Mikko Jaaksola

Software testing failures through the history and how to prepare for them

Information Technology

Helsinki Metropolia University of Applied Sciences

Master's Degree

Information Technology

Master's Thesis

4 December 2018

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Mikko Jaaksola 54 pages 4 December 2018
Degree	Master´s degree
Degree Programme	Master´s degree in Information Technology
Instructors	Harri Airaksinen, Director Ville Jääskeläinen, Principal Lecturel

Since the day when first software development projects started there has been both successful and unsuccessful project outcomes. From the 50´s to today, IT projects has become more and more complex and project timetables more and more stricter. This has introduced new challenges to software testing. When the testing is not at the required level, the consequences may be catastrophic.

This study reveals some of the biggest software failures in the history. These cases were picked from different centuries, starting from 50´s. Each case stands as an example of how software testing was not at required level. The idea was to pinpoint the main reasons why these projects have failed and to find the solutions how the failure could have been avoided. This study focused to solve how the software testing and software errors are studied and documented. How much the software errors may cost if they are not noticed during the software project and these errors are found out later after the project has gone online. Also, how much software bugs cost to solve during a project was taken into investigation.

Main purpose of this study was to point out the importance of software testing and how much money may be lost in badly planned and failed IT/IS projects. The original scope was to focus only in software testing and how that has been documented. Based on that idea a testing framework was created that could improve software testing. After a short description about how failed testing cases were documented the scope moved to focus on how much losses were badly tested IT/IS projects causing. Keeping the original scope in mind some testing related main roles, testing processes and tools are presented.

The results of the study show that failed software projects picked to this thesis caused over 474 milliard euros losses. This included 57 projects from 1962 to year 2018. Project cancellation percent in these cases was 29,8%. Study also shows that when talking about software testing and project planning, the same issues seems to be on the table today that were already noticed 50 years ago.

Keywords	Software testing, failure, software bug, cost, losses

Contents

Abbreviations/Acronyms

5GL	*Fifth-generation programming languages*
AECL	*Atomic Energy of Canada Limited.* Company that manufactured Therac-25 radiation therapy machines
ASM	Also known as *Assembly*. It is a low-level programming language for a computer or other programmable device
CLR	*Common Language Runtime.* The virtual machine component of Microsoft's .NET framework manages the execution of .NET programs
COCOMO	*Constructive Cost Model*. It is a procedural software cost estimation model
CSS	*Cascading Style Sheets.* Language used for describing the presentation of a document written in a mark-up language
DBCS	*Double-byte Character Set.* Character encoding in which either all characters (including control characters) are encoded in two bytes, or merely every graphic character not representable by an accompanying single-byte character set (SBCS) is encoded in two bytes
ERB	*Earth Radiation Budget.* Balance between incoming energy from the sun and the outgoing longwave and reflected shortwave energy from the Earth
ESA	*European Space Agency*. Intergovernmental organisation of 22-member states dedicated to the exploration of space

GDP	*Gross domestic product.* Is a monetary measure of the market value of all final goods and services produced in a period of time
INS	*Inertial Navigation System*. A navigation aid that uses a computer, motion sensors (accelerometers), rotation sensors (gyroscopes), and occasionally magnetic sensors (magnetometers), to continuously calculate via dead reckoning the position, orientation, and velocity (direction and speed of movement) of a moving object without the need for external references
ITIL	*Information Technology Infrastructure Library*, is a set of detailed practices for IT Service management (ITSM) that focuses on aligning IT services with the needs of business
ITK method	Methodology to evaluate complexity and cost of developing and maintaining application software for creating information systems (also known as Method CETIN). It is an algorithmic model assessment value software, developed by consortium of Kazakh IT companies
LOC	*Lines of Code.* Method to estimate the size of software project. Sometimes also called SLOC (Source Lines of Code)
NASA	*National Aeronautics and Space Administration*. USA space program
PL/I	*Programming Language One*. It is a procedural, imperative computer programming language designed for scientific, engineering, business and system programming uses
PRICE	It is generally acknowledged as the earliest software for parametric cost estimation developed in the 1970´s

SBUV	*Solar Backscatter Ultraviolet*. Method that measure ozone layer concentrations over globe
SLIM	*Software Lifecycle Management*. It is an empirical software effort estimation model published in 1978
SQL	*Structured Query Language.* Special-purpose programming language designed for managing data held in a relational database management system, or for stream processing in a relational data stream management system
TOMS	*Total Ozone Mapping Spectrometer.* Instrument that detects ozone
UI	*User Interface.* The space where interactions between humans and machines occur
VV&T	*Verification, Validation and Testing.* Independent procedures that are used to together for checking that a product, service or system meets the requirements that are described to fulfil its purpose

1 Introduction

The scope of this thesis is to study the importance of software testing. How software testing has changed through the history and what could be consequences if testing fails. For this purpose, testing cases were taken into deeper investigation and examined what have happened and if possible, how that could have been prevented.

Need for the study came from my previous job experiences. I was in a role of project management in a continuous delivery and a continuous service side in big IT company. Small and bigger software changes were daily tasks and testing was in a big role of that. Through the years I had cases were testing was done perfectly and in some cases the quality was far from perfect. I noticed an unpleasant trend that when the timetable was strict, almost always testing was the phase were time was spared. Of course, there are cases where testing requires only a very small amount of work and can be quickly done, but there are also cases where a small software change, that seemed to be easily tested, could have had unplanned effects to the functionality of the product. For that reason, I got interested of studying how the importance of testing has been documented and how the lack of proper testing has been documented.

The main purpose of this study is to find out how much money is wasted in failed IT/IS projects annually. How many cases are software testing related and what have went wrong in these cases. Idea is to bring awareness in the field of IT, why good planning and testing is so important and what could be the consequences if these are not taken into consideration. One mission is also to give some advices and answers on how to avoid making mistakes and how to improve testing in general.

Structure of the Thesis

Chapter 2 includes fundamentals of software testing and research plan. In this chapter, the background, main problem and scope of the thesis is presented. Chapter 3 includes investigated software projects. In this chapter, the selected case studies and what went wrong and why, from the testing point of view is explained. Their financial losses have been investigated and presented. Chapter 4 includes theoretical background. In this chapter, the history of software testing is investigated. Different periods are introduced

and how the testing has changed through the history. Chapter 5 includes tips about how to improve testing. In this chapter, the necessary roles regarding software testing and what are the main responsibilities of these roles is outlined. Chapter 6 includes data analysis and findings. In this chapter, the analysis of data gathered IT projects is revealed. Key findings are also presented in this chapter. Chapter 7 includes thesis summary. In this chapter, the results from the study is concluded. Chapter 8 includes the complete list of references used in this thesis.

2 Fundamentals of Software Testing and Research Plan

The testing of software is the most important phase of software development, when talking about software quality. Testing typically consumes 40-50 % of development efforts and consumes more effort for systems that require higher levels of reliability, it is a significant part of the software engineering, wrote J.J. Marciniak in year 1994. (Marciniak, 1994). With the development of programming languages and stricter implementation project schedules, testing process raises to be even more important in the future. The amount of software in a device doubles roughly every 18 months. As a result, there are more and more pieces of software functioning within the same system, requiring more and more careful study and testing to ensure that the entire system functions successfully. New additions to a system always have the potential to cause more problems than they solve. Now even with the evolution of Fifth-generation languages (5GL) the implementation processes have speed up and got more efficient.

“While fourth-generation programming languages are designed to build specific programs, fifth-generations languages are designed to make the computer solve the problem for you. This way, the programmer only needs to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or algorithm to solve them.” (Godse & Godse, 2007)

Test automation has brought lots of solutions to how to improve testing process. At the same moment it has also brought its own challenges of how to use it efficient and at the required level of quality. Proper use of test automation in case like extensive low-level interface regression testing, could save companies lots of time and money compared to a situation where it is done manually. For some methodologies like Scrum, Lean or DevOps test automation has become critical part of process. To achieve quality enough continuous delivery and continuous testing, automation is highly recommended.

Based on studies, almost 40 to 50 percent of total cost of software development comes from reworking defective requirements, code and design (Jones & Capers, 1986.). In 1988, B.W. Boehm and P.N. Papaccio wrote the book *Understanding and Controlling Software Costs*. In that book they presented the theory about time spent on detect prevention will decrease the time spent for repair. Theory was that every hour used on proactive work will decrease the reactive repair time from three to ten hours. If the software

is already in operation, a simple software requirement error and fixing it could cost 50 to 200 times more, than what it would take to fix in the requirements stage (Boehm & Papaccio, 1988). When putting this to context, it is easier to understand. At first one simple one row of requirement could spread to pages of design documents, then into many hundreds of lines of code and into multiple pages of user documentation and finally tens of test cases. The original one row of requirement is much easier to investigate and correct than after all of the above have been added to it.

Software testing and software quality is widely researched subject even today, not to mention the oldest studies, starting from the late 70´s (Myers, 1979). Almost seems that everything related to that has been already studied and documented. Then, how could it be that still today same issues are still on the table? Based on studies the annual cost of software failures was at least 40 – 65 milliard euros in 2002. Which is more than the GDP of Luxembourg. (Tassey, 2002)

2.1 Background

When talking about software testing, companies are usually using some specific govenance model, which is based on some best practises used in IT field. In software development the process is mainly based on ITIL chance management framework. Example of simple software development is shown in Figure 1. This process only describes the five main stages of software development.

Figure 1. Example of software development process model.

However, there are also situations when testing does not follow any written or described testing process. This might be because of programming and testing are made by same person and step by step testing process is felt to be too slow or stiff. One reason might be that the project timetable is too busy and usually testing is the phase which is saved in a hurry. There are also some other reasons which are going to be presented later in the study.

Compared to simple software development process as shown above, there are also more complex and specific process models. For example, Microsoft SureStep (Figure 2) is methodology model made by Microsoft and is widely used in Microsoft ERP product portfolio implementation projects. It is flexible framework and can be used in Traditional and Agile Project models as well.

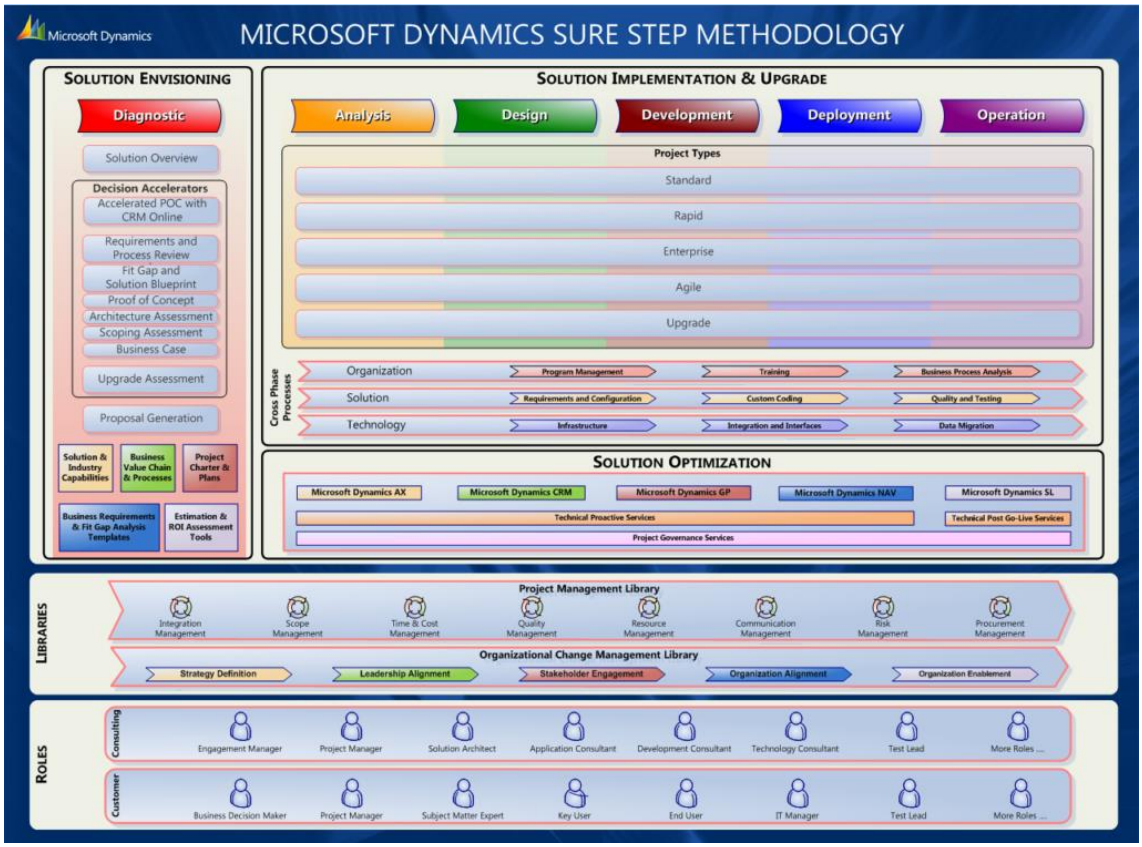


Figure 2. SureStep methodology made by Microsoft.

