Journal of Computer and Communications, 2014, 2, 25-35
Published Online April 2014 in SciRes. http://www.scirp.org/journal/jcc
http://dx.doi.org/10.4236/jcc.2014.26004

Software Dysfunction: Why Do Software Fail?

Edward E. Ogheneovo

Department of Computer Science, University of Port Harcourt, Port Harcourt, Nigeria
Email: edward_ogheneovo@yahoo.com

Received 6 February 2014; revised 5 March 2014; accepted 13 March 2014

Copyright © 2014 by author and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

Software is pervasive in modern society, but we are often unaware of its presence until problems arise. Software is one of the most important and yet one of the most economically challenging techniques of this era. As a purely intellectual product, it is among the most labor-intensive, complex, and error-prone technologies in human history. Until the 1970s, programmers were very meticulous in planning their code, rigorously checking code, providing detailed documentation, and exhaustive testing before the software is released to users. However, as computer became widespread, attitudes changed. Instead of meticulously planning code, the attitude of the average programmer today is possibly hacking sessions or writing any sloppy piece of code and the compiler will run diagonally, a situation called, "code and fix", where the programmer tried to fix errors one by one until the software compiled properly. As programs grew in size and complexity, the limits of this "code and fix" approach became evident. In this paper, we studied the various reasons why software fails. Our studies reveal that the major reasons why software fails are poor or no design at all, inadequate testing of codes, and attitudinal changes among programmers and other factors.

Keywords

Software, Software Dysfunction, Software Engineering, Programs, Programmers

1. Introduction

Software is pervasive in modern society, but we are often unaware of its presence until problems arise. Software is one of the most important and yet one of the most economically challenging techniques of this era. As a purely intellectual product, it is among the most labor-intensive, complex, and error-prone technologies in human history [1] [2]. Computer users often experience code bloat, ugly, inefficient and poorly designed code resulting in software dysfunction. It is pretty difficult to find good software that is completely reliable. A good software is

How to cite this paper: Ogheneovo, E.E. (2014) Software Dysfunction: Why Do Software Fail?. *Journal of Computer and Communications*, **2**, 25-35. http://dx.doi.org/10.4236/jcc.2014.26004

a software that is usable, reliable, defect free, cost effective and maintainable. The truth is that most software sucks. Dijkstra [3] notes that "the average computer user has been served so poorly that he expects his system to crash all the time, and we witness a massive worldwide distribution of bug-ridden software for which we should be deeply ashamed."

Software engineering is different from other engineering disciplines. Mann [2] notes that "to overemphasize the uniqueness of software's problem, when automotive engineers discuss the cars on the market, they don't say that vehicles today are no better than they were ten or fifteen years ago. The same is true for aeronautical engineers: nobody claims that Boeing or Airbus makes lousy planes. Nor do electrical engineers complain that chips and circuitry aren't improving. Engineers constantly notice shortcomings in their designs and fix them little by little. Software engineering is a discipline just like any other engineering discipline, yet many software engineers believe that software quality is on the decline. In fact, software quality is getting worse." In the last 25 years or so, software defects have wrecked a European satellite launch, delayed the opening of the hugely expensive Denver Airport for a year, destroyed NASA Mars mission, killed four marines in an helicopter crash, induced a U.S. Navy ship to destroy a civilian airliner, and shut down ambulance system in London, leading to as many as 30 deaths. Between 1985 and 1987, a computer-controlled radiation therapy machine manufactured by the government-backed Atomic Energy of Canada massively overdosed patients in the United States and Canada, killing at least three. This was due to software error arising from poor or inadequate design or no design at all.

Today, we rely more and more on sophisticated, software-based systems for mission-critical, business-critical and safety-critical applications. Companies can no longer write all of the software themselves; they often buy commercial-off-the-shelve (COTS) software and making use of Open Source software [4]. The result is that most of the software often contains faults which may not manifest immediately but manifest when the software is operational. These faults that manifest themselves only when the software is operational but fails to manifest itself when the software is dormant are called dormant faults. The presence of faults in code often causes errors when a function or method is called from a code segment containing the fault. The error in turn causes a system failure. In general, software is implemented based on its design. Thus software failure is the product of a design flaw [5].

