Ville Salonen

Automatic Portability Testing

Master's Thesis
in Information Technology
October 17, 2012

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Author: Ville Salonen
Contact information: ville.salonen@iki.fi
Title: Automatic Portability Testing
Työn nimi: Automaattinen siirrettävyystestaus
Project: Master's Thesis in Information Technology
Page count: 68
Abstract: The complexity of today's software calls for automatic testing. Automatic tests are even more important when software is developed for multiple environments. By writing automatic tests on unit, integration, system and acceptance testing levels using different techniques, developers can better focus on the actual development instead of performing manual tests and customers can be sure that there isn't a constant regression in features compared to previous versions. If tests are written in the same languages as the actual and using portable tools, the same automatic test suites can be performed on all environments which the software itself is required to support without manifold increase in testing personnel.
Suomenkielinen tiivistelmä: Nykypäivän ohjelmistojen monimutkaisuus johtaa automaattitestien tarpeeseen. Automaattiset testit ovat vielä tärkeämpiä, kun ohjelmistoa kehitetään usealle eri alustalle. Kirjoittamalla automaattitestejä yksikkö-, integraatio-, järjestelmä- ja hyväksyntätestaustasoilla käyttäen useita eri testaustekniikoita, kehittäjät voivat keskittyä paremmin varsinaiseen kehitykseen manuaalitestauksen sijaan ja asiakkaat voivat olla varmoja, etteivät ohjelmiston ominaisuudet hajoa uusien versioiden myötä. Jos testit kirjoitetaan samalla kielellä kuin ohjelmisto ja käyttäen siirrettäviä testaustyökaluja, samat automaattitestit voidaan suorittaa kaikilla niillä alustoilla, joita ohjelmisto tukee, moninkertaistamatta testaushenkilöstön kokoa.
Keywords: Software testing, portability, software development, automated tests.
Avainsanat: Ohjelmistotestaus, siirrettävyys, ohjelmistotuotanto, automatisoidut testit.

Preface

I first dabbled in portability testing as a teenager writing web pages. At the time, Microsoft Internet Explorer and Netscape Navigator were the two major browsers and web pages were notoriously difficult to write so that they would be displayed similarly in both browsers. Later on as a professional software developer I've made software for multiple different platforms such as Windows, Linux and Mac computers, Android and Symbian smartphones and even Microsoft Xbox 360 game console.

Unfortunately most platforms have minor differences between different versions of the platform. One time I was involved in developing a software which was required to work on Windows versions of XP, Vista and 7 and which used Windows Communication Foundation to communicate between different software instances. One of the new features of Windows Vista was User Account Control which was developed to secure the platform by restricting different kinds of operations software can perform. One of these restrictions concerned Windows Communication Foundation. It took us a whole week to locate the source of the problem and another week to refactor the software to work around this problem. This and other similar experiences made me want to learn more about portability and how to use software testing to ensure portability of developed software.

This thesis wouldn't have been completed with the help of my thesis instructor Tapani Ristaniemi who provided valuable insight, my colleagues Ilkka Laitinen and Jaakko Kaski at Sysdrone Oy who helped me keep the writing going by having weekly stand up meetings on the progress of the thesis and Tommi Kärkkäinen who helped shape the idea behind the thesis during the thesis seminar course.

Glossary

Configuration Testing The testing process of finding a hardware combination that should be, but is not, compatible with the program. [54]

Compatibility (1) The ability of two or more systems or components to perform their required functions while sharing the same hardware or software environment. (2) The ability of two or more systems or components to exchange information. [45]

Portability The ease with which a system or component can be transferred from one hardware or software environment to another. [45]

Testing (1) The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. (2) (IEEE Std 829-1983 [5]) The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items. [45]

Contents

1 Introduction

"Program testing can be used to show the presence of bugs,
but never to show their absence!"
Edsger W. Dijkstra