For more specific, like any other Software Process, SureStep tells about Who is Responsible for what and Who shall do What in Which order. SureStep defines process phases, milestones roles, artefacts, cross-phase-processes and additional processes for project management. SureStep supports a broad range of products: Dynamics AX, Dynamics NAV, Dynamics GP, Dynamics SL and Dynamics CRM. And different project types: Rapid Implementation, Full Implementation, Optimization and Upgrade. Products and project types can be combined.

2.2 Problem

Testing and software quality is one part of the overall quality and needs also take into deeper investigation. Main problem about the subject is that the specified testing process is sometimes lacking, or it is not strictly followed. This may cause problems during the software development process or after the project has “completed”. More than often lack of testing during the development project causes problem later. In these cases, costs of

fixing the problems later could raise multiple time higher than what proper testing during project would have cost. There are many studies that indicates different kind of cost estimates in which phase of project the effect is noticed.

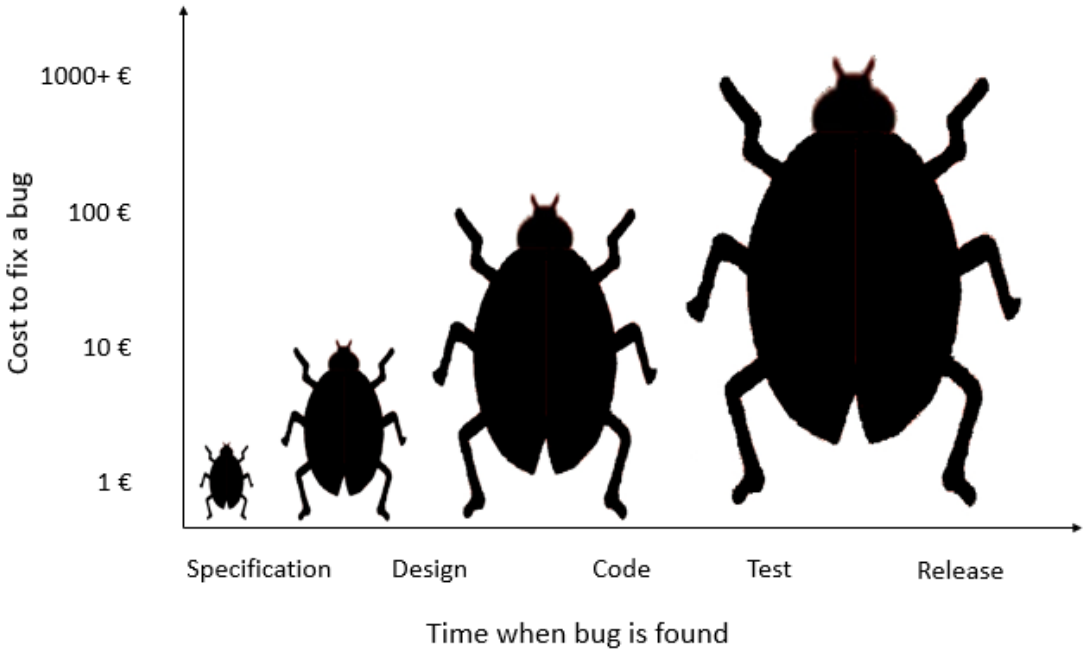


Figure 3. Cost to fixing bug from Ron Patton´s book *Software Testing* (2nd edition) (2005)

Ron Patton (2005) estimates that software cost-to-fix is increased 10 times per project phase. Oldest calculations seem to point out to year 1983. Back then Barry W. Boehm wrote the book *Software Engineering Economics*, where he pointed out the first linear cost estimates for fixing software defects (Figure 4) (Boehm, 1983).

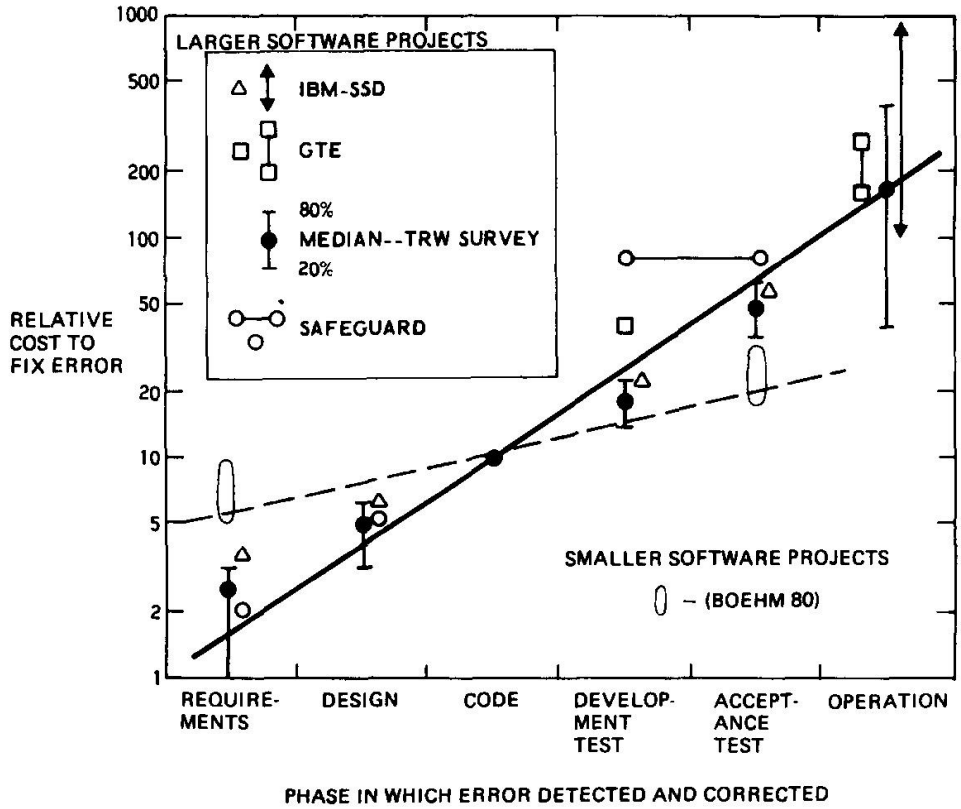


Figure 4. Cost estimate based on Barry W. Boehm study (1983).

Even today these analyses are true in most cases because it generally is harder to find an error once the project approaches it´s development cycle end. For example, bugs that are created in the design phase and not noticed or fixed during that stage of development most likely cost more to fix later because they can have wider impact and are most likely more complex to fix. Also changes made to fix the bug may affect the solution´s functionality. In that case it will also require more time to make all the changes, which will add the cost, effort and time. What is missing in from these charts above is at what phase the defects were introduced. First estimates based on what phase defect is introduced made Steven McConnell as we can see in chart below.

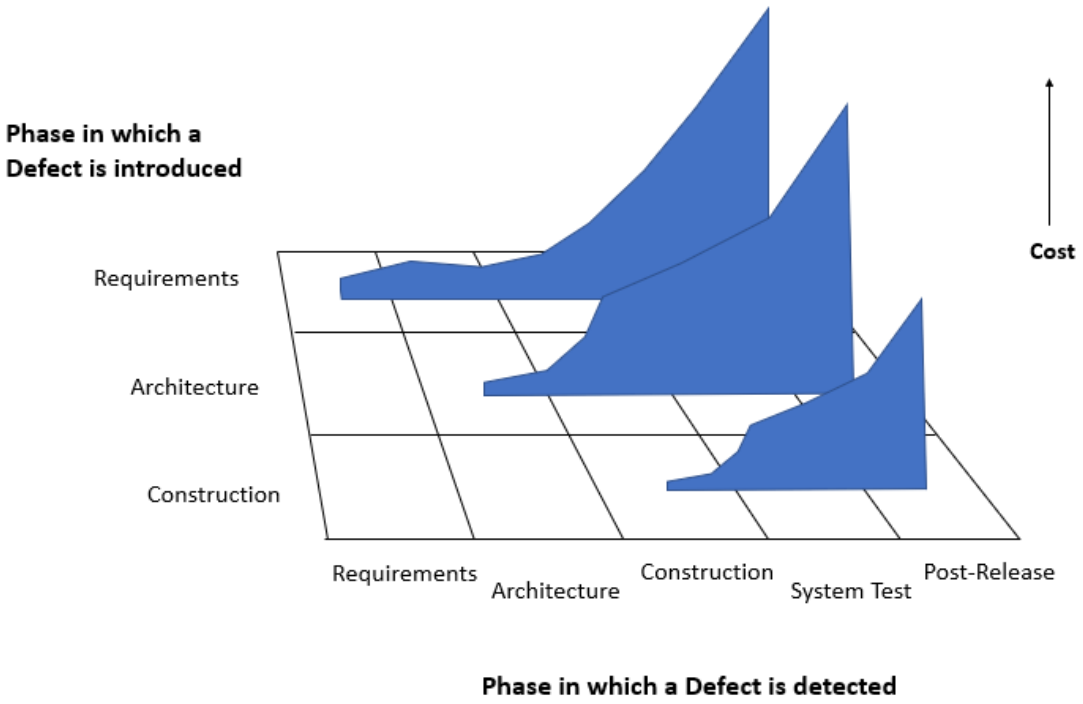


Figure 5. Cost of fixing defects from Steven McConnell's book *Code Complete*, Second Edition (2004)

" A table (Figure 5) in Steven McConnell's book Code Complete, Second Edition, shows the average cost of fixing defects, based on when they are introduced and detected. For example, a defect that is introduced in the architecture phase costs 10 times as much to fix if it is detected in the construction phase, 15 times as much if it is detected during the system test, and 25 to 100 times as much if it is detected post release."

2.3 Method and Scope

The main target of the study is to research software related cases through the history, which have failed for some reason. Idea is to focus and find out the root causes for the failure or catastrophe and is that somehow connected to testing. These root causes are then going to be presented and analysed. After analysing, solutions of "how the failure could have been prevented" are introduced into the selected cases. These failures are also examined from a cost perspective and how

much have the failures cost for the company or individual. For this thesis the studied timeline is set to be from the late of 50´s, when the software testing is said to have started and the end date is year 2017.

The secondary target is to study and gather data on how the software errors or big software failures have changed the testing policy or have there been any influence and is there any cost related context between failed cases.

The study follows quantitative research process. Source data is gathered and then presented into mathematical models, which are then further analysed and based on that some hypothesis are introduced. This process is shown in the above (Figure 6).