At each stage of software development, errors may be introduced into the software. For instance, the requirement analysis may be incomplete, the design may be inadequate or omitted entirely. The earlier an error is introduced into the software during the Software Development Life Circle (SDLC), the more severe and costly its impact is likely to be because the error is expanded in subsequent stages of the development. Errors are discovered and fixed during the testing stage of SDLC. However, in actual fact, it is difficult to discover all errors. These undiscovered errors remain in the software and are dormant until the software become operational and they will start to manifest themselves. This is particularly true for large and complex software.

These anomalies are not only restricted to system level but also to component and subroutine levels which will eventually affect the system's normal operation. Consider a C++ code segment that computes the division of two integer values using variables x and y with x having a value 5 and y having a value 2.

```
double IntDivision (int x, int y)
{
  return x/y;
}
```

Since x = 5 and y = 2, the result of the computation will be 2 instead of 2.5. This is because the division of two integer values will always be an integer since the fractional part is discarded or truncated. If the computation were to be financial calculation, it would cause system failure and the consequence would be financial loss. This could be very disastrous especially if the computation involves large sum of money.

Software failure is very common with airplane crashes. Just as pilots never intend to crash, software developers do not aim to fail. When a commercial plane crashes, investigators look at many factors, such as the weather, maintenance records, the pilot's disposition and training, and cultural factors within the airline. Software failure jeopardizes an organization's prospects. If the failure is large enough, it can steal the company's entire future such that the airline could be permanently closed due to lack of customer's patronage as a result of lack of confidence on the airline and its operators.

It must be noted that large-scale software tends to fail three to five times compared to small ones. The larger the software, the more complex it is. Greater complexity increases the possibility of errors, because no one really

understands all the interacting parts of the whole or has the ability to test them. Thus the greater the software, the more complex it is both in its static elements (the discrete pieces of software, hardware, etc.) and its dynamic elements (the couplings and interactions among hardware, software, and users; connections to other systems, etc). It suffixes to say, therefore, that it is impossible to thoroughly test software of any real size. Pressman [6] notes that "exhaustive testing presents certain logical problems… Even a small 100-line program with some nested paths and a single loop executing less than twenty times may require 10 to the power of 14 possible paths to be executed… To test all of these 100 trillion paths assuming each could be evaluated in a millisecond, would take 3170 years". The construction of new software devoid of errors that is pleasing to both the buyer and the user is perhaps the most difficult problem facing software engineers today. This is because software engineers do not devote enough time to the design of their software.

2. What Is Software?

Software can simply be defined as computer programs and associated documentation [7]. It consists of a number of separate programs, configuration files, which are used to set up these programs, system documentation, used to describe the structure of the system, and user documentation for explaining how the software should be used. Thus software is a general term used to describe all programs which control the activities of a computer system. The term program describes a sequence of instructions given to a computer to guide it in processing information. There are two major groupings of software system. These are: generic software and customized software. Generic software are software produced by software development companies and sold to users who want to buy them in open market while application software are software made for a particular application, that is, those software developed by software engineers targeted at solving a particular problem.

2.1. Attributes of a Good Software

- **Functionality:** Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and acceptable.
- **Maintainability:** Good software should be easy to maintain, it should be able to evolve so as to be able to meet changing requirements.
- **Dependability:** The software must be trustworthy. It should be reliable, secure, and safe.
- **Efficiency:** The should not make wasteful use of system resources
- **Acceptability:** The software must be accepted by the users for which it was designed. This means it must be understandable, usable and compatible with other systems.
- **Survivability:** Good software should be able to withstand and adapt to any environment where it is put to use.

2.2. Critical Software

Critical software systems are technical or socio-technical systems that people or businesses depend on. They are very strategic in the life of individuals and businesses. The effects of the failure of critical software could be very catastrophe if not properly managed. Critical software, ironically, often have characteristics that make failures more likely such as: real-time constraints, concurrency, and harsh physical environments. The three main types of critical software system according to [7] are:

- **Safety-critical:** These are software used for protecting human lives. These software are used in areas such as cars and aircrafts, chemicals and nuclear plants, medical equipments, etc. Most safety-critical systems have real-time constraints. The throttles and brakes of a car must respond immediately the brake is pressed. Aircraft must also respond promptly to their controls to avoid accidents which could result in loss of lives and property.
- **Mission-critical:** These are software used for essential tasks such as telephone routing, financial transactions, stock control, house-hold appliances e.g., televisions, videos, etc. Sometimes, these appliances may be recalled by their manufacturers if the software fails. This could be very expensive in terms of providing maintenance and installing new software.
- **Business-critical:** These are software used for protecting confidential information. They are often employed in areas such as banks and other financial institutions, government applications, military applications, etc.