Lockheed Martin F-22 Raptor is a 5th generation fighter plane [56]. Australian Air Marshal Angus Houston has described it as as "the most outstanding fighter aircraft ever built" [10]. Each of them costs about 150 million U.S. dollars [101]. On February 11th, 2007 a group of these fighters were en route from Hickam Air Force base in Hawaii to Kadena Air Force base in Japan [19]. As they were near 180th meridian, the computers aboard the planes crashed. Navigation, communication, fuel and all other systems shut down. The pilots tried to reset their planes but the problem persisted [20]. What had happened? Problems started exactly on 180th meridian which is also known as the International Date Line. When one passes the International Date Line, the local time is shifted 24 hours [7]. The software aboard the fighter planes had passed tremendous amount of testing but the developers and the testers had not managed to test them against this particular scenario. A minor omission with dramatic consequences.

Software permeates many things in the world: it adjusts the braking system in your car, stabilizes photos you capture with your camera, handles your money at the bank and enables to you to share all about your latest vacation to your friends near and far. Whereas the computers executing this software are rarely in error, humans writing the software are not infallible. Even if the software works as its developers intended, it might not be the software the users actually needed or wanted. The backbone of the creation of software is the software development process. This process is in place to ease estimation of work load, to keep the project from falling behind deadlines and a lot of other things. One of these things is the testing of the software.

Software has changed from simple trajectory calculations of the 1940s to hugely complex and intertwined systems of today. When a user encounters a problem, she rarely has a clue where the error actually happened. Even the developers might be perplexed. To bring order to this chaos, computer scientists have formulated many test processes which can be integrated to the software development process. Carefully planned and executed testing allows the users to succeed or just relax using a

well-working software. It also gives the developers a peace of mind. As software is given increasing amounts of responsibility of our daily lives, this testing also becomes increasingly necessary.

Simply making the software work on a single platform might not be enough. In 1940s there were just a handful of computers at all and most of these were designed for some specific purpose such as calculating trajectories of ammo. These days many people have much more powerful devices containing much more complex software in their pockets, backpacks or their desktops. Despite what device the user is using at the moment, they usually want to have access to their data and favorite applications. This means that the software developers have to support a vast array of different hardware and software. This brings us to the concept of portability of a software.

In my experience even the basic testing of the developer software is too often done poorly and in haste. When the portability is added to the mix, developers usually just concentrate on getting the software working on their favorite platform. This leads to forcing a lot of the testing process on the end users. In extreme cases, this ad hoc nature of testing might even to lead to fatal consequences. When the time came for writing my master's thesis, I pondered on many different topics but ultimately chose this. I wanted to delve deeper into the general software testing and find out what other people are doing when faced with the issue of portability.

While this thesis discusses general themes related to portability, the main focus is on topics related to software development performed at Sysdrone Oy (later Sysdrone) and more specifically its health technology projects. Health technology software is increasingly targeted directly at the actual people who monitor and improve their own health. This leads to a wide array of potential devices used to access and execute these health technology software. The software must function correctly no matter which supported platform it is used. Malfunction is at least annoying and in worst case, even fatal.

1.1 Objective of the Thesis

The objective of the thesis is to plan a practical automatic portability testing environment for the kind of software typically developed at Sysdrone. To support the planning process, this thesis is used in defining software testing and portability and describing different aspects of these concepts. The most important question this thesis answers is:

- *How can Sysdrone use automatic testing to detect and mitigate portability issues in its software development process?*

Related supporting questions are:

- *What is automatic testing, why is it important and how it can be done?*
- *What is portability, what kind of problems are related to it and how can these issues be detected automatically?*

First supporting question is used to explore why automatic testing is so important, how it is used in software development process and what kind of benefits and costs it presents. Automatic testing is a common subject in software development but according to my experiences it is usually understood in too shallow context. My goal is to gain in-depth knowledge about different kinds of testing levels, test techniques and how these can be interwoven with software development processes.

Second supporting question is used to explore why portability is an important feature of a software, what kinds of problems are related to portability and why performing portability manually is not recommended. Based on my experiences, portability is a major issue and usually software is developed primarily on a single platform. This may lead to a situation where other platforms are usually tested only later on in the software development process and the testing is performed manually. Manual testing leads to potentially unsystematic testing and major resource requirements.