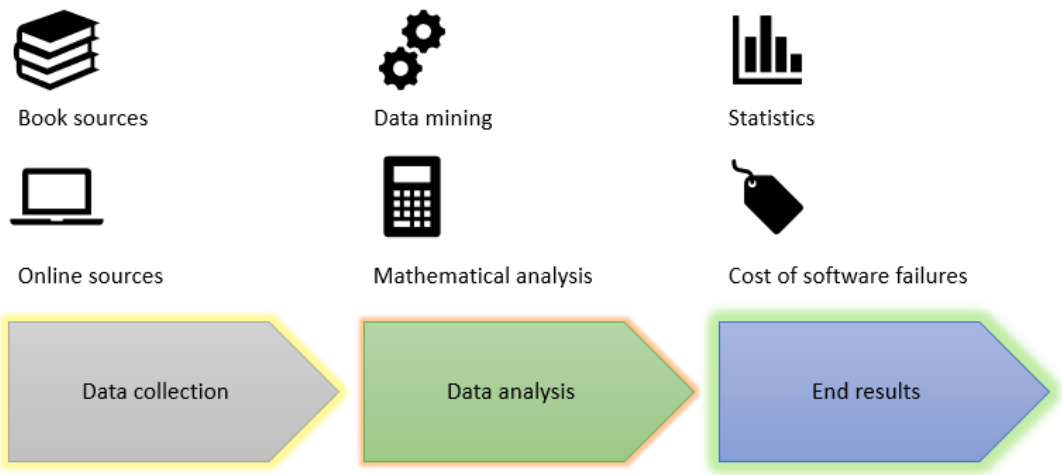


Figure 6. Research plan

3 Case Studies

Through the history there has been situations where software development or quality of testing has not reached the targets as well as planned. In this thesis, one or more historical cases are presented from different decades. These cases are then inspected from the perspective of what went wrong, what was the main issue, how it could have been prevented and how much that fail cost.

One purpose of the thesis is to research biggest or most appeared issues/bugs in software development, which are related to testing. These issues are then listed and inspected more specific. When the required data of testing related issues are gathered, idea is to study and compare if these issues still exist on today. If so, is there something that could be done to prevent these issues not to appear today or in the future.

3.1 Case Study 60´s – The Mariner 1 Spacecraft (NASA)

In 1962 NASA announced that Mariner 1 Spacecraft is going to be launched on July 22 to space with a plan to fly-by Venus and collect a variety of scientific data about Venus, like atmosphere, magnetic field, charged particle environment and mass. Mariner 1 was the first spacecraft in the American Mariner program. Until this day the program included ten Mariner probes.

“The vehicle was destroyed by the Range Safety Officer 293 seconds after launch at 09:26:16 UT when it veered off course. The booster had performed satisfactorily until an unscheduled yaw-lift (northeast) manoeuvre was detected by the range safety officer. Faulty application of the guidance commands made steering impossible and were directing the spacecraft towards a crash, possibly in the North Atlantic shipping lanes or in an inhabited area. The destruct command was sent 6 seconds before separation, after which the launch vehicle could not have been destroyed. The radio transponder continued to transmit signals for 64 seconds after the destruct command had been sent.
The failure was apparently caused by a combination of two factors. Improper operation of the Atlas airborne beacon equipment resulted in a loss of the rate signal from the vehicle for a prolonged period. The airborne beacon used for obtaining rate data was inoperative for four periods ranging from 1.5 to 61 seconds in duration. Additionally, the

Mariner 1 Post Flight Review Board determined that the omission of a hyphen in coded computer instructions in the data-editing program allowed transmission of incorrect guidance signals to the spacecraft. During the periods the airborne beacon was inoperative the omission of the hyphen in the data-editing program caused the computer to incorrectly accept the sweep frequency of the ground receiver as it sought the vehicle beacon signal and combined this data with the tracking data sent to the remaining guidance computation. This caused the computer to swing automatically into a series of unnecessary course corrections with erroneous steering commands which finally threw the spacecraft off course.” (NASA, 2017)

Root Causes

NASA has reported that the cost of software failure was 18.5 million dollars, which is around 135 million dollars when converted into today´s economy. It has said to be most expensive hyphen in the history. Later on, the error was corrected to missing superscript bar (overline) not hyphen.

” The bar was left out of the hand-written guidance equations. Then during launch the on-board Rate System hardware failed. That in itself should not have jeopardized the mission, as the Track System radar was working and could have handled the ascent. But because of the missing bar in the guidance equations, the computer was processing the track data incorrectly.” (Ceruzzi, 1989)

What Was Learned

How could it have been prevented? One speculated reason to error was strict timetable. In 1960´s there was hectic Space Race between Soviet Union and United States of America. Soviets had already made multiple attempts to fly by Venus in the early 1960´s. Even if the error is due to a hardware and software failure combined the software bug still managed to pass tests. Of course, during that time software testing was not something like today. Code was made by using punch cards. Typos or incorrect commands was not possible to caught by simply looking at what was typed on computer screen.

3.2 Case Study 70´s – Undetected Hole in The Ozone layer (NASA)

It was year 1975 when U.S Congress ordered NASA to develop a complex program to monitor and research a weather effect which occur in the upper atmosphere. This phenomenon was called ozone layer and Congress was eager to found out the health of it.

Three years later, in 1978 a Nimbus-7 satellite was designed to measure the Earth Radiation Budget (ERB), but for the extra the satellite also included two new sensors from NASA which were planned to measure the total amount of ozone over the entire globe, in a given area of atmosphere. These sensors were called the Solar Backscatter Ultraviolet (SBUV) sensor and the Total Ozone Mapping Spectrometer (TOMS). Sensors which are sensitive to radiant energy, especially in the ultraviolet area of the spectrum uses the technic that molecules and aerosol particles reflect specific wavelengths of UV rays while on the other hand ozone absorbs UV rays at different levels in the atmosphere. The amount of reflected ultraviolet energy up to the spacecraft, researchers could analyse and produce estimate profiles about how thin or thick the ozone was at different locations and altitudes.

It took more than eight years until in 1985 British scientist team found a substantial reduction in ozone layer over Antarctica. Team was using a ground-based ozone spectrophotometer called Dobson and noticed that the amount of ozone in stratospheric layer over Antarctica was actually 40 percent less than it had been in the previous year. This finding was a total surprise to the science community. Scientist were expecting that the anthropogenic ozone exhaustion would first happen in the upper levels of stratosphere, around 30 to 50 km high. Due that they expected that the signal from the sensors over a total column of ozone to be weaker. After the finding, NASA researchers quickly checked the TOMS data and shockingly notices that it had also detected an extensive loss of ozone all over of Antarctica. How come, that they had not noticed this discovery earlier? (NASA, 2001)

Root Causes

Data analysis software for TOMS data had been programmed to notice (flag) and put aside data points that diverged widely from the expected measurement data. This meant

that initial measurements should had set off the alarm, but they were just simply forgotten. So, the team failed to notice the issue in time because it was way more serious than experts and scientists have expected.

Scientist´s assumptions led to a situation where ozone layer hole was discovered only after seven years. If the software would had been programmed in the way that all the deviated measurement data would gave the alarm, problem would had been noticed earlier and it could had been reacted sooner.

What Was Learned

After discovering the ozone layer reduction, both NASA and ESA started to monitor ozone levels. These levels continued to decrease over Antarctica for upcoming years, so action was required. Finally, it only took 2 years to react these alarming measurement results. In 1987, 43 nations signed the “Montreal protocol” in which they agreed to reduce the CFC´s, which is up to 200 times more efficient than carbon dioxide, by 50 percent by the year 2000. One simple human mistake and years of environmental or atmospheric problems were left unnoticed by everyone.

Lesson to be learned in this case is, never make assumption how something should go. By limiting the alarm level to single value and not the values bigger than that, like in this case led the system to work incorrectly.

3.3 Case Study 80´s – Therac-25

The Therac-25 was a radiation therapy machine produced by Atomic Energy of Canada Limited (AECL) in 1982. It was used in treatment of cancer. Therac-25 was the third-generation version of radiation therapy machines and the first two versions (Therac-6 and Therac-20) were partnership project with Compagnie General Radiographique (CGR) of France.

More and more are computers displayed to be a part of safety-critical systems and due to that, had been involved into accidents. Maybe most of the well-known software and computer related accident happened to a computerized radiation therapy machine,

which is called the Therac-25. Accidents happened between 1985 and 1987, when six people suffered from massive overdoses of radiation using Therac-25 machine. These overdoses led to a deaths and serious injuries. These accidents have been portrayed to be the worst radiation disasters in the medical history. (J.A. Rawlinson, 1987)

What was causing the accidents was the reason that instead of low power beam, the high-power electron beam activated without the protective spreader plates to rotating into their right places. The previous model was relying on hardware interlocks to prevent these kinds of situations, but new Therac-25 was using software-based safety interlocks. Sadly, in this situation software interlock was failing due to the counter issue. One-byte counter in the software testing rote was often overflowing. If the person how operated the machine put the instructions at the precise moment when the counter overflowed the interlock would fail.

When the high-powered electron beam activated it hit the patients almost 100 times amount of radiation than it should had and causing the patients possibly lethal amount of beta radiation. A few days later first symptoms of radiation poisoning, the burns appeared. For three patients the overdose was too strong they died because of the poisoning. (Leveson & Turner, 1993)

What can be learned from the Therac-25 case, is that safe system cannot be made by concentrating on specific software bugs. Basically, every complex software system is possible to code in the way that it might behave unexpected in under the wrong conditions. In this case some basic mistakes led to a bad software development practises and to a machine that was built to use software only for safe operations. However, one particular software error does not make difference if the whole system design is inadequate. (Bowen & Hinchey, 1999)

Root Causes

Investigations pin pointed four main causes or faults that made the accidents possible. These were:

1. Management was inadequate and there were no clear procedures about how to follow through the reported incidents

2. Arrogance behaviour toward software and removal of safety hardware mechanism
3. Presumably not even above the mark software development practises
4. Unrealistic estimation of risks and lowering attitude toward the results obtained

To open the points one by one. Firstly, reported incidents was given to the user by notice “MALFUNCTION” followed by number 1-64. In user manual there were no clear guide how to act in any given incident. Also, user had the opportunity to reject or override the incident by pressing letter P to continue the operation despite the notification.

Secondly, overconfidence in software led to a situation where safety interlocks were removed from the hardware and then software was a single point of failure. This type of design is risky even if the hardware is the single point of failure, not to mention the software. Software was also reused from the previous version where the hardware interlocks still exists. Further investigations also showed that the “bug” that allowed the radiation levels to rise was also found in the previous version Therac-20 where the hardware interlocks were preventing this to happen. The first safety analysis on the Therac-25 did not include software, even nearly full responsibility for safety rested on it.

Thirdly, software engineering practises were pointed to be incomplete or inadequate. Software was not fully tested with the hardware during development but when the equipment was already installed to hospitals. Also, user interface testing was inadequate. At least one of the accident was reason due to race condition which caused by user interface timing and software misinterpretation.

Lastly, after the first accident happened and was reported back to the AECL, the manufacturer company did not even start to investigate the reposted issue. Based on their risk assessment reports the safety class has improved by five levels because of the new microswitch repair and because of that the failure of this switch could had not been caused the radiation burns to the victim. It did not help, that the original hazard analyses had passed the software totally. Due that they had no insight about the code and they started to assume that every software error was equally likely. Gained probabilistic risk assessment reports created unreasonable confidence about the machine and its functionality. Because the software was not included in the beginning, they could not handle the feedback rationally.

What Was Learned

This case indeed was the watershed moment, that revealed the consequences of how things can go seriously wrong if the life-critical system is using the code which is not properly tested and designed. After accidents FDA declared the machine to be “defective”. AECL issued software patches and hardware updates which eventually allowed the machine to return to service. Lessons to be learned:

- In life-critical systems, do not use single point of failure solution
- Critical software errors should not be able to override just by pushing any key

3.4 Case Study 90´s – Ariane 5, Flight 501 (ESA)

On Tuesday 4th of June in 1996, the Ariane 5 launcher was waiting to start it´s maiden flight. All was set, and the launch started. Only after 40 second has past from the initiation, everything went wrong. Launcher has reached about an altitude of 3700 m when it started to veer off from it´s flight path and finally broke up and exploded. (European Space Agency, 1996)

Launch took, place from the Kourou in French Guiana, where the Europe´s Spaceport, Guiana Space Centre is. Purpose of the launch was to carry four European Space Agency´s (ESA) cluster satellites. These satellites were designed to research the Earth´s magnetosphere and how it interacts with the solar winds. All were identical, and they were supposed to fly in the formation which allow them to take measurements at the same time to provide the best detailed three-dimensional research of Earth and Sun interactions and what happens in the close to Earth space.