2.3. What Is Software Failure?

Software failure can be explained from the software-centric and system-centric approaches. According to [8], the software-centric approach view "failure" as a property of the software itself. That is, the software is considered in isolation, and not the context of the system in which it operates. The system-centric approach view "failure" in relation to the system. That is, the approach considers software failure within the system itself. The system-centric approach is similar to the modeling of human performance, an unsafe human act is considered harmful only in the context of the system within which it occurs. Therefore, a failure occurs when a system fails to perform its required function(s) [9]. As noted in [10], discovering the unexpected is more important than confirming the known. Thus in software development, the "unexpected" one relates to defects. These defects when unattended to often cause failure [11] [12]. Thus software failure is an incorrect result to the specification or unexpected software behavior perceived by the user at the boundary of the software system, while a software fault is the identified or hypothesized cause of the software failure. Therefore, fault is the cause of software failure while failure is the effect that occurs as a result of the fault. Thus fault is the cause of failure [13]. **Figure 1** shows the relationship between defect, fault, and failure.

To understand software failure, we present the definitions of some concepts from [14] used in this research work. These are:

Fault/Defect: An incorrect step, process, or data resulting in an incorrect result. Therefore, faults are concrete manifestations of errors within the software. One error may cause several faults and various errors may cause identical faults. Faults are problems in the code.

Failure: The inability of a system or component to perform its required function within the specified performance requirement. A failure occurs when the user perceives that a software program ceases to deliver the expected service. The user may decide to identify the various levels of failure, such as catastrophic, which could be major or minor, depending on their impacts and consequences on the system such as monetary value (cost), human life, and property lost [15]. Thus failures are departures from the operational software system behavior from user expected requirements. A particular failure may be caused by several faults and some faults may never cause a failure. Also, a system may be reliable but not correct, *i.e.*, it may contain faults but if we never execute those faults, it is reliable. On the other hand, if we define correctness as the conformance of the code to the specification, a system may be correct but not reliable because the user may try to use the system in ways not permitted in the specification and the system may crash. Failures are incorrect external events.

Errors: These are the differences between computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. They are defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools. It is also defined as the discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. Errors occur when some part of the computer software products result in an undesired state.

Mistake: This is a human action that results in software containing a fault. It is a human action that produces incorrect result. Examples include omission or misinterpretation of user requirements in a software specifications, and incorrect translation or omission of a requirement in the design specification.

Debugging: This is a diagnostic process where, given a failure, an attempt is made to find the associated fault.

Mean-time-to-Failure (MTTF): The expected time that the next failure will occur (observed). It is the hours of operation divided by the number of failures.

Mean-time-to-Recover (MTTR): The expected time until a system will be repaired after a failure is observed.

Types of Software Failures

Different types of software failures can be identified. These are:

- Process failure
- Real-Time anomalies

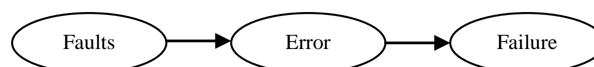

Figure 1. Relationship between fault, error, and failure.

- Accuracy
- Abstraction
- Constraint
- Reuse
- Logic
- Faulty code
- Operator error

Process Failure: Process failure occurs as a result of human errors or negligence arising from failures in development such as poor development methodologies and errors in operation. A good example of process failure is the Therac-25. The Therac-25 was a radiation treatment machine malfunction for the treatment of cancer and tumors. The machine malfunction by delivering excess dose of radiation and caused the death of six people. This was due to software error arising from human error and inaccuracy. Another case of process failure occurs on August 6, 1997, Korean Air Flight 801 crashed into a rocky hillside in the middle of Guam. Of the 254 passengers on board, 228 were killed due to negligence of the flight crew of system failure resulting from software modification which could not accurately cover the required radius of 55 nautical miles, 102 kilometers.

Real-Time Anomalies: Software bug made soyuz stray. A computer software error sent a Russian spacecraft into a rare ballistic descent that subjected the three men on board to chest-crushing gravity loads that made it hard to breath. Their capsule landed nearly 300 miles off-target.

Approximation/Accuracy: Loss of precision in converting from integer to float or division of two integers where the fractional part is discarded or round-up to integer since the division of two integer values gives an integer. This is precision loss which can result in error if not properly handled. A good example of failure caused by precision error is in Patriot Missile that embeds software where the fault in the software caused the missile to miss target.