1.2 Research Methods

The research of software testing and portability is done mainly by reading academic and other professional literature. Some real world examples and supporting data are found from blogs, Internet statistics and discussion boards such as Stack Overflow.

The information about software being developed at Sysdrone is achieved by my 2-year experience of the company and by consulting my colleagues at Sysdrone.

Plans for automated portability testing environments are made with the supporting theoretical background of the previous parts of the thesis and by researching existing portability testing environments which are already either planned or developed elsewhere.

1.3 Constraints of Research

Software development, portability and software testing are complex concepts with a vast array of different aspects. This thesis is mostly concerned with the kind of software testing and portability issues which are typical in the software projects developed at Sysdrone.

Although the thesis concerns portability testing, it is more focused on the kind of portability testing which can be automated. Thus this thesis does not delve into the topic of for example manually testing a huge array of Android handsets. This kind of testing, although useful, is so labor intensive that small development teams at Sysdrone cannot perform such testing.

1.4 Structure of the Thesis

Chapter 2 defines software testing, provides arguments supporting its necessity, description of its complexity and how automation of tests help mitigate the effects of the complexity.

Chapter 3 defines portability, provides arguments for why portability is increasingly important feature of a software, describes why portability is a hard problem and how software portability can be tested.

Chapter 4 defines different test levels and generally describes, which kind of aspects of a software each of these levels tests.

Chapter 5 specifies different kinds of test techniques. The list of test techniques is mainly derived from IEEE's Software Engineering Body of Knowledge but it is also supported by various other literature sources. Each test technique is inspected from the point of view of portability and automation.

Chapter 6 presents Sysdrone as a case study of a software development company. The types of software developed at Sysdrone are categorized and the need for encompassing testing for software related to medical devices is discussed.

Chapter 7 contains plans for how Sysdrone could develop automatic portability testing environments for different types of software typically developed at the company.

Chapter 8 contains the summary of the thesis.

2 Software Testing

This chapter contains the definition of software testing, arguments supporting its necessity, description of its complexity and how automation of tests helps mitigating the effects of the complexity.

2.1 What is Software Testing?

Definitions for software testing can be hugely varied. One commonly heard everyday description of why testing is performed is that it is done to ensure or improve the quality of the software. This description, although true, is too vague to be of any use when assessing software testing because "quality" has not been precisely defined. According to motorcycle enthusiast and philosopher Robert M. Pirsig [83]: "Any philosophic explanation of Quality is going to be both false and true precisely because it is a philosophic explanation. The process of philosophic explanation is an analytic process, a process of breaking something down into subjects and predicates. What I mean (and everybody else means) by the word 'quality' cannot be broken down into subjects and predicates. This is not because Quality is so mysterious but because Quality is so simple, immediate and direct." Thus a more precise definition is required.

IEEE Computing Society's Guide to Software Engineering Body of Knowledge [44] defines software testing as follows: "Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems. Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior." The definition includes the mention of quality but it defines specific ways to achieve better quality. This definition also underlines the fact that often it is not possible to test all scenarios: instead a specific subset of the most important scenarios is selected for verification.

According to Cem Kaner [55]: "Software testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test." The information produced by testing can be used by the stakeholders in various ways. If the quality of the product or service has been proven satisfactory, stakeholders can decide to publish the product to users. In another case the testing might have identified a life-threatening and unhandled scenario which needs

further resources so that the risk can be mitigated or possibly even completely eliminated.

The definition of testing has evolved over the years [22]. In year 1979 testing was defined as "the process of executing a program or system with the intent of finding errors". One definition from year 1983 was: "Testing is any activity aimed at evaluating an attribute of a program or system. Testing is the measure of software quality." More recently in year 2002 testing was defined as "a concurrent lifecycle process of engineering, using, and maintaining testware in order to measure and improve the quality of the software being tested." Of course, there are variations between definitions by different people in any given year but these examples illustrate the shift in attitudes regarding testing. Software testing is being regarded as a more integral part of the software development process than in the past.

