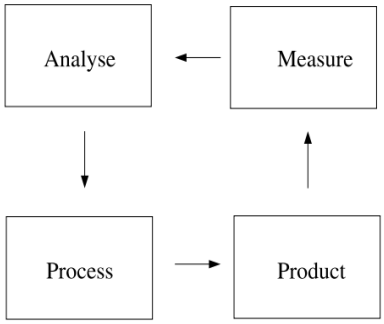

Figure 1: Control process feedback: the essence of engineering improvement

Fig. 1 [9]

Forensic process engineering analyzes the amount of time a project takes to produce a product [9]. Often, software engineers fail to estimate the amount of time required to implement a solution for a problem. The results of this test showed that longer solutions needed to be broken into smaller steps that would each take less than 5 days, which allowed the engineers to plan and finish the project in a reasonable amount of time [9]. This prevents the project from losing relevance or failing to be completed.

Forensic product engineering analyzes patterns of project failures to design new preventative tests for these failures [9]. Many errors are repetitive, so this type of testing can be helpful for future reference.

3.3 RELEVANT STANDARDS

The IEEE Standard 829-2008 is used to document software tests [10]. This standard outlines how to document processes and write reports about different types and levels of tests in the field.

The Penetration Testing Execution Standard defines the procedures for penetration testing, which includes black box and white box testing [11, 7, 8]. This standard outlines the tools, research, and vulnerabilities that a tester should look out for.

4 CASE ANALYSIS: THE Y2K BUG

4.1 CASE DESCRIPTION

The Y2K bug was an issue that was brought to the attention of software engineers at the turn of the century. Storage was limited and expensive, so programs that used the date to perform calculations used a shortened 2-digit year instead of the full 4-digit year [12]. This wasn't a problem in the 20th century, when the first two digits of every year were '19'. However, the year '2000' made a 4-digit year necessary to distinguish '1900' from '2000'. Many systems had to be updated, from weapons systems to air traffic control systems, which made a relatively small and simple bug an expensive fix [12, 13]. Figure 2 outlines expenses incurred by prominent corporations to fix the Y2K bug [13].

Table 2. General costs to fix the Y2K problem (The statistics in this table were gathered from Computerworld December 22, 1997;31(51):2,5,6,8 and Computerworld June 22, 1998;32(25):7, 8, 10)(as of December, 1997–8)

Corporation	Y2K budget	Lines of code	People on project
Atlantic Energy	$3.5 M	25 M	7
Canadian Imperial Bank of Commerce	$150 M	75–100 M	250–300
C.R. Bard	$11 M	8 M	10
Merrill-Lynch	$200 M	170 M	300
Nabisco	$22 M	17 M	50–60
Union Pacific	$44 M	72 M	104

Fig. 2 [13]

The government and other companies worked proactively for around 2 years to update their systems [13]. The process was long and tedious because the error was repeated in so many programs. Additionally, part of the problem was convincing management to divert resources toward fixing the bug [12, 13]. Some bugs cause minor inconveniences, and don't need to be fixed immediately. Software engineers had to acknowledge the Y2K bug as a serious problem to obtain the resources needed to fix it.

The Y2K bug also had an effect on the general public. News stories brought the problem to the world's attention, and people responded by hoarding materials [13]. Shopping centers and ports also closed down during New Year's weekend because the managers worried that operating during that time could result in more harm if something did go wrong [13].

Overall, efforts to fix the bug were successful, and no dramatic failures happened when the year switched over [13]. This led conspiracy theorists to state that the Y2K bug was blown out of proportion [13]. However, many smaller or less visible systems such as ATM machines or satellites did fail, showing that the bug was a problem worth concern [13].

4.2 CASE INVESTIGATION

The Y2K bug was a design error that led to logic errors when the year was input as ‘00’. The date system for all the affected programs had a poor design that couldn’t handle future inputs. The software engineers who popularized the two-digit date solution made a decision that fixed a storage issue in the short run but had to be revisited in the future because the solution caused a new problem. Unfortunately, the bug became a much larger and more expensive problem to fix because of the sheer number of systems that were using the two-digit year in their date systems. This design flaw led to a logic error when the year ‘00’ was used. This year caused an unexpected output when the program read the date as ‘1900’ instead of ‘2000’ and performed calculations that subtracted one hundred years instead of adding one. The logic to obtain the four-digit year from the two-digit year was incorrect and resulted in failure.

The root cause of this bug was human error. The software engineers created a poor solution to fix their storage problem and ended up causing a future logic problem instead. Software engineers must consider all potential problems when designing and planning a program.