Abstraction: The Y2K problem actually caused bugs such as incorrect interpretation of 99 to 00, that is, the algorithm incorrectly interprets year 2000 as 1900. This occurs as a result of lack of data abstraction which caused poor encapsulation of year data. This caused the date to spread throughout the code without abstraction mechanisms.

Constraint: Typical examples: stray pointer and buffer overflow. These days, however, recent languages such as Java and C# (C-Sharp) have constraint checking on data types that helps the restriction on the data type that cannot be violated. The use of "Sandbox" also helps to guide against malicious faults.

Reuse: Software reuse is a technique for building software to address the need to improve software development efficiency and quality. It involves the use of artifacts from existing systems to build new ones in order to improve quality and maintainability and to reduce cost and development time [16]. Software reuse is a sought after goal. It is aimed at getting the most from design and implementation work that is being done. However, software that contains bugs will always result in failure whenever such software is reused and the bug manifests itself. Therefore, it is necessary to verify and test software to ensure that there are no bugs that could cause failure before such software are reused.

Logic: These are "obvious" flaws in logic processing. A good example is the AT&T failure of 1990. The failure was attributed to software error resulting from software upgrade of switch. The upgrade caused the switch to develop errors and these errors are routed traffic to other switches while trying to reset the switch by sending "out of service" message. This message caused other switches to crash, sending "out of service" message, thus propagating the problem. The failure was caused by a missing "break" statement (C++, Java, C# languages requires the break statements) in the software that controls the switch. The failure caused about 9 hours crash which caused about $60 million lost in revenue.

Faulty Code: These are poorly written codes which often cause buffer overflow and runtime problems. Faulty code is not exercised unless discovered by malicious hackers.

Operator Error: Most times errors that occur in system software are due to mistakes on the part of the operators. A computer operator may be inexperienced in the use of computer or a particular operating system or even application software. Thus when using the software, he may perform some actions that may introduce errors or faults into the program.

Examples of Critical Software Failure

As software becomes increasingly important, the potential impact of bad code will increase to match. Software defects have caused a lot of devastating effects in the last few decades. Software failure are very prevalent

everywhere. Here, we present some examples of cases of software failure:

- **Ariane-5 Satellite Launcher:** On June 4 1996, there was total failure of Ariane-5 launcher on its maiden flight. Approximately 40 seconds after takeoff, the Ariane-5 rocket launcher was shut down and then lost control due to software failure resulting from internal variable exceeding a limit imposed by the underlying code which resulted in buffer overflow. It was reported that incorrect control signals were sent to the engines and these caused swiveled which caused it to break up and self-destructed. The software failed and the system and the backup system shut down. Although it was reported that the same software worked perfectly for a previous version of the rocket (Ariane-4 rocket launcher), which was a smaller version, it failed completely for Ariane-5 launcher. The reasons are (1) the Ariane-4 launcher was smaller and as such has a lower initial acceleration and build up of horizontal velocity than Ariane-5; (2) the value of the variable on Ariane-4 could never reach a level that caused overflow during the launch period [17]. Ironically, the code that failed was generating information that was not even necessary as soon as the launch had started.
- **The Therac-25:** This is a radiation therapy machine used to administer drug to patients. The machine administered massive overdoses of the drug resulting in the death of patients. This was caused by software upgrade. The software upgrade allowed the operator to type faster when entering data. The parts of the machine that were not upgraded could not cope with the speed of data entry, thus causing the machine to administer the wrong doses. It took several weeks of intensive investigation to determine the cause of the problem.
- **The US Navy Ship Yorktown:** In 1997, a prototype US Navy ship while trying to adjust a valve setting, a sailor mistakenly entered a zero into a database field; the ship was out of action for three hours. The fault was caused by a software failure due to improper input (data) field validation. The software program that the sailor was using on Windows NT should have refused to accept his command but did otherwise, and the program shut down the system. This is an operator error. However, the software was not properly designed. It should have being designed to recognize and reject an illegal command by throwing exception [18] [19].
- **The WMF Bug:** There are many instances in which Microsoft software had failed particularly security vulnerabilities that have given the company a reputation for buggy code. These faults resulting from Microsoft software products have made customers to shift attention to other software products. These days, it must be noted that not only the defects but the reaction to a software problem, requires serious attention by software providers.
- **The Marines' Osprey Crashes:** In December 2000, the US Marine Corps' new hybrid plane-helicopter, known as the V-22 Osprey, crashed, killing four marines. The accident was largely due to software error.
- **The Mars Climate Orbiter Loss:** In 1999, NASA lost contact with one of its well-publicized unmanned spacecraft, the Mars Orbiter. This was largely due to software failure which failed to convert different units of measurement. The "root cause" was software failure. The spacecraft failed to translate English units into metric units in a segment of ground-based, navigation-related mission software. Project managers failed to specify the system of measurement that was used by the subcontractors that developed the software as a result of the different metric systems that were used. Thus coding error caused it to overwrite two memory addresses, thus resulting in buffer overflow.
- **US Telephone Networks Failure:** In the summer of 1991, telephone networks in several major United States cities suffered failure. The problems were traced to telephone switching software consisting of several million lines of code. The potential losses were enormous [20].
- **The Bank of New York:** On November 20, 1995, the bank of New York's securities transaction software had a storage fault. For about ninety minutes, it lost information on $32 billion in transactions. The bank was forced to borrow $23.6 billion from the US Federal Reserve for a day, at a cost in interest of $5 million. Although the information was later recovered, the bank's reputation suffered as customers began to doubt the bank [20].
- **US Patriot Air Defense Collided with Incoming Scud Missile:** In the 1991 Gulf War, 28 US troops lost their lives when the Patriot air defense system collided with an incoming Scud Missile. The problem was caused by software error resulting from software upgrade. An upgrade to deal with missile was made, but at one place in the upgraded software, a necessary call to the subroutine was accidentally omitted. Although, the problem was detected, a warning was issued, and a software patch dispatched to users more than a week before the accident occurred. There was need to reboot the engine occasionally to correct the error which would have taken only one minute. There were warnings but the battery managers refused to heed to the warnings [21].