However, software testing can be a lot of work and dangerously often when the deadlines are closing in the software testing is the first thing to be left out. This may partly be because software testing usually pays back only in the long run. Thus it is easy to adapt a similar mind-set that was used in the build-up of the economic crisis in late-2000s: "I'll Be Gone, You'll Be Gone" [36]. The consequences of this kind of behavior are discussed in the section 2.2.

2.2 World Without Software Testing

What would happen if software testing wasn't performed? One example was presented in the Introduction of this thesis but there are countless others.

Peter Sestoft [100] has described very unfortunate results from a poorly performed software testing: Faulty baggage handling system at Denver International Airport led to a delay of a year for the opening of the whole airport and caused financial losses of 360 million dollars. Ariane 5 rocket used control software which was originally developed for Ariane 4 but the code was not tested with the new rocket and the rocket launch failed and caused losses of hundreds of millions of dollars. Even though financial losses of these kinds of scales are very unfortunate, poorly performed testing can also lead to fatal consequences: Patriot missiles used in Gulf War performed imprecise calculations and the resulting rounding errors made the defensive missiles miss incoming Scud missiles and led to the death of American soldiers in 1991. Therac-25 radio-therapy equipment and its faulty control software led to fatal radiation doses given to cancer patients in 1987.

In United States, National Institute of Standards & Technology estimated that the inadequate infrastructure for software testing resulted in an annual national cost of

22.2-59.5 billion U.S. dollars in year 2001 [98]. Based on the above evidence it is safe to assert that the software testing is indeed important.

2.3 Complexity of the Software Testing

Bugs are result from any of these reasons [103]:

- User executed untested code. Time constraints often result in at least partially untested codebase.
- Code statements were performed in a surprising order.
- The user entered untested input values. Testing all different input value combinations can be very difficult.
- The user's operating environment was never tested. For example building a large network with thousands of devices takes a lot of resources and such testing environment is rarely built for internal testing.

Unfortunately computer cannot guarantee that the software works correctly. When a program code is compiled as a runnable software, compiler only makes sure that the form of the software does not contain errors: the errors in the meaning of the software cannot be detected by the compiler. However, finding these errors in the meaning of the software can be found in better ways than just randomly experimenting with the software: these better ways are collectively called software testing [100].

On the surface, testing the software doesn't sound so hard. You write the software and verify that it works correctly. Unfortunately even a simple program can be so complex that the programmer can't think of all the possible combinations.

Consider the following program: "The program reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral." What kinds of tests would be required to adequately test this simple program? Myers [73] lists 14 test cases: 13 of them represent errors that actually have occurred in different versions of the program. One additional requires that each of these test cases specify the expected output from the program. According to Myers' experience, qualified professional programmers score on average only 7.8 out of the possible 14.

The above program can very likely be written with most programming languages in less than 50 lines of code. To put that figure in perspective, Linux kernel version 2.6.35

```
def is_leap_year(year):
    if year % 400 == 0: return True
    if year % 100 == 0: return False
    if year % 4 == 0: return True
    return False

assert not is_leap_year(1900), '1900 should not be a leap year.'
assert is_leap_year(1996), '1996 should be a leap year.'
assert not is_leap_year(1998), '1998 should not be a leap year.'
assert is_leap_year(2000), '2000 should be a leap year.'
assert not is_leap_year(2100), '2100 should not be a leap year.'
```

Figure 2.1: A simple example of a test program that verifies leap year algorithm.

contained 13 million lines of code [53]. Windows Server 2003 contained 50 million lines of code: its development team consisted of 2,000 people but its testing team consisted of 2,400 people [59].

It is not effective to apply the same type of testing to all different type of software. For example when developing a simple game, small bugs or issues under intense CPU load are usually not as serious as they would be in a medical device or space shuttle. If the same type of testing which is performed for space shuttle would be required for all mobile games, the increased resource requirements would likely lead to a massive decline of the amount of mobile games. This further complicates the testing process because a development team cannot simply copy a testing process from other project with a good reputation of catching bugs and be done with it because it might prove to either be a lot more exhaustive than required or it might not detect defects which were acceptable in the other project but which are unforgivable in this project.