In the case of the Y2K bug, preventative testing methods were used to discover the potential failure it could cause. The bug was found much earlier than when it was set to become a problem and cause systems to fail. The programmer William Schoen discovered the Y2K problem in 1983 [13]. This allowed the software community time to perform maintenance on important systems and bring them up to date. White box testing is a good method to check if the internal code is using a two-digit year and if a year after 1999 causes incorrect calculations that result in an unexpected output.

IEEE Standard 830-1998 recommends specifying the requirements for software systems [14]. This standard could have been used to plan a better date format because it requires software engineers to outline the needs of the client and the capabilities of the product. If the capability of the product is to record the date correctly, then this standard would allow them to plan for that effectively in the beginning stages.

4.3 RECOMMENDATIONS

In a technology reliant world, it is important to prevent our software systems and programs from failing. The impacts of coding errors can cause real-world effects since many of our power, water, and even weapons systems rely on computers. Software failure can have negative impacts on the health and safety of people and the environment.

To avoid problems like the Y2K bug in the future, it is important to standardize the method of counting dates and other large iterable numbers. The new standard should outline the number of digits required to anticipate future needs, so that costly future maintenance can

be avoided. This would provide a long-term solution for programs that rely on the date or other stored numbers to perform calculations.

In general, software engineering should focus on standardizing languages, operating systems, and development tools in the future to prevent errors like the Y2K bug from being spread to other systems. Without standards, new technology is built off of previous faulty unregulated technology, which allows problems like bugs to continue into the future.

5 REFERENCES

[1] "IEEE Standard Glossary of Software Engineering Terminology," in IEEE Std 610.12-1990, vol., no., pp.1-84, 31 Dec. 1990, doi: 10.1109/IEEESTD.1990.101064.

[2] V. K. Lazarus and S. H. Nggada, *International Journal of Software Engineering and Its Applications*, vol. 12, no. 3, pp. 19-28, Dec. 2018, doi: 10.21742/ijseia.2018.12.3.02

[3] IGI Global, "Handbook of Research on Machine Learning," IGIGlobal.com. https://www.igi-global.com/dictionary/investigation-of-software-reliability-prediction-using-statistical-and-machine-learning-methods/59093#:~:text=1.,their%20impact%20on%20the%20systems (accessed Mar. 1, 2023).

[4] M. Sharma, "Classification of software failures | Software Engineer," IncludeHelp.com. https://www.includehelp.com/basics/classification-of-software-failures-software-engineer.aspx (accessed Mar. 1, 2023).

[5] TotalView, "What Are Software Bugs?," TotalView.io. https://totalview.io/blog/what-software-bugs (accessed Mar. 1, 2023).

[6] ThinkSys, "16 Types of Bugs in Software Testing," ThinkSys.com. https://www.thinksys.com/qa-testing/types-software-testing-bugs/ (accessed Mar. 1, 2023).

[7] T. Hamilton. "White Box Testing - What is, Techniques, Example & Types." Guru99.com. https://www.guru99.com/white-box-testing.html (accessed Feb 13, 2023)

[8] Synopsys, "*Penetration Testing.*" Synopsys.com. https://www.synopsys.com/glossary/what-is-penetration-testing.html (Accessed Feb 17, 2023)

[9] L. Hatton, “Forensic Software Engineering: an overview,” *Computer Science,* pp. 1-12, Dec. 2004, Available: https://www.leshatton.org/Documents/fse_Dec2004.pdf

[10] *IEEE Standard for Software and System Test Documentation*, IEEE 829-2008 , IEEE, New York City, USA, Jul. 18, 2008, doi: 10.1109/IEEESTD.2008.4578383.

[11] *Penetration Testing Execution Standard*, The PTES Team, 2017. Available: http://www.pentest-standard.org/index.php/PTES_Technical_Guidelines

[12] J. E. Schultz, "Managing a Y2K project--Starting now," *IEEE Software*, vol. 15, *(3)*, pp. 63-71, 1998. Available: https://login.lp.hscl.ufl.edu/login?url=https://www.proquest.com/scholarly-journals/managing-y2k-project-starting-now/docview/215832136/se-2, doi: https://doi.org/10.1109/52.676742.

[13] M. Manion and W. M. Evan, “The Y2K problem and professional responsibility: a retrospective analysis,” *Technology in Society*, vol. 22, no. 3, pp. 361-387, 2000. Available: https://www.sciencedirect.com/science/article/pii/S0160791X00000154, doi: 10.1016/S0160-791X(00)00015-4

[14] "IEEE Recommended Practice for Software Requirements Specifications," in IEEE Std 830-1998 , vol., no., pp.1-40, 20 Oct. 1998, doi: 10.1109/IEEESTD.1998.88286.