- **Mass Polar Lander (MPL):** Like the Mars Orbiter, the Mars Polar Lander (MPL) free fell to the surface due to software failure resulting in the shutdown of the engines of the MPL. The software interpreted spurious signals at the deployment.
- **Titan/Centaur/Milstar:** In 1999, an airspace rocket called Titan IV B-32/Centaur TC-14/Milsrar-3 lost control and crashed due to software error. An incorrect roll rate filter constant zeroed the roll rate data, resulting in the loss of roll axis control and then pitch control. The loss of altitude control caused excessive firings of the reaction control system and subsequent hydrazine depletion. The accident investigation board concluded that the failure of the Titan IV B-32 mission was due to an inadequate software development, testing, and quality assurance process for the Centaur upper stage.
- **SOlar Helicopter Observatory (SOHO):** NASA Project, The SOHO spacecraft, on June 25, 1998 got lost. The loss was due to overconfidence and complacency resulting in inadequate testing and review of changes to software.

2.4. Classification of Failures

Jalote *et al.* [14] both at Microsoft Corporation proposed a classification of failure as:

- Unplanned events,
- Planned events, and
- Configuration failure

Unplanned Events: These are traditional failures like crash, hang, incorrect output or no output at all. These are caused by software failure. Other forms of unplanned events are: disasters, system errors, employee error, application error, operations overruns, utility failure, hardware failures, functionally incorrect response, and untimely response such as too fast or too slow.

Planned Events: These events often cause system to shut down in a planned manner to perform some housekeeping tasks. Examples include updates requiring restart, configuration changes requiring a restart.

Configuration Failures: These failures occur as a result of due to configuration setting. In many systems, configuration failures account for a large percentage of failures [22]. Examples include application/system incompatibility error, installation/setup failures.

2.5. Causes of Software Failure

A Lack of Logic: Poor or no designs at all. Most often, software engineers or developers do not have a good design before coding or writing programs or software. This is the major cause of software failure today. Developers most times boot their computers and navigate to the programming language location and they will start coding without having a design. Such software is bound to fail. It is like an architect that is building a house without plan. Soon he will discover that some parts of the house have errors. What he does is simply to demolish those parts and start building again. Building a house this way will waste time and costs more, yet the house architect may never correct all the errors in the house. In fact, the house may even collapse at in no distance time. The same is true of the software. Once software does not have a very good design, the software is bound to fail as it is put into use. Again, most developers of software often have one major or comprehensive design called high-level design. For software to be properly developed and have minimal errors, the software must have both high-level and low-level (or detail) design. Before coding, both the high-level and detail designs must be properly done and correction made so as to minimize the errors that might arise from the software.