2.4 Automatic Testing

Figure 2.1 presents a simple example of how a programmatic unit test suite can be implemented. The tested function `is_leap_year` is given an integer value and the function returns `True` or `False` based on whether the given year is a leap year or not. Below the function is a list of test cases and their expected values. For example year 2000 is a leap year because it is divisible by 400 but year 2100 is not a leap year even though it is divisible by 4 because it is also divisible by 100 but not divisible by 400. The test program provides some kind of output whether tests succeeded or failed. In

this particular test program the output is simply printed in the console.

Because automatic tests do not require manual resources, they can be performed easily. For example, when developer is writing the software, she can easily perform all or specific tests to ensure that new changes won't break any existing tests. Even though tests are automatic, some of them may be so complex or there may be so many tests that performing all of them can take many hours or even days. This kind of testing would interfere with the development. Because of this all tests in bigger software projects are only performed during the night or even only when releasing a new build.

According to Robert C. Martin [61], not only is manual testing highly stressful, tedious and error prone, it also immoral because it turns humans in machines. If a test procedure can be written as a script, it is also possible to write a program to execute that test procedure. This leads to cheaper, faster and more accurate testing than the manual work performed by humans, and it also frees humans to do what we do best: create.

According to research, code reviews are one of the best ways in finding bugs. Unfortunately this sort of code reviewing is a manual process and gathering the right people for performing code review or maybe even having to train them takes a lot of effort. Because of this, it is impossible to analyze the whole codebase with code reviews. Fortunately same type of analysis can to a degree be automated by employing static code analysis. This kind of tool analyzes either the program code itself or the compiled bytecode and compares it to a database of common bug patterns. Catching bugs as early as possible makes fixing them easier [57]. Because static code analysis can be performed even in realtime as the developer is typing the code, bugs can be caught very early. Static code analysis cannot be used to detect all possible problems: for example an usability problem or incorrectly implemented mathematical algorithm cannot be detected if they are technically correct program code.

Automatic testing is especially important with rapid application development. Rapid application development attempts to minimize development schedule and provide frequent builds. This is done so that user can evaluate the evolution of the software in small increments and provide feedback to development team so they can ensure that the software reflects the actual needs and preferences of the user [25]. Because of the rapid speed, there is no time for a comprehensive manual testing each time a build is shipped but if the testing would not be performed, the specified level of quality could not be ensured. The best way to achieve rapid shipping of new builds without compromising the quality is by employing automatic software testing.

2.5 Chapter Summary

Software testing is a process to improve the quality of a software under development. However quality should be defined more explicitly in the context of each software. Software testing is an important part of the software development and its omission can cause a lot of serious problems. Unfortunately nearing deadlines often result in omitted or poorly performed testing.

Complexity of the modern software projects leads to a vast number of different operation combinations which have to be tested. Even very simple pieces of software can have so many combinations that even qualified professional programmers fail to recognize missing test cases. Thus automated testing is a preferred way of testing software in many complex and detailed test cases. Even though automated testing is a very valuable tool, all software testing cannot and should not be automated.

3 Portability Testing

This chapter introduces the concept of portability, lists different types of portability, presents reasons for developing portable software, contains the definition of portability testing, and description of problems related to portability and portability testing.

3.1 History of Portability

The amount of work required for moving a software from one environment to another is dictated by how portable the specific software is. Ideally software could moved between environments without any modifications to the source code. Unfortunately, this is rarely possible in the real world [38].

Peter J. Brown has given the following definition for portability [16]: "Software is said to be *portable* if it can, with reasonable effort, be made to run on computers other than the one for which it was originally written. Portable software proves its worth when computers are replaced or when the same software is run on many different computers, whether widely dispersed or at a single site."

There is no single specific definition for an environment of a software. Over the years, the environments have changed considerably.

First electronic digital computer, ENIAC, was developed for calculating ballistic trajectories. The programs for ENIAC were first developed on paper and later input in the computer itself via physical interface consisting of hundreds of wires and 3,000 switches [23]. ENIAC had no storage for programs but instead they were only stored in the position of ENIAC's wires and switches. This was a problem because when users wanted to run some another program on ENIAC, they had to change it physically by changing the wires and switches. This could take days [96]. The only environment of ENIAC programs was the ENIAC itself.