Launcher´s self-destruction activated automatically due to a huge aerodynamic force, which were tearing the rocket boosters off. The rupture of spacecraft had started a moment before, when it started to turn away from the original course. This was caused the pressure of three powerful jets in the boosters and in main engine. The spacecraft was doing a sudden course alteration, something that was not required. The rocket thought it had to compensate because of the wrong turn that had not happened. The on-board computer, which was controlling the steering, got signals from the inertial navigation system (INS) to change the course. The INS use accelerometers and gyroscopes to monitor

motion. The data that was coming from the INS looked almost like real data, but the data was actually error message coming from the diagnostic system. In fact, the whole navigation system had gone to offline. (Gleick, 1996)

Software error caused the explosion and ESA has reported that the total costs of this failure was up to 370 million dollars. Some estimates are even higher and take account the whole Ariane project which took 7 years and approximately 7 billion dollars.

Root Causes

After the investigations the whole chain of events was clear, starting with the reason of what caused the destruction of the rocket and what was the primary cause that started the disaster. Main problems were the flight control and guidance system. More detailed technical explanations are given in the report, which concludes:

" The failure of Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system. The extensive reviews and tests carried out during the Ariane 5 development programme did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure." (European Space Agency, 1996)

So, what went wrong? Immediate reason for the problem was software error. What made this possible was design failures in the system and project management problems. Software error took place when the system tried to convert a 64-bit floating number of data to a specified 16-bit integer, which made the number of data to overflow. When the navigation system went offline, the identical unnecessary backup unit took the control. This unit was designed to jump in the situation like this, but the backup unit had also failed in the identical way just a couple of milliseconds before it was actually running the same software.

This was caused by greater horizontal acceleration due to more powerful solid boosters. Compared to Ariane 4 this was five times more rapid launch. Ariane 5´s inertial reference

system was the same as in earlier version. Because there were no reported issued with Ariane 4 they did not expected to have any issues with Ariane 5.

What Was Learned

Different roles and groups were interviewed and here are the interpretations from their point of view.

Programmers:
The whole disaster is obviously the result of a program defect. Wrongly managed software exceptions caused by a data conversion (64-bit -> 16-bit). Better programming practice would have prevented this failure from occurring. (Bashar, 1997)

Designers:
It was obviously result of design error. The design specification for the system accounted only for the random hardware failures and due that the exception handling mechanism was inefficient to revive from a random software bug. As a result, a properly functioning processor in the INS was terminated and after that the backup processor failed similarly. Simply a better design like a system that mediate software exceptions from stopping hardware devices which were working properly and would have stopped the issue.
(Bashar, 1997)

Requirements engineers:
It was obviously result of faulty analysis of changing requirements. The requirements for the new spacecraft Ariane 5 were different what was in the earlier models. Still, the single section of alignment code that caused the failure of launcher was not even actually needed after lift-off, like it had been on older versions of Ariane. The function stayed operational in new model Ariane 5, even it did not fill any actual requirement. This failure could have been prevented if the traceability and requirement analysis would have been better. This point of view was supported by software maintenance studies. Based on requirements engineers the whole disaster was possible due to the “if it ain´t broke, don´t fix it” approach by maintenance team.
(Bashar, 1997)

Test Engineers:

The whole catastrophe is clearly the outcome of insufficient verification and validation, review and testing. Like the original disaster report notes that there was no actual test that would have checked if the INS would act correctly in real like situation. Necessary tests that could have been performed during acceptance testing by suppliers, would have possible revealed the failure. (Bashar, 1997)

Project managers:

The whole catastrophe is clearly the outcome of inefficient project management and project development processes. For instance, the whole development review process for spacecraft was incompetent. When the specifications, rationale documents and code was inspected there were no any external persons included in the project. Also, the code and the code related documentation were often illogical. To prevent or expose the issue, processes for the project management and cooperation with engineers should have been better and more closely facilitated. (Bashar, 1997)

.

All these comments above are right interpretations. One software error could have been noticed if the overall project management or process would have been better. What have learned from this case could be summarized to these points

1. Only run software project in safety critical systems when it is really required
2. Always remember to test both what the system should do and what it should not do
3. Never plan a system that uses system shut-down as a default exception processing response when it does not even have fail-safe state
4. When doing critical calculations, always try to restore best values even in the situation where definitely right values are not possible to compute
5. When doing testing, always try to use real equipment instead of simulations
6. To develop the software review process, incorporate outside participants and verify all suppositions created in the code

3.5 Case Study 2000´s – National Cancer Institute, Panama City

When talking about software failures that causes human casualties, one can speak of a catastrophic mistake. One of that kind is a case of National Cancer Institute that happened in 2000. U.S based company, Multidata Systems International, created the cancer treatment software that unfortunately miscalculated the right amount of radiation for the patients that was going under treatment.

The software was designed so that the radiation therapist could draw “blocks” on the computer screen. These blocks were metal shields that protects healthy tissue for not getting the radiation. Total of 28 patients that were getting treatment for prostate, colon and cervical cancer, were overexposed to radiation at the Panama National Institute of Oncology. Number of overexposures varied from 20 to 100 percent from what was the given dose. From those 28 patients, 9 of them have died based on reports. In five cases the direct cause of dead was radiation overexposure. Based on those reports from 2017 the researchers estimated that many of the patients that suffered from radiation overexposures will most likely suffer from radiation related complications. (FDA report, 2017)

Root Causes

Software allowed the technicians to use four shielding blocks for healthy tissue, but the doctors wanted to use five instead. After a while the doctors noticed that there is loophole in the software. They were able to draw a one single huge block and left a hole in the middle. This one block was then size of five blocks. What the doctors did not knew was that the software calculated different treatment dosage depending how they draw the hole. When the hole was drawn correctly the dose was also correct but when it was drawn wrongly the calculated dosage could be more than twice.

When examining the case reports, following factors were considered to be the reason of overexposures:

- Panama National Institute of Ontology did not had the proper verification for treatment plan
- The process of how the beam block data was entered into the software
- The way the Multidata software was understanding the data that was handling the beam block

What Was Learned

Most important lesson was that do not use the solution in a way it is not meant to use. In this case users should had contacted the supplier for needed changes or what would happen if they use it against standard process. Software supplier made the assumption that the solution is used correctly and with the quality assurance procedures, so they did not tested cases were procedures was not followed. The physicians, who were legally required to double-check the computer's calculations by hand, are indicted for murder.

3.6 Case Study 2010´s – Stock Market Mistake (KCG)

An American company, called Knight Capital Group, which operates in global financial market. More specific electronic execution and institutional sales and trading. In the year of 2012 KCG was the biggest trader in US equities, market share of around 17%. It´s trading division Knight´s Electronic Trading Group, handled daily more than 3.3 billion trades, which converted to money equal over 21 billion dollars.

The New York Stock Exchange (NYSE) was planning to take into action a new Retail Liquidity Program. This program was meant to offer improved pricing to retail investors with the aid of retail brokers and the launch date was set to August 1, 2012. To prepare for this operation, KCG updated their high-speed, automated algorithmic router. This router was designed to send orders for execution into the market. This trading algorithm was called SMARS and one of it core function was to receive orders from KCG´s other systems which uses the same trading platform. It was coded to function in the way, that when it receives big orders from the trading platform, it simply divided them into smaller orders. Idea was to find buyer or seller that would match for the number of same shares. So basically, the bigger the "parent" order, the more "child" orders it would have created.

The reason for the SMARS update came from need to replace the old system. This old and unused code was called "Power Peg" and in this point it has not been used in 8-years. When the new code was updated it has a function that should recreate an old and unused "flag" that was formerly used to activate functionality of Power Peg. The new code had completed the tests and seemed to work reliably and correct.

Root Causes

This new software was manually deployed between July 27 and July 31 in 2012. One by one the servers received the new update, in the end eight servers had been updated. This is what the SEC report indicated about the deployment process.

"During the deployment of the new code, however, one of Knight's technicians did not copy the new code to one of the eight SMARS computer servers. Knight did not have a second technician review this deployment and no one at Knight realized that the Power Peg code had not been removed from the eighth server, nor the new RLP code added. Knight had no written procedures that required such a review." (SEC Filing, 2013).

On Wednesday 1st of August in 2012, the markets opened in the morning. This day started like an average Wednesday. KCG´s trading platform started to process orders from broker-dealers and handle them for the new Retail Liquidity Program. Even in this point everything went smoothly. The problems started when the seven correctly working servers had operated their orders and it was eighth server turn. When the orders arrived at this server the new code activated the old "flag" even it was meant to overrate the old purpose, but it went the other way around and it woke up the old Power Peg to alive. To understand why this was so catastrophic, the purpose of the old code needs to take into deeper review. Old code counted the shares which was sold and bought against a parent orders every time child orders were completed. So, this means that Power Peg kept a record of child orders and stop them as soon as the parent order was ready. When the eighth server "flag" was activated the operation started to route child orders but was not keeping track about the number of shares of parent orders, like a never-ending loop.

In the morning people started to notice that something was wrong. In the Wall Street people were holding their breath for a couple of minutes and started to quickly wonder how this could be possible and why it was still continuing. In the fast trading world these minutes welt like hours. Within the disaster KCG´s trading executions created more than half of the total daily trading amount. Due to that, some stocks rise over their value and due to that other stocks dropped in their value to scale the error.

Whole disaster took 45 minutes. During that time KCG tried to stop these trades several times. Sadly, there were no kill-switch or any documented info about how to react in this

kind of situation. This led to a situation where they had to diagnose and solve the issue in live production environment and at the same time almost 10 million shares were traded in every minute. After a while they noticed that they can´t solve the issue what was affecting the false orders. What happened next only made the situation worse, they started to uninstall the new code from the servers that was actually deployed correctly. So, in the panic they erased the working code and left the damaged code into servers. Now instead of one server all the eight servers were activating the old Power Peg function. Finally, after 45-minutes of trading they managed to stop the system. Harm had already done, during the time servers were online they processed 212 parent orders and these orders SMARS converted to millions of child orders. As a result, these child orders converted into 4 million transactions which equals almost 400 million shares. When all was over, company suffered $460 million losses and almost had to close the doors for good.

What Was Learned

What happened in August 1, 2012 is a book example for every development team about how not to operate. Even the software has been writing and tested well it is not enough if it is not delivered properly to market. Not only one person could be blamed for this case. Company had not thought about the deployment process in the level required for the risk there were. Basically, process was only built counting on human actions and without any safety backups. Mistake could happen so easily in this kind of process, it could be just misunderstanding of instructions which cause wrong execution. To prevent these kinds of disasters, couple of points should take into consideration.

- An automated deployment system
- Better configuration and test automation

When deploying software, it should be a reliable and repeatable process. Also, consider that automate as much as is moderate and as much as the situation allows.

3.7 Case Study 2017 – VR Ticketing System

Finnish state railways (VR) started the new ticketing system project in 2008. It was the largest IT project in VR´s history. The reform of the million euros Sales system was finally

almost ready. The ticket machines and the conductor handheld machines have been replaced and the old network service has been updated to become more modern.

Actually, the reform was supposed to be ready in the winter, but the timetable had doubled twice because the software and hardware had been unfinished. The publication was first transferred from March to June and then to September 14. Train traveling on the new ticketing system was a side thing, the bigger news was the train ticket pricing reform. The train ticket would get a discount if it would buy it in advance. The crowd rush was expected, so the VR had played it safe. The experts had designed a system that could withstand even the toughest visitor peaks. Capacity of computer system was raised even over the expert recommendation.

People were swarming in the Railway station project rooms all the night at 14th of September, when 40 experts were monitoring and guiding the deployment. Old systems were shut down and traffic was moved to new system. First ticketing machine was opened at 5 o´clock in sales office in Turku. Seven in the morning statistic started to raise when people started to wake up. One hour later the traffic was already lively. Between eight and nine the number of visitors broke all previous records, including the security clearance that the system was designed. Soon, some of the customers could no longer access the service, and even those who got there had to wait for the download of the pages at worst tens of seconds. The situation got worse when the noon approached. Next to the headquarters of VR in the main ticket sales office of railway station, customers started to get nervous. Equipment for customer service became increasingly slower, and the new cheerful green ticket sales machines started to get jammed all over Finland. (TiVi, 2011)

Root Causes

All train tickets are sold through the same booking system, which is why the customer service provided by the online service also hindered the purchase of paper tickets. Eventually the tickets were only sold by the conductors. However, only the number of visitors did not explain all the problems, there was something else wrong in the system.