Inadequate software testing: Most often, software developers do not adequately test their code before releasing them to customers. For software to be reliable and free of defects, the software must be properly debugged, rigorously tested. Defects in the code must be located and corrected such that the program meets its requirements. Testing must be repeated severally to ensure that the changes have been correctly made. Also, the programmer should consider testing the software in all ways that might be anticipated by the user. The programmer should test the software with all available data both imaginary and real to ensure that the system is able to handle them without crashing. This kind of robust testing will in no doubt help the programmer to remove faults from the software as much as possible.

Attitudinal changes among programmers: Until the1970s, programmers were very meticulous in planning their code, exhaustively and rigorously checking their code, providing standard and elaborate documentation and peer-vetting, and providing exhaustive testing before the software is released to users. However, as computer

became widespread, attitudes changed. Instead of meticulously planning code, the attitude of the average programmer today is possibly hacking sessions or writing any sloppy piece of code and the compiler will run diagonally, a situation called, "code and fix", where the programmer try to fix errors one by one until the software compiled properly. As programs grew in size and complexity, the limits of this "code and fix" approach became evident.

In addition to these factors, Robertson and Williams [23] proposed the following reasons for software failure.

Software changes introduce incompatibilities: Software especially large and complex software must evolve during its lifetime if they must remain useful. As such, when they evolve, incompatibilities and errors are initially not in the software may now be introduced which may result in software failure.

Software is attacked by a hostile agent: In this case, there is change in environment where the software is applied. Here, the change is done explicitly with the intent to cause the software to fail. For example, the environment in which the software is being used might have adverse effect on the software. Some time, the software user might intentionally delete some part of the software so that the software will no longer perform its functions properly thus resulting in failure.

Failure resulting from unanticipated applications or use: Most often, failure can occur if a software product is used in such a way that the software developer did not anticipate users might put the software into. For example, in embedded systems or applications, the number of ways the environment can change becomes so large that the programmer cannot realistically anticipate every possible failure.

In addition to failures related to the faults introduced in the software during its software development life cycle (SDLC), software may also fail due to external causes. These are:

- **Human error:** Using software in inappropriate way, input incorrect data into the software, attempting to divide by zero often result in errors that could cause critical failures, which in turn could result in the loss of human lives and property.
- **Management laxity:** Most times, before a failure occurs, the fault must have given some warnings which most times management often fail to ignore for reasons best known to them. Such reasons include consideration of effort and costs that such faults may cause the organization.
- **Support systems:** software used to perform complex tasks such as controlling some devices, need some level supporting systems, e.g., operating system computer hardware, electric power, etc.
- **Cyber Security:** cyber threats such as viruses and hacking activities can undermine the capabilities of software system due to software vulnerabilities especially if such threats are transmitted through the network.
- **Environment:** Natural catastrophes such as fire outbreak, flooding, lightning, earthquake, etc., can affect the computers that embedded the software.

3. Testing of Software Systems

Software testing is a process of verifying and validating that a software system meets the business and technical requirements and works as expected. Testing is the process of evaluating a system or its component(s) with the aim to finding whether it satisfies the specified requirements. This activity results in determining the difference between the actual and expected results. System testing also identifies defects, flaws, or errors in the code that must be fixed. ANSI/IEEE 1059 standard defines testing as 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. From the foregoing, it can be stated that testing is not just about finding bugs but it also encompass the process of ensuring that the software meets its requirements in terms of the outcome of the result(s).

Testing is very important at each stage of software development process. Every time software is developed, the software must be well tested in order ensure that it is of high quality and that it is very reliable. In other words, there should be robust testing of software before it is released to the user or customer. According to [23], in testing software; "think diabolically! Think of every possible thing a user could possibly do with your system to demolish the software. You need to make sure your program is robust—in that it can properly respond in the face of erroneous user input." This type of testing is called robustness testing, whereby test cases are chosen outside the domain to test robustness to unexpected erroneous input [10], and is included in defensive testing which includes tests under both normal and abnormal conditions [3] [24]. Knuth [25] lent credence to this when he notes that "my test programs are intended to break the system, to push it to its extreme limits, to pile complication on complication, in ways that the system programmer never consciously anticipated. To prepare such test

data, I get into the nearest, nastiest frame of mind that I can manage, and I write the cruelest code I can think of; then I turn around and embed that in even nastier constructions that are almost obscene." Therefore software must be well tested at all stages of the development process and the final product should be exhaustively tested to ensure that errors will be very minimal if any at all before it is released to users to ensure reliability of the software.