Manchester Baby was the first stored-program computer [96]. Stored-program computer is able to store programs in its memory and change between them without users changing any wires or flipping thousands of switches. Despite the move to memory storage, the environment of the programs was still the machine itself.

Programs for these early computers were written directly in machine language which had hardware-specific instructions such as subtraction. Machine languages are also called first-generation programming languages [34]. These computers had no operating

systems so the only environment of programs running on these computers was the hardware itself. The machines differed from each other in such dramatic ways that there was no way to directly move a program from one kind of machine to another.

One of the biggest problems with machine languages was that they were closer to the language of the machine instead of the language of the user. Writing programs in 0s and 1s was difficult and changing larger programs was practically impossible. First step towards higher levels was the development of assembly languages. Assembly languages are symbolic representations of the machine languages. Programs written in assembler were translated to machine languages and thus their portability is nearly impossible. Assembly languages are called second-generation programming languages. [34] An example of an assembly program can be seen in code listing 3.1.

```
# verify that   1 + 2 + 3 + .. + n = (1 + n) * n / 2 is true for
# n = 3. Later on, we'll write a program for arbitrary n.
#

main:                                   # beginning of the program

# initialize $8, $9, and $10 as 1, 2, 3.
   li $8, 1                             # $8 now contains 1
   li $9, 2                             # $9 now contains 2
   li $10, 3                            # $10 now contains 3

# compute 1+2+3, result in $11
   add $11, $8, $9                      # $11 = $8+$9 = 3
   add $11, $11, $10                    # $11 = $11+$10 = 3+3 = 6

# compute (1+3)*3/2, result in $12
   add $12, $8, $10                     # $12 = $8+$10 = 1+3 = 4
   mul $12, $12, $10                    # $12 = $12*$10 = 4*3 = 12
   div $12, $12, $9                     # $12 = $12/$9 = 12/2 = 6

# subtract, result in $13
   sub $13, $11, $12                    # $13 = $11-$12 = 6-6 = 0
                                        # verified
j  $31                                  # end the program
```

Figure 3.1: A program to verify that 1+2+3=(1+3)*3/2. [75]

First true general-purpose computers were developed in the 1950s and 1960s. Whereas previous computers were typically built for some specific purpose such as calculating ballistic trajectories, general-purpose computers could used for many different kinds of computers. During the same time high-level programming languages (also called third-generation programming languages) were developed. First of them, Fortran was introduced in 1954 and finally developed in 1957. High-level programming languages abstract the differences between different computers [34]. Compilable high-level languages are compiled from the source code written by the programmer to objects which contain the program in a machine language with some additional data of entry points and external calls [12].

Up to the introduction of operating systems, computers could only run one program

at a time so the environment of a program consisted only of the machine itself. First operating system, GM-NAA I/O was developed by IBM for 704 mainframe in 1956 [46]. With GM-NAA I/O the environment consisted of both hardware and operating system. GM-NAA I/O didn't allow for multiple programs to run simultaneously[46] so other programs couldn't interact nor interfere with a program.

As computers were loaded with operating systems, portability didn't mean just transfering a program from one hardware to another. In 1978 Ritchie and Johnson [50] wrote: "The realization that the operating systems of the target machines were as great an obstacle to portability as their hardware architectures led us to a seemingly radical suggestion: to evade that part of the problem altogether by moving the operating system itself."

According to an article published in The CPA Journal [35], the definition of portability hasn't changed from the 1970s. These sources still define portability as an ability to move a software from hardware platform and/or operating system to another. I argue that this is too narrow of a definition.

Oglesby et al. [78] have made the following observation: "The history of software development has shown a trend towards higher levels of abstraction. Each level allows the developer to focus more directly on solving the problem at hand rather than implementation details." Because the nature of software development has changed due to increasing levels of abstraction, I argue that the definition of portability should be broadened to better reflect the real problems developers face when moving a software from one environment to another.