The new service was designed to withstand the greatest ever-experienced number of visitors, the 420 000-download record during the umbilical crack. At the first day, downloads were 1.2 million. In the computer world queue is not known, but all the e-commerce visitors are pushing for a virtual ticket booth at the same time. All customers on the web service will receive poor service if their devices run out of service. Slow issues began to be solved in many different directions, as the reservation system was built in co-operation with subcontractors. Accenture had been responsible for building the system, but its subcontractor Enfo had provided ticker machines and conductor equipment. Tieto was responsible for the data centers and servers running the system. In adding, seat reservation system legacy from the old ticket sales software, which was originally made by Logica.

VR asked Tieto to triple the server capacity at the first day of project. This however did not solve the issues. It was revealed that the servers worked unstable because their software had a known, but unresolved bug. The problem was not revealed in the load tests. Tickets are sold in machines, counters, online and on the train, and no load from all direction could have been simulated in advance, realistic enough. It was also revealed that the ticket machines had a problem that unnecessarily burdened the other booking system. Ticket machines was using Oracle´s Weblogic product and more specific software version 10. This version had known bug which affect memory handling, especially when there were high usage numbers. Finally, this led to a situation where all the automatic ticket machines were decided to close for repair. This took near two weeks.

After first days VR noticed that number of visitors is going to stay to high levels and will not go back to what it was in the past. From the biggest rush the numbers stayed over 500 000 download which still was more than double from the previous one.

A feature that restricts the concurrent number of visitors was switched on in the online store. There was a one problem with this limiter. Some customers no longer got a ticket, even though they had already paid the ticket and money had disappeared from the account. An improperly operated limiter could kick the customer out of the service before they could get their ticket. Even this problem had not been found in the tests. The feature was not originally intended to never be used. (TiVi, 2011)

What Was Learned

Stress tests were planned based on previous user numbers. Even though it is hard to predict what would be the real situation, there are no harm to run tests using oversized values. In this case this would had revealed the bug that was on servers and hopefully it would had been repaired. Second issue was known bug in ticket machine software. This is something that supplier should had took care of during the project. Issue was repaired easily when the Oracle gave the fix to Enfo, but it was costly to fix in the situation where all systems were down, and customer get financial losses every minute.

This case could be summarized to these two guides.

- Always try to prepare for unknown. When planning the test cases use also values that could sound exaggerated
- Take care of the known issues. Some bugs could be harmless but at least these should be checked before it is safe to leave them as they are

4 History of Software Testing

This section explains the background about Testing and testing related methods and processes. Software testing as a term is almost as old as the actual first rows of code. Back in 1950´s John W. Backus created the first programming language FORTRAN and it is staged that software testing also started on that time.

Back in 70´s G. Myers wrote that, Testing is a process of executing a program with intent of finding an error (Myers, 1979). That book was called “Art of Software Testing”.

In different publications, the definition of testing varies according to the purpose, process, and level of testing described. Miller gives a good description of testing in “Introduction to Software Testing Technology,”

> “The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances.” (Miller, 1981)

Actually, Miller describes the testing as an activity of software quality insurance. Miller claims that the most important purpose of testing should be finding errors. He defines that when the test process has a high chance to find still not discovered error it is a good test and when it finds still not discovered error it is a successful test. Of course, this generic class of software testing operations is possible to further divide.

The concept of testing itself develop with time. What happened was that the evolution of testing definition and targets of testing led to situation were testing techniques started to evolve as well. For next the concept evolution of testing is taken into examination and for that let´s use the testing process model that was proposed by Gelperin and Hetzel in 1988, in the book called “The Growth of Software Testing”. (Gelperin & Hetzel, 1988)

4.1 The Debugging-Oriented Period

The era that is called The Debugging-Oriented Period actually meant that testing was not separated from debugging. In timeline this meant the time before year 1956.

It was year 1950 when Alan Turing published the well-known article called “Computing Machinery and Intelligence”. That article is recognized to be the very first what comes to program testing and in the article, Turing present the question about “How would we know that a program exhibits intelligence?”. Noted in other way, if the starting point is to build this kind of program, then how we would know if the program has fulfilled its requirements? In the article Turing identifies operational testing scenario. In that scenario software program and the reference system, in this case human should behave exactly same way so that the tester itself cannot separate one from another. This method is studied to be the original form of testing method known today as a functional testing. In that time the terms program testing, checkout and debugging were not separated.

4.2 The Demonstration-Oriented Period

The era that is called The Demonstration-Oriented Period actually meant that testing was making to verify that the software fulfils its specification. In timeline this meant the time between 1957 and 1978.

Until 1957 testing, which was then called the program checkout at that time, was separated from debugging. In the year 1957, Charles L. Baker commented that the program checkout had two goals, to make sure that the program works and to make sure that the program resolves the problem. (Baker, 1957)
The second goal was considered as the centre of testing, as “make sure” has often been transposed into the test target for meeting the requirements. The difference between testing and debugging was based on determining success. During this time, the definitions emphasize that the purpose of the test is to prove the correctness. Based on that the ideal test is therefore only possible when the program does not contain errors. In the 1970s, there was also a broad idea that software could be tested exhaustively. Due that idea, many research was established that were studying the coverage of testing. This was also mentioned in the Goodenough and Gerhart´s study, that exhaustive testing is identified either in program paths or in the program input domain. (Goodenough & Gerhart, 1975).

4.3 The Destruction-Oriented Period

The era that is called The Destruction-Oriented Period actually meant that testing was planned to detect implementation errors. In timeline this meant the time between 1979 and 1982.

This era was crucial, what comes to software testing. Testing process was actually executed for the purpose of finding the errors. Test cases were actually more valuable if the errors were found. This is the opposite way of thinking than what was in the demonstration-oriented period. Instead of picking the test cases that most likely do not find any errors the testers chose cases which has the higher probability to find the errors. This step led to early connection of other validation/verification activities with testing.

4.4 The Evaluation-Oriented Period

The era that is called The Evaluation-Oriented period meant that the purpose of testing was to detect faults in requirements, design and implementation. In timeline this meant the time between 1983 and 1987.

It was year 1983, when the Institute for Computer Sciences and Technology released the Guideline for Lifecycle Validation, Verification, and Testing (VV&T). In this guideline the methodology that combines analysis, review and test operations to offer product evaluation within the software life cycle. For each stages of life cycle there are stage related functions and products. These evaluation techniques are divided into three categories; basic, comprehensive and critical. These sets are cumulative in the way that each one is included in its follower.

4.5 The Prevention-Oriented Period

The era that is called The Prevention-Oriented period meant that the purpose of testing is to prevent faults in requirements, design, and implementation. In timeline this meant the time since 1988.

In year 1983, Boris Beizer wrote the book called *Software Testing Techniques,* which includes the widest list of testing techniques. In that book Beizer noted that planning the test cases is actually maybe the most efficient bug inhibitor operation what is known. This statement extended five years later to actual definition of error prevention testing. (Beizer, 1983)

This definition led to a whole different way of thinking about how the testing should be done and how important the early testing is. In 1990, Beizer continued to develop testing process and introduced the four stages of thinking about testing. These stages were; 1) make software work, 2) break the software, 3) reduce the risk and 4) state of mind, in other words a worry about testing throughout the whole life cycle. One year later, Hetzel introduced own definition about testing. The definition stated that “Testing is planning, designing, building, maintaining and executing tests and test environments.” (Hetzel, 1991)

Both of these two, were huge innovators about how the testing is done more better and at what phase of the project testing should be done and with what intensity. Even this era is also focused to software requirements and design so that the implementation errors are avoided, it still differs from the previous era by the mechanism. This means that the prevention framework emphasizes the test planning, -analysis and -design operations, while the evaluation framework mostly counts on analysis and reviewing methods other than testing.

How the Static testing differs from Dynamic testing? When talking about static testing, the program is not performed as such. The real program code is examined manually or with some software testing tools. In evaluation techniques it is considered to be verification technique. Term “Code 10” testing is also referred to static testing. Code 10 comes from the code 10 error, which is the error that causes device to crash and the device won’t start. This means that during the testing different walkthroughs, document reviews, inspections and feasibility analyses are performed. Developers usually runs static testing first, and only after that any other type of testing, to discover any code breaks, undeclared variables, syntactical errors and so on.

How about dynamic testing? Dynamic testing performs the program code with actual selected test cases. This type of testing interplays with the actual system by submitting an input value, then collecting the output value and finally crosschecking this value with the expected result. This way it can find out the errors by interacting straight with the

system. This also allows to allocate a different team to perform the testing, because they don´t need to know about how the implementation part of the system is working. (Kaner, Falk & Hguyen, 1999)

Next box testing methods and techniques are introduced. Techniques are listed in (Table 1.) but these techniques have not been taken into deeper investigation. Only different box approaches are explained.

Table 1. Box testing techniques

Black box testing	White box testing	Grey Box testing
Functional & System testing	Unit testing	Integration testing
Stress testing	Error handling testing	
Performance testing	Desk checking	Regression testing
Usability testing	Code walkthrough	
Acceptance testing	Code reviews and inspection	
Beta testing	Code coverage testing	
Ad hoc testing	Statement / Path / Integration testing	
Regression testing	Function testing	
Intersystem testing	Complexity testing	
Parallel testing	Mutation testing	
Boundary value		

The box approach:

- White box. It is also called as open box testing, clear box testing, transparent box testing, structural testing and glass box testing. To perform this type of testing the tester should have the full knowledge of how the system implementation works. White box method tests different data structures, system states, loops, paths and decision points. Compared to black box testing, the white box testing can reveal the defects fast and the coverage of test is also considered fully complete. Also, if compared to black box testing, this method requires huge testing knowledge with a knowledge of different testing tools, like source code analyzers and different debuggers. (Sommerville, 2001)

- Black box. For this method the name is an omen. In black box method the tester does not have knowledge about the system architecture, internal structure of the system or how does it work. Usually even the source code is not available. So, this means that the tester only knows about how the software is intended to work and is unaware of how the software do the activity. Because the tester only knows the system functionality and is unaware about the system implementation, this situation is also called functional testing. The lack of info of the internal program functionalities usually led to a longer testing times. (Sommerville, 2001)

- Gray box. This method is combination of white box and black box testing. Testers usually have at least some sort of knowing of the internal functions of the system. This required information has been usually received from architecture diagrams and from detailed design documents. Test performance is executed at black box level. Compared to other two the source code is also partly accessible. For an example of this kind of testing is database table check by querying after performing some test. (Kaner, Bach & Pettichord, 2001)

Although focusing on the prevention or removal of errors in the early stage of development cycle can be considered to be a critical activity, however it is still obviously not enough. Developers and testers have plenty ways to deliver feedback for the application when the coding have begun.

4.6 Period From 2000 to Today

The evolution of software testing in this millennium has changed more what comes to different ways of doing and planning of testing. Maybe the biggest change happened in year 2001, when 17 representatives from different development methodologies were gathered in conference held in Utah and signed the Agile manifesto. Since then the twelve principles of Agile software methodology have spread to worldwide and have made big influence on software development and testing. In 2002, one of the well-known certification for testing came in the picture. The International Testing Qualifications Board also known as ISTQB was founded in Edinburgh in November 2002. To this day ISTQB has administered over 740 000 exams and issued more than 530 000 certifications in over 120 countries world-wide. Also, in 2002 the first version of popular bug tracking tool JIRA was released by the Australian software company Atlassian Software. JIRA is

widely used also for its issue tracking and project management properties as well as bug tracking. Today JIRA is the most popular issue management tool by over 75 000 customers.