Types of Software Testing

IEEE Standard Glossary of Software Engineering Terminology, IEEE Standard 610.12-1990, identified three types of software testing. They are:

- Performance testing
- Regression testing
- Robustness testing

Performance Testing: This is the testing conducted to evaluate the compliance of system or component with specified performance requirements.

Regression Testing: This is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

Robustness Testing: This is testing whereby test cases are chosen outside the domain to test robustness to unexpected, erroneous input.

4. Evaluation of Software Failure

Microsoft Office Systems 2003 through its Customer Experience Improvement Program (CEIP) technology. CEIP is an elaborate, programmable, event recording system for products, which can be used to record both failure data and usage data. In order to be able to record failures of a software product through CEIP, the product must be programmed to record events using CEIP provided Application Programming Interfaces (APIs). Generally, for capturing failure information, three types of events are captured. These are 1) application termination events which record normal exit, crash exits, hangs, and user forced exits. Some of these exit events are recorded by a handler that is executed before exiting while others are identified and recorded at the restart of the application using some tracking mechanisms. These events are used mostly to identify crashes and hangs, 2) assert failure, this alert is often used to identify failures, and 3) alerts which is given when some special situations arise (e.g., file does not exist, network not available, file writing fails, etc.). Many of these alerts signify failures. These alerts usually provide additional information in order to determine the cause of a failure whenever such failure occurs. Thus the alert events are used to identify configuration failures. However, the assert failure usually record separate events from the alert events.

Table 1 shows a sample of a report of software failures collected as recorded in CEIP. Data about different types of failures is recorded and their failure rates were also determined. Quite frequently, managers' focus on crash-failures and hang-failures; since these are most disruptive for users, thus users attach highest weight to these failures. The table shows a part of a sample report that was generated by CEIP tools. The table shows examples of some products, number of sessions, the number of crash failures, and the number of hang failures, and

Table 1. An example of CEIP report on software failures.

Product	No. of Sessions	No. of Crash Failures	No. of Hang Failures	Session Length (meters)	Crash Failure Rate (per hr)	Hang Failure Rate (per hr)
A	33,000	300	1000	3,140,000	0.0057	0.0191
B	422,000	1200	8700	46,450,000	0.0015	0.0112
C	20,000	100	700	2,540,000	0.0023	0.0165
D	24,000	100	1000	5,940,000	0.0010	0.0101
E	153,000	600	3300	12,920,000	0.0027	0.0153
F	12,000	100	200	900,000	0.0066	0.0133
G	648,000	2600	29,900	183,530,000	0.0008	0.0097

total session length. Failure rates for both crash failure and hang failures are then computed as number of failures per session or number of failures per hour of usage. Also, from this table the mean time to crash and hang and how they evolve with time can also be calculated based on past data.

5. Conclusions

As software becomes increasingly important, the potential impact of bad code will increase to match. The construction of new software that is pleasing to both user and buyer, which does not contain errors, is an unexpectedly hard problem. It is perhaps the most difficult problem facing software engineers today. Software defects have caused a lot of devastating effects in the last few decades. As software becomes increasingly important, the potential impact of bad code will increase to match. The real problems, however, lie in software's basic design or rather, its lack of design. During design phase, there should be high-level design and detailed design where each component in the design is properly analyzed and tested to avoid design flaws. Some level of software failure will always be with us whether we like it or not as long as software practices and failure rates remain as they are today. Indeed, we need true failures—as opposed to avoidable blunders—to keep making technical and economic progress. But too many of the failures that occur today are avoidable. Given today's IT practices, failure is a distinct possibility; and it would be a loss of unprecedented magnitude. Patents and taxpayers will ultimately pay the price for the development, or the failure, of buggy software. And as society comes to rely on IT systems that are ever larger, more integrated, and more expensive, the costs of failure may become disastrously high.

Finally, software failure tended to resemble the worst conceivable airplane crash, where the pilot was inexperienced but exceedingly rash, flew into an ice storm in an untested aircraft, and worked for an airline that gave lip service to safety while cutting back on training and maintenance. If you read the investigator's report afterward, you would be shaking your head and asking, "wasn't such a crash inevitable?" The same is also true of software. Yet failures, near-failures, and plain old bad software continue to plague us, while practices known to avert mistakes are shunned. It appears that getting quality software is not an urgent priority in most organizations. The tragedy of software engineering is not that we don't know how to plan and conduct software rigorous and elaborate test, but that we know how and just don't want to do it.