Later in the year 2004 automation testing technology field was introduced, when popular web application tool Selenium was released. A year after that, in 2005, SoapUI is released to SourceForge. SourceForge is a web-based ware house for source codes. Its purpose is to function as a centralized hub for open source and free software projects. SourceForge was the first product that provide this service to free and open source projects. It might be best known for offering revision control systems, like SVN, Bazaar, CVS, Mercurial and Git. It also provides centralized storage and project management tools for project developers. When the SoapUI was released it became a popular tool for the testing of web services. These are only a few milestones in the field of testing. New and more innovative solutions are released one after another.

Software testing techniques and methods have change a lot in the last decade. Just by moving from Waterfall method to Agile, not to mention adaptation of different testing tools. Software testing has also changed in the business wise. Outsourcing of testing in the market was a non-existent in 1998. This market grew to 8.5 milliard euros business by 2010. It is also reported that testing industry in India has grown from zero to 3 milliard euros at short notice and employing nearly 65 000 people. This trend seems to be even growing in the future.

One trend that seems to be growing is increased adoption of DevOps. DevOps offers programmers way to write software codes in small portions. These “portions” are then combined, assessed, monitored and implemented in just a couple of hours. Compared to traditional way of writing larger portions of code over weeks or months and then spending further weeks for testing.

DevOps helps programmers to reduce design time without losing quality. Then testers or QA will gain benefits using hybrid testing model. Hybrid model including both manual and automating testing. It basically takes advantage of both forms of testing approaches. It helps to overcome challenges of testing complex test data in a variety of test cases in a shorter period of time, without overlooking details.

5 How to Improve Testing

As mentioned earlier in Chapter 2, studies have shown that testing usually consumes 40-50 % from total development effort. This is known fact but how often this information is practiced or utilized? Tools for estimating resource load or project phases has been available since 1981. In the late 70´s Barry W. Boehm developed the constructive cost model named COCOMO, which is maybe the best known and widely used model. This model, also referred as COCOMO81, was published in Boehm´s 1981 book called “Software Engineering Economics” as a model for estimating effort, cost and schedule for software projects. This model was based on 63 projects at company named TRW Aerospace, where Boehm was Director of Software Research and Technology. The study examined projects ranging in size from 2 000 to 100 000 lines of code (LOC), and programming languages from ASM to PL/I. These projects were based on Waterfall project model which was the prevalent software development process in 1981. Later in year 1995 next version COCOMO 2.0 was developed and finally published in 2000 in the book “Software Cost Estimation with COCOMO II” (Stutzke, 1996). There are also many other tools for cost / resource estimation like PRICE, ITK method and SLIM.

What COCOMO offers is the tools to estimate development project schedule and workloads. In Figure 7 different project phases or lifecycle stages of the entire project are presented. The schedule data that can be obtained from COCOMO divides stages into percentage slots based on history data used in study. For better estimates also, company specific history data should be also added to calculations.

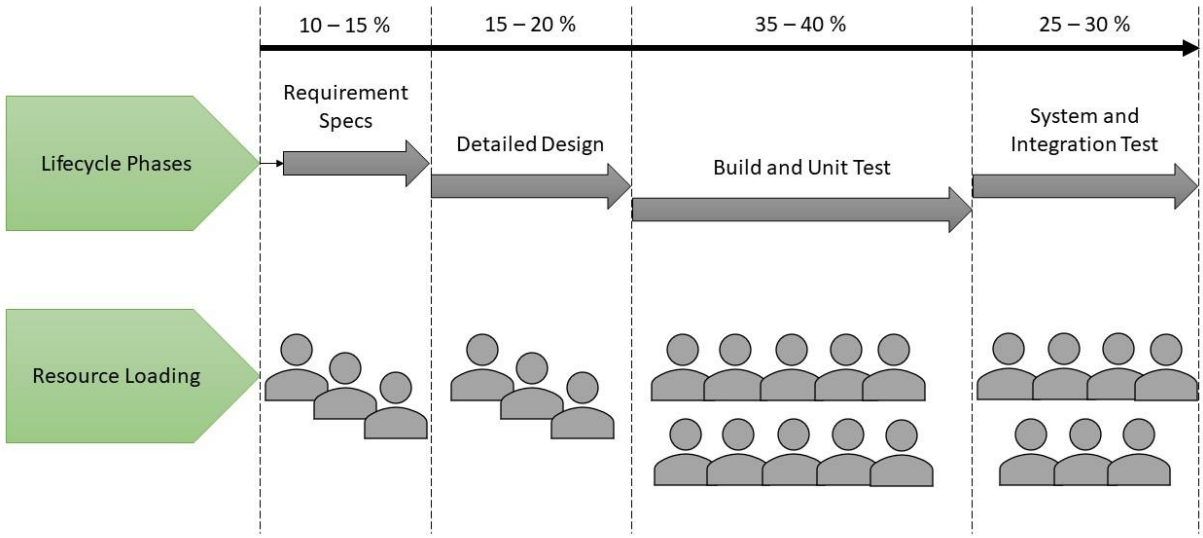


Figure 7. Project effort stages based on COCOMO model

For example, the timetable can take 10 months from the project start date. The timetable then includes all the stages of the project´s life cycle. The whole duration of the project will then be divided into life cycle segments. To use the example of 10 months period, it can be then divided as follows;

- Requirement Specs ~ 1.5 months (10 - 15%)
- Detailed design ~ 2 months (15 - 20%)
- Build and Unit Test ~ 4 months (35 - 40%)
- System and Integration Test ~ 2.5 months (25 - 30%)

5.1 Roles and Responsibilities

In this section different roles and responsibilities are taking deeper investigation and what kind of roles are necessary and recommended for software testing parts of the development process and generating a daily build that can be deployed to any environment. Name of the roles can vary in different sources and companies.

Project Manager

This role is the one, that is liable of knowing these questions related to software project; what, when, why, where and who. On a practical level, this means knowing the project stakeholders and the ability to communicate effectively with everyone. Also, creating and managing the project schedule and budget are project manager responsibilities. Some processes also belong to project manager´s, like scope management, issues management and risk management.

Project manager tasks:

- Create a project plan
- Choose the methodology used on the project
- Recruit project staff
- Run and manage the project team
- Submit tasks to project team members
- Ground a project schedule and define each phase
- Handle deliveries according to the plan
- Deliver regular updates to upper management

Analyst / Consultant

The Analyst / Consultant is responsible for taking and documenting the requirements of business customers before the solution is developed and implemented. In some businesses this person can be called a Business-, Systems-, Business Systems-, Requirements- or Application Analyst / Consultant. Further in this thesis term Consultant is used to describe this role.

Business analyst tasks:

- Assist in project definition
- Collect requirements from business units or users
- Gather business and technical requirements
- Ensure that project deliveries meet the requirements

- Test solutions to validate goals

Technical lead / Technical architect

This role interprets the business demands into a technical decision. Due to this responsibility, it is useful to have the Technical Lead involved in the design phase to hear business requirements from the client´s point of view and makes questions.

The Technical Lead is the team leader of the developer team and operates closely with the developers to offer estimates and technical particulars for the proposed solution. The Project Manager uses this information to create the work program and work breakdown structure documentation for the software project. It is important for the success of the project, that Technical Lead can efficiently report the situation of the software project to the Project Manager to make sure that the possible issues can be resolved immediately. This role is also responsible for creating and implementing practises and standards with the development team.

Software developer / Programmer

The front-end and back-end Software Developer is responsible for creating timeline and cost estimates based on technical requirements that are coming from the Technical Lead or Analyst. This role communicates the situation of the software project to the Project Manager or Technical Lead and is also responsible for making the deliverable packages. To reduce the project risk and ensure the best possible chance for software project to succeed, the other team members needs to efficiently communicate the technical requirements to the Software Developer.

Software developer duties:

- Creating the required solution

Subject Matter Expert

The Subject Matter Expert (SME) has an excellent knowledge of a technology, discipline, business process, product or entire business area.

Subject Matter Expert tasks:

- Review the requirements for the traceability matrix and ensure that the requirements have coverage
- Review test cases for integration testing related with the inventory management system
- Perform code walkthrough for accounts payable interface to legacy system
- Establishing user requirements for payroll application
- Refine and determine the feasibility, completeness and correctness of end user´s requirement
- Provides support associated with the design of the status quo application, its features and its abilities
- Offers input for the construction and design of test cases and business scenarios
- Confirms executed test results

Test Manager / Test lead

The Test Manager or Test Lead must understand how the testing matches into the organizational structure. This means that the Test Manager has to well determine own role inside the organization. This is often done by creating a mission statement or an identified testing mandate. The Test Manager needs to understand the discipline of testing and how the testing process is carried out efficiently and still fulfilling the typical leadership roles of a manager. The Test Manager must implement and master or preserve an operative testing process. This includes the creation of a testing infrastructure that supports solid communication and a cost-effective test framework.

Test Manager duties:

- Defining and implementing the role testing plays within the organization
- Defining the scope of testing within the context of each release / delivery

- Deploying and managing the appropriate testing framework to meet the testing mandate
- Implementing and evolving appropriate measurements and metrics
- Designing, deploying and managing the testing attempt for any given commitment / release
- Managing and increasing the testing capabilities required for meeting the testing mandate
- Retaining a qualified testing personnel

Software tester / Tester / Test engineer

The Software Testers make sure that the solution satisfies the business requirements and that it is free of defects, errors and bugs. Software Testers have an important role in the test planning and preparation phases. They should review and support to the test plans, as well as analyse, review and monitor technical requirements and design methods. Software Testers are involved in identifying test conditions and in creating test models, test cases, test method specifications and test data, and can automate or help automate tests.

Software tester duties:

- Setting up the test environment or assisting doing so
- When the test execution starts, the work required to carry out the test environment tests is started
- Execute and record tests, evaluate the results found and document problems
- Tracking the test and test environment, frequently using tools for testing and often collecting performance data
- Throughout the testing life cycle, software testers review each other´s work, including defect reports, test specification and test results

Here are also some optional roles that are recommended when talking about larger scale development.

Now when the recommended roles are introduced, it is time to check how they can be allocated to an example project. Project is the same 10 months project mentioned earlier. Earlier different project stages where described and what is the usual workload in which stage. In Table 2. these workloads are divided for different roles. For this example, total resource effort is selected to be a 100-person months (PM). Table illustrates the typical resource loading based on the percentage breakup of elapsed time.

Table 2. Resource loading chart

Resource loading chart											
	Months										
Resource	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	Total PM
Project Manager	1	1	1	1	1	1	1	1	1	1	10
Business Consultant	2	3	4	4	2	2	2	2	2	2	25
Programmer			4	4	5	6	8	6	4	3	40
Technical Architect		1	1	1					1	1	5
Tester					3	4	4	4	3	2	20
Total effort	3	5	10	10	11	13	15	13	11	9	100

The percentage of time elapsed may not be exactly the same as the resource month spent in particular life cycle stage. For example, the first project phase (Requirements) could be 1.5 months (15 percent) elapsed time, but as shown in the table the actual resource load spent in M1 and M2 could be only 8 (3+5) person months, which is 8 percent from the total project effort. Normally, the biggest person month effort is spent during the Build and Unit Test phase, which is highlighted in orange. In this example, this Testing related phase takes 52 person months, which is 52 percent of whole project time. When doing resource loading in real project, it is good to keep in mind that it is a complex operation and needs to be treated with care. Incorrect allocation of resources affects project delivery periods and overall quality of outputs. Once the resource loading is done, it is fairly easy to also use for project cost estimations. When total PM is calculated, rate per hour, week or month can be inserted for each resource roles.

5.2 Test Plan

Successful and comprehensive testing of system modifications and enhancements is crucial to both customer acceptance and building customer confidence. System modifications will be tested several times during development and acceptance. Each successive testing step needs to have specific testing criteria and benchmarks defined, as well as clearly defined passing criteria.

During the Analysis phase, test planning is launched to form the high-level plan for testing and to form the universal standards and procedures which must be followed when leading software testing and validation. It becomes a living document that is refined and updated as more information is gained throughout the project. The plan should include specific test cases or scenarios and the expected results. Ideally the plan will include test scripts that document exactly how tests will be performed and what data will be used. Separate testing criteria can be established for each proposed system modification or enhancement.

Data used when testing system modifications or enhancements can come from numerous sources. It can be derived from the customer's daily operations, generated via SQL Server scripts, or simply made up. The specific data source and the structure of the data for testing need to be determined and documented to avoid confusion and delays when the testing begins. This is especially true for any testing that is to be performed by the customer. Inherent in this step is the determination of responsibility for deriving or creating the data used for testing. Once the data source has been decided, the team needs to determine both the amount of data needed for testing the proposed system modification or enhancement and responsibilities for deriving or creating the data needed for testing.

It is important for the customer to understand the various types of testing activities that will be performed and for them to commit appropriate time and resources to assist with and perform testing where appropriate. Refer to the activity descriptions in the phases for more information on each type of test.

The outcome of gathering Quality and Testing Standards will be the establishment of a deliverable Test Plan that examines and defines the following areas:

- Test Objectives and Goals
- Test Approach and Assumptions
- Testing Responsibilities
- Quality and Testing Scope
- Expected Testing Results
- Testing Tasks and Deliverables
- Defining Interactions with other Organizations
- Testing Procedures and Walkthrough
- Test Status Tracking and Reporting
- Test Environment and Resource Requirements
- Testing Schedules
- Definition of Test Specification (Script) Template

At a minimum, determination of the Quality and Testing Requirements Scope should address the following test scenarios:

- **Function Testing** – Independent testing of the system modification (custom code), executed during Development by the Customer and the Consultants.
- **Feature Testing** – Independent testing of the system configuration, executed during Configuration by the Application Consultants.
- **Unit Testing** – Independent testing of the system modification (custom code), executed during Development, by the Application Consultants / Tester.
- **Process Testing** – Complete testing of related functions and features that make up a defined business process, executed during Development by the customer and the consultants.
- **Sub-Process Testing** –Testing the relevant properties of the specified business process, executed during Configuration by the Customer and the Application Consultants.
- **Data Acceptance Testing (DAT)** – Testing steps executed by Data Owners and Key Users in the Development phase before Integration Testing. During DAT, the customer not only checks the data migrated but also confirms that the data may be inquired upon, processed upon and reported upon.

- **Performance Testing** – Testing of business processes and integration, focusing on achieving high transaction volume that is expected during peak times so as to validate that the system performance meets the business requirements. Executed by the implementation project team, typically using test automation software.
- **Integration Testing** – Integrated testing of business processes executed by the Key Users before system sign-off. Focuses on end-to-end business processes such as development, interfaces, reports, and integrations to external systems.
- **User Acceptance Testing (UAT)** – Final testing executed by the Key Users before system sign-off. The End Users selected to execute the UAT must have received appropriate training before the start of the UAT.

All the Testing described above will follow a typical process described in the flowchart (Figure 8.) at the end of this section.

Testing can encompass Business Processes, Workflows, and Reports, which should be reviewed with the customer. It should be communicated to the customer that written test cases/scripts will need to be developed to support the testing efforts. It should not be the Application Consultant's responsibility to define and document these test cases. These test cases should be developed by the Customer, or at least with the Customer’s assistance. The Application Consultant should drive and manage the execution of the testing efforts.

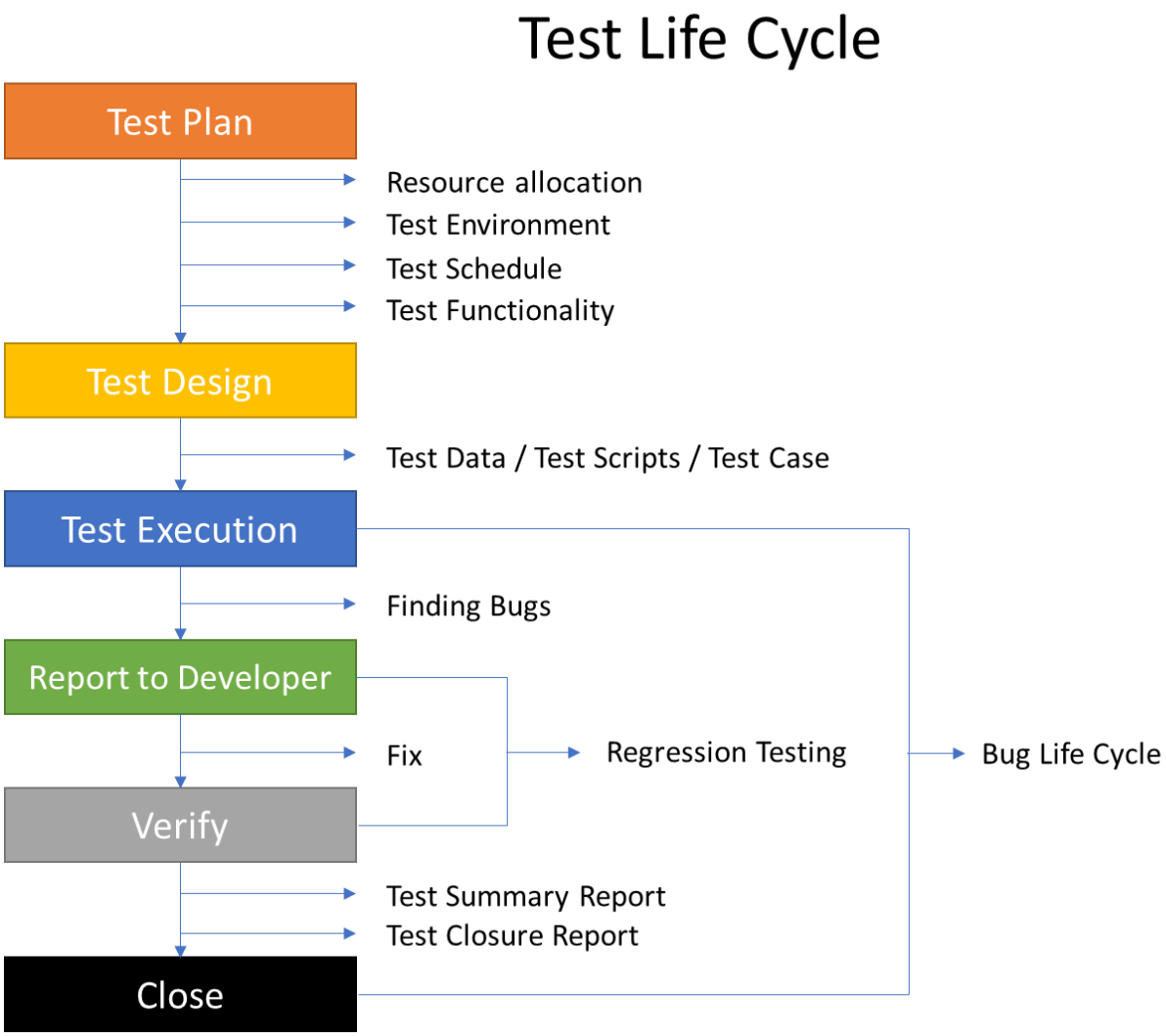


Figure 8. Test life cycle flowchart

6 Financial Losses of Software Projects

For this thesis, 57 different software related public and private sector cases were studied. The cases were selected based on the availability of cost data. Some of these cases are already presented in more detail in Chapter 3. Case studies. These cases were examined from the point of view of how much financial losses they caused. To achieve losses, estimated budgets were removed from total project costs (total cost – budget = loss). There were also three cases were losses were measured in human lives.

Selected cases used in this thesis are also compared with previously published studies. Comparisons have been made to achieve possible correlations regarding cost, time or software testing.

6.1 Analysis

Cases which were studied were all around the world. In Table 3. shows how they were divided.

Table 3. Cases by country

Country	Amount	Percentage %
Canada	3	5%
Finland	11	19%
France	1	2%
Sweden	2	4%
Denmark	1	2%
UK	11	19%
USA	28	49%
Total	57	100%

Most of cases were from USA due to fact that most of them were public sector projects which were public information to study. Also, many of private sector cases were easy to study due to high media awareness back in days, which has brought the facts of the case to anyone to be investigated.

Total financial losses of these 57 cases was 474 milliard euros, which is more than twice the GDP of Finland, based on 2017 statistics. Biggest single loss was around 430 milliard

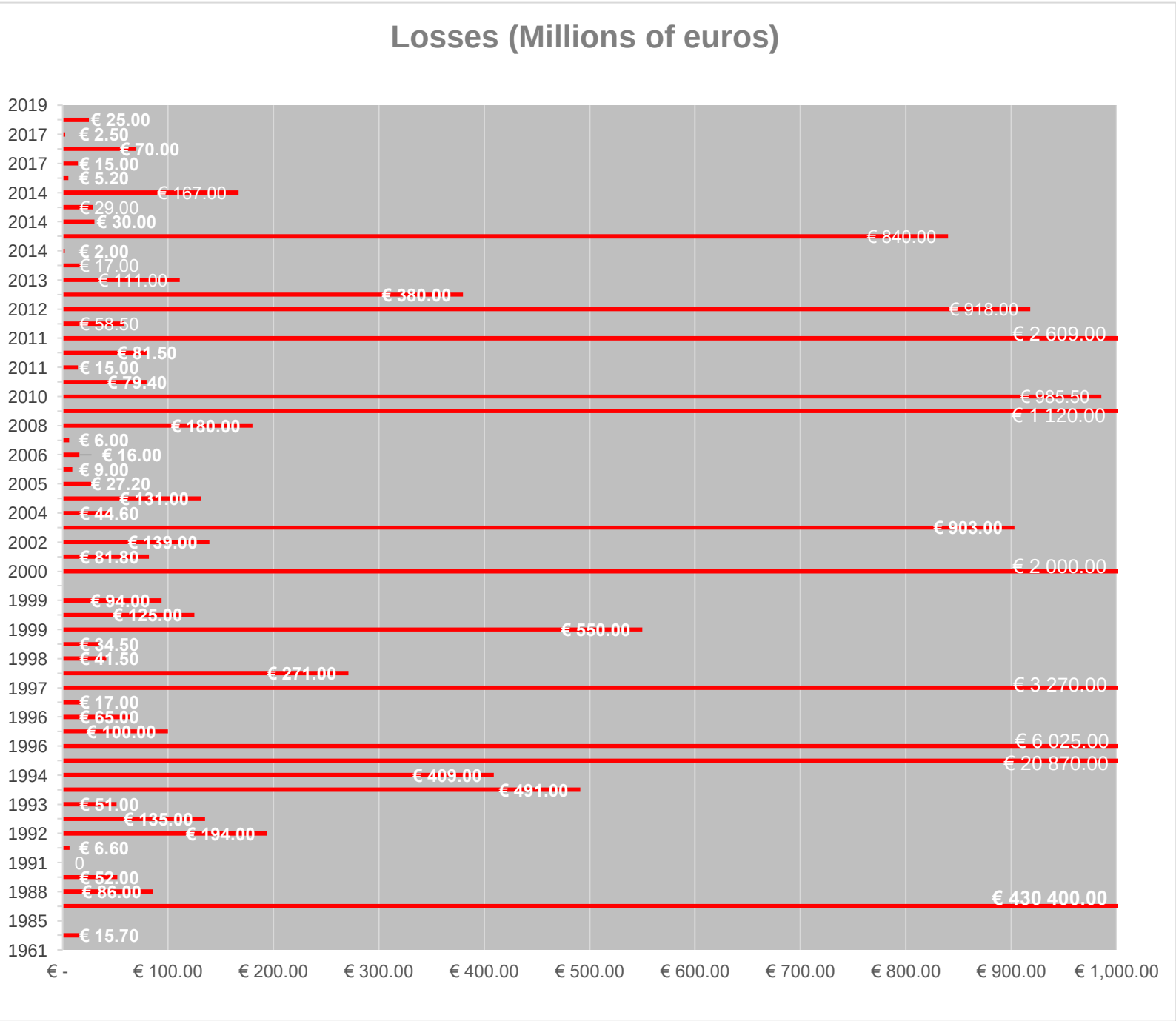
euros, which happened in 1987. This event was called Black Monday and it was stock related catastrophe. During that one-day stock markets around the world crashed. It was a combination of automated stock trading system and human mass panic, which caused fast and widely spread effects. The exact amount of losses is not fully known, but some calculations estimates losses to be around 430 milliard euros.