References

[1] Kumaresh, S. and Ramachandran, B. (2012) Defect Prevention Based on 5 Dimensions of Defect Origin. *International Journal of Software Engineering & Applications* (*IJSEA*), **3**, 87-98. http://dx.doi.org/10.5121/ijsea.2012.3407

[2] Nggada, S.H. (2012) Software Failure Analysis at Architecture Level Using FMEA. *International Journal of Software Engineering and Its Applications*, **6**, 61-74.

[3] Dijkstra, E.W. (1976) A Discipline of Programming. Prentice-Hall, Upper Saddle River.

[4] IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, 1990.

[5] Pressman, R.S. (2007) Software Engineering: A Practitioner's Approach. 6th Edition, McGraw-Hill, Boston.

[6] Rothenberger, M.A., Dorley, K.J., Kulkarin, U.R. and Nada, N. (2003) Strategies for Software Reuse: A Practical Component Analysis of Reuse Practices. *IEEE Transactions on Software Engineering*, **29**, 825-837. http://dx.doi.org/10.1109/TSE.2003.1232287

[7] Storey, N. (1996) Safety-Critical Computer Systems. Addison Wesley Longman, London.

[8] Chu, T.L., Martinez-Guridi, G., Yue, M. and Lehner, J. (2006) A Review of the Software-Induced Failure Experience. American Nuclear Plant Instrumentation Control and Human Machine Interface Technology.

[9] Robertson, P. and Williams, B. (2006) Automatic Recovery from Software Failure. *Communications of the ACM*, **49**, 41-47. http://dx.doi.org/10.1145/1118178.1118200

[10] Box, G. and Hunter, S. (2010) 25 Great Quotes of Software Testers. https://hexawise.com/posts/25-great-quotes-for-software-testers

[11] Mann C. C. (2002) Why Software Is So Bad, Technology Review, July/August, 2002, 33-38. www.technologyreview.com,

[12] Chillarge, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K. and Wong, M.-Y. (1992) Orthogonal Defect Classification—A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*, **SE-18**, 94-96.

[13] Hobbs, C. (2011) The Changing Nature of Software Failure. China Technology Innovation Conference.

[14] Jalote, P., Murphy, B., Garzia, M. and Errez, B. (unpublished) Measuring Reliability of Software Products.

[15] Weiner, L.R. (1993) Digital Woes. Addison-Wesley, Boston.

[16] Slabodkin, G. (1998) Software Glitches Leave Navy Smart Ship Dead in the Water. Government Computer News, July 13. http://gcn.com/Articles/1998/07/13/Software-glitches-leave-Navy-Smart-Ship-dead-in-the-water.aspx

[17] Ehrlich, W.K., Iannino, A., Prasanna, B.S., Stampfel, J.P. and Wu, J.R. (1999) How Faults Cause Software Failure: Implications for Software Reliability Engineering. Int'l Symposium on Software Reliability Engineering, Austin.

[18] Smedley, R. (2009) Trusted Software? Between the Button and the Bomb, Is There an Operating System You Can Trust? LinuxPro, March, 70-73. http://www.linuxformat.co.uk

[19] Sommerville, I. (2006) Software Engineering. 8th Edition, Addison-Wesley, Boston, 5-6.

[20] Williams, L. (2006) Testing Overview and Black-Box Testing Technique, 34-59. http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf

[21] Carlone, R.V. (1991) Patriot Missile Defense: Software Problems Led to System Failure at Dhahran, Suadi Arabia. http://www.ima.umn.edu/~arnold/disasters/patriot.html

[22] Chillarege, R. (1996) What Is Software Failure? *IEEE Transactions on Reliability*, **45**, 354-355. http://dx.doi.org/10.1109/TR.1996.536980

[23] Lyu, M.R.-T. (2002) Software Reliability Theory, Encyclopedia of Software Engineering. John Wiley & Sons, Inc, Hoboken.

[24] Charette, R.N. (2005) Why Software Fails [Software Failure]. *IEEE Spectrum*, **42**, 42-49. http://dx.doi.org/10.1109/MSPEC.2005.1502528

[25] Krasner, H. (1998) Using the Cost of Quality Approach for Software. *CROSSTALK*: *The Journal of Defense Software Engineering*, **11**, 6-11.

[26] Dowson, M. (1997) The ARIANE-5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, **22**, 84. http://dx.doi.org/10.1145/251880.251992

[27] Knuth, D.E. (1992) The Errors of TeX: Software Practice and Experience, in Literature Programming. *CSLI Lecture Notes*, **19**, 607-681.