From Finland there were 11 cases included in the study and 8 of them were from the public sector. All of them are between 2006 and 2018. The total losses of these cases were 259 million euros, which is 23,55 million euros per case. Budget of these cases ranged from 1 million to 200 million euros.

6.2 Findings

Cases were picked between 1962 – 2018. In Table 4. all cases are listed based on their appearing year and what are the costs. If multiple cases in same year, they are listed in separate rows.

Table 4. Financial losses per project year



As seen from the table the trend was been same since 1962. Amount of bigger financial loss cases, above 1 000 million euros, still occurs today. Seems like nothing was been learned from the history. In 1995 author of the Standish Group report wrote, “When a bridge falls down, it is investigated and a report is written on the cause of the failure. This is not so in the computer industry where failures are covered up, ignored, and/or rationalised. As a result, we keep making the same mistakes over and over again.” Maybe this

trend won´t change until IT projects are handled like bridge building projects. From those 57 cases, 17 of them were cancelled which is 29.8%. There are total of 21 cases from the 2010 to 2018 and 9 (42.9%) of them were cancelled during the project.

6.3 Correlations to Other Studies

In 2008, Dr John McManus and Dr Trevor Wood-Harper published a study, were 214 Information System projects were examined within European Union (McManus & Wood-Harper, 2008). Purpose of this study was to examine project failure. Failure was described in those projects that did not met the original time, cost and quality requirements criteria. Study revealed that only one in eight (12.5%) information technology projects can be considered truly successful. In other terms, it means that even 87.5% of IT projects fail.

Project values of these projects ranged from 1 to 80 million euros. Most of them, 87 cases (40,6 %) were between 10 to 20 million euros. Second biggest group was project s under 1 million euro, which was 23,8 %. Actually, 96% of cases were between 1 to 20 million euros. So average IT/IS project in Europe does not cost more than 20 million euros, based on the study of McManus and Wood-Harper. Earlier there was mention about project losses in Finland, which was 23,55 million euros per case, based on 11 example cases which were studied. This means that only in Finland the costs or losses were more, than what was the average project value of one IT/IS project.

In McManus and Wood-Harper´s study, projects were examined at which point or phase of project cancellation or overruns happened. Project overruns were determined based on schedule or budget. In Table 5. project phases are listed based on Waterfall project method. Table also shows what was the amount of completed project after different phases.

Table 5. Project completions, cancellations and overruns based on McManus and Wood Harper´s study in 2008.

Waterfall method lifecycle stage	Number of projects canceled	Number of projects completed	Number of projects overrun (schedule and/or cost)
Feasibility	None	214	None
Requirements analysis	3	211	None
Design	28	183	32
Code	15	168	57
Testing	4	164	57
Implementation	1	163	69
Handover	None	163	69
Percentages	**23.80%**	**76.20%**	**32.20%**

For example, after Feasibility stage no project were cancelled, which is only logical, but after the next stage when project requirements were described 3 projects were cancelled out of 214 and so on. Total cancellation percentage based on their study was 23.8%. Earlier in this thesis a cancellation percentage results were introduced, based on the 57 global project which were studied in this thesis. This result was 29.8%, which is in line compared to McManus and Wood-Harper´s study. This 6% difference can be explained due to smaller sampling for which this Master´s thesis work has been selected.

What is also interested based on the table above, only 4 out of 168 projects were cancelled at or after the Testing phase. Most critical stages are Design and Code, which led to almost all, 43 out of 51 cancellations investigated in the study. Also, these two stages led the most number of project overruns, 57 out of 69 which is 82.6% of all overruns. Testing caused 12 projects to went overrun, budget or time.

Other studies also confirm that actual success ratio of IT project is quite low. Standish Group – Chaos report in 2015 indicates that 29% of projects are successful, based on projects on budget, of cost and with expected functionality. Study made by Oxford University in 2003 shows that only 16% of IT projects are successful.

The exact amount of IT projects globally in one year is hard to estimate. Despite that, studies indicate that software project failures costs even 1.45 trillion euros, annually. (Tricentis, 2016)

7 Summary

Myers wrote the book," Art of software testing" in the 70´s. By this day this book manage to hit the nerve of current time software developing. Even today there are same problems taking considered of than back in the 70´s. In the 3rd Edition of Art of Software Testing Myers describes,

> At the time this book was first published, in 1979, it was a well-known rule of thumb that in a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were expended in testing the program or system being developed. Today, a third of a century and two book updates later, the same holds true. There are new development systems, languages with built-in tools, and programmers who are used to developing more on the fly. But testing continues to play an important part in any software development project. Given these facts, you might expect that by this time program testing would have been refined into an exact science. This is far from the case. In fact, less seems to be known about software testing than about any other aspect of software development (Myers, 2011).

Today there are more articles and books about software testing than back in the 70´s. Still it seems that importance of testing is hard to explain or point out based on studied publications.

Reason for the study came from own experiences in the field of IT. Need for better understanding about how the software testing could be done, plan and develop better.
Original scope of this thesis was to examine the software testing, and what impact poor testing may cause, cost and time. Also, how the bad testing is documented and studied. Based on that information, plan was to create and publish step-by-step guideline about how testing could or should done, to achieve better project success ratio. When gathering the data for the case studies started, it was notable that this data was not so easy to find. Like mentioned earlier about differences of bridge falling and software project falling, software project is usually left without further investigation and documentation about what went wrong. Due that, scope moved to more in the direction on what costs, poorly

designed or failure IT project will cause globally and locally. Despite the reason was it software testing related or not.

Suggestion for the software testing improvement was done based on COCOMO II project schedule and resource workload estimation model. Before that recommended project roles were introduced. These roles are gathered from different testing and project management related frameworks, like Waterfall, Agile, Prince and Scrum. COCOMO recommends that build and testing phase should be allocated to at least 40% resource load, which is the same that J.J Marciniak also introduced in 1994.

Data research for the IT cases was done investigating old news report, blog posts and project final reports. Some public-sector projects also provided more specific final statements about the reasons why projects went wrong. When the information was gathered, idea was to analyze statistics and compare them to other studies to notice if there were any correlations to those.

Results of this thesis were that studied 57 cases affected 474 milliard euros losses. Exact number of how many IT projects failed annually is hard to estimate, but studies have implicated that annually failed projects could cost as much as 1.45 trillion euros. From those 57 cases 17 (29,8%) were cancelled during project. When compared to other studies it was clear that project cancellation percent was in line with other studies, like McManus and Wood-Harper´s which was 23,8 %.

There are plenty of publications about defect costs, bad requirement analyzing or bad management. These are also the main reasons behind McManus and Wood-Harper´s study when failures were examined. When talking about project failure, studies shows that bad management is one the main reason. On other hand, sentence by project stakeholders about the comparative success or failure of projects propensity to be made early in the software projects life cycle. This has caused the phenomenon were many project managers tend to plan for a certain failure rather than success. This was noticed by McManus and Wood-harper, when project stage reports were examined. When complexity of risk that is associated to software project delivery is taken into consider, it is not surprising that only small number of projects are carried out with the original time, cost and quality requirements. Also, when the statistics lean on to that idea that it is more than 70% change that it fails, how can you prepare to something else than failure. One

solution could be developing an alternative methodology for project management based on leadership, stakeholder and risk management. This may lead to a better understanding of management issues and could lead to a more successful delivery of IT projects.

When thinking about future of testing. One big possibility and possible game changer is AI. Artificial Intelligence or Singularity could change the whole idea of testing and how it is done. So far, software development teams are using AI for improving the efficiency of development teams and overall product quality. For now, we can only wait and see what the future will bring

References

Beizer, B. (1983) “Software Testing Techniques,” Second Edition, Van Nostrand Reinhold Company Limited, 1990, ISBN 0-442-20672-0

Bowen, Jonathan P. and Hinchey, Michael G. (1999) High-Integrity System Specification and Design, London, UK

Ceruzzi, Paul. (1989). “Beyond the Limits – Flight Enters the Computer Age, p.203

European Space Agency. (1996) No 33-1996: Ariane 501 – Presentation of inquiry board report. Retrieved November 14, 2017 from (http://www.esa.int/For_Media/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report

Gelperin, D. and Hetzel, B. (1988). “The Growth of Software Testing”, Communications of the ACM, Volume 31 Issue 6, pp. 687-695

Gleick, James. (1996) “A Bug and a Crash”. Retrieved November 14, 2017 from (https://around.com/ariane.html

Godse, D.A, Godse, A.P. (2007).” Fundamentals of programming”, Pune, India

Goodenough, J.B. and Gerhart, S.L. (1975). “Toward a Theory of Test Data Selection,” IEEE Transactions on Software Engineering, pp. 156-173

Jones, Capers, (1986). ed. Tutorial: Programming Productivity: Issues for the Eighties, 2nd Ed. (Los Angeles: IEEE Computer Society Press)

Kaner, C., Bach, J., & Pettichord, B. (2001). Lessons Learned in Software Testing. John Wiley & Sons, Inc., New York, NY, USA.

Kaner, C., Falk, J. & Hguyen, H. (1999). Testing Computer Software, 2nd edition, John Wiley & Sons, New York, pp. 124.

Korhonen, Suvi. TiVi. (2011). VR myöntää: it-ongelmat olisi pitänyt tunnistaa etukäteen. Retrieved May 6, 2018 from https://www.tivi.fi/CIO/2011-10-05/VR-my%C3%B6nt%C3%A4%C3%A4-it-ongelmat-olisi-pit%C3%A4nyt-tunnistaa-etuk%C3%A4teen-3187161.html

Leveson, N. G. and Turner, C. S. (1993) An Investigation of the Therac-25 Accidents. IEEE Computer, 26, 7 (July 1993) pages 18-41. Retrieved November 14, 2017 from https://web.stanford.edu/class/cs240/old/sp2014/readings/therac-25.pdf

Marciniak, J. J. (1994). “Encyclopedia of software engineering”, Volume 2, New York, NY: Wiley, pp. 1327-1358

McManus, John and Wood-Harper, Trevor. (2008). “A Study in project failure”. Retrieved May 5, 2018 from (https://www.bcs.org/content/ConWebDoc/19584

Miller, E.F., “Introduction to Software Testing Technology,” Tutorial: Software Testing & Validation Techniques, Second Edition, IEEE Catalog No. EHO 180-0, pp. 4-16

Myers, Glenford J., Sandler, Corey, Badgett, Tom. (2011). The Art of Software Testing (3rd Edition). John Wiley & Sons, Inc., New York.

Myers, Glenford J. (1979). The Art of Software Testing. John Wiley & Sons, New York

NASA, (2017). Mariner 1. Retrieved November 7, 2017 from https://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=MARIN1

NASA, (2001). Michael King and David Herring. Research satellites for atmospheric sciences, 1978-present. Retrieved November 7, 2017 from https://earthobservatory.nasa.gov/Features/RemoteSensingAtmosphere/remote_sensing5.php

Nuseibeh, Bashar, (1997). "Ariane 5: Who Dunnit?" IEEE Software, p.15–16.

Patton, Ron. (2005). Software Testing (2nd edition)

Rawlinson, J.A. (1987). “Report on the Therac-25.” OCTRF/OCI Physicists Meeting, Kingston, Ontario, Canada.

SEC Filing. (2013). Release No. 70694. In the matter of Knight Capital Ameriacas LLC. Retrieved December 16, 2017 from https://www.sec.gov/litigation/admin/2013/34-70694.pdf

Sommerville, I. (2001). Software Engineering, 6th Edition, Addison-Wesley, 442-444.

Stutzke, Richard, (1996). "Software Estimating Technology: A Survey"

Tassey, Gregory. National Institute of Standards and Technology. (2002). “The Economic Impacts of Inadequate Infrastructure for Software Testing”. Retrieved October 26, 2017 from https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf

Tricentis. (2016). Software Fail Watch. Retrieved May 5, 2018 from https://www.tricentis.com/software-fail-watch/

U.S. Food & Drug Administration. (2017) FDA Statement on Radiation Overexposures in Panama. Retrieved December 16, 2017 from https://www.fda.gov/Radiation-EmittingProducts/RadiationSafety/AlertsandNotices/ucm116533.htm