For example, languages such as C# and Java are not compiled directly to machine code but instead they are compiled to bytecode. This bytecode is then executed in a virtual machine which doesn't provide a direct access to neither operating system or hardware for the program. There are ways to break out of these restrictions with tools such as Java Native Interface but one of the key benefits of using these kinds of languages is the ease of portability so using native interfaces is disadvantageous.

As another example an HTML5 application can't interact directly with operating system nor hardware. The Internet browsers act as operating systems for web applications.

Defining the portability of these kinds of software in terms of hardware and operating system support is irrelevant because the developer has no access to hardware and operating system. However these types of software still face portability problems such as in the case of executing a Web application in different browsers. Thus the problem of portability is not simply solved by abstracting just hardware and operating system.

As hardware becomes more general-purpose, the specialty functionality moves from

hardware to software. For example whereas before some software would have required some specific math co-processor which the software would have been written directly against, these days software usually require some third party libraries to provide the needed functionality. Portability between different available third party libraries becomes the more essential problem for developer to solve instead of solving the problem of portability between specific hardware components.

3.2 Different Types of Portability

There are a lot of different types of portability. Typically only specific types of portability are related to a particular software. Thus it is important to define the portability context of a software before planning how to test portability of a software. This is by no means a comprehensive list but are listed more as examples of what type of portability issues might be encountered.

One of the most typical portability issue is the wide selection of operating systems. Mainstream desktop and laptop operating systems are Microsoft Windows, Apple Mac OS X and Linux-based distributions. On the mobile front the most common ones are Google Android, Apple iOS, Windows Phone, RIM BlackBerry OS and Nokia Symbian. In addition to these there are lot of more specialized operating systems in embedded devices, mainframes and other different hardware platforms.

Embedded software have a wide array of different hardware architecture. These different hardware architectures have different characteristics such as whether they are big endian or small endian. Embedded software will not be discussed in more detail in this thesis because of the reasons described in 1.3 but hardware architecture may present issues even on mainstream computers and operating systems. For example, Apple Inc. sold its computers with four different kinds of processors in years 2005 and 2006. Previously Apple had used PowerPC-based processors which were available as both 32-bit and 64-bit versions. In year 2005, Apple announced its plans to switch over to x86-based processors manufactured by Intel [3]. First versions of its computers came with 32-bit Core Solo and Core Duo processors [4] but at the end of year 2006, Apple released its first computer with 64-bit x86 processor, Core 2 Duo [5]. 64-bit support has since been dropped in the most recent version of OS X but at the introduction of 64-bit Intel processors, there were four different combinations of Apple computer processors: 32-bit Intel, 32-bit PowerPC, 64-bit Intel and 64-bit PowerPC. With the introduction of Windows 8 and its ARM supports, developers on Windows platform will soon be in a similar situation.

Browsers are one major source of portability issues. These issues are described in

more detail in sections 3.4 and 7.2.

Hosting a web server application requires a hosting site. A hosting site can be a privately owned computer with a typical Internet connection but a bigger traffic requires more serious hardware and network capacity. It is possible to buy a dedicated high-volume network connections and powerful server machines to host a site but buying enough for peak traffic without spending a lot of money can be tricky. One way to solve this is to use cloud-based hosting. There are many options for this type of hosting. Some of the most well-known are Amazon Web Services and Microsoft Azure but there are many others.

In addition to these there are some third party libraries which provide major functionality. One example found in the realm of games and other graphic-intensive software are APIs such as OpenGL and DirectX. OpenGL can be found on most operating systems but DirectX is limited to Microsoft Windows and Xbox 360 gaming console. When a major game title is planned, the developers have to decide whether to support both.

3.3 Motivation for Portability

Portability requires extra resources. Why then is portability an important feature of a software? As Peter J. Brown stated [16]: "It costs planning and effort to produce software that is portable. Moreover, on any one computer, a portable program may be less efficient than a specially hand-tailored one. Nevertheless, given the huge cost of rewriting non-portable software, an investment in portability is normally one that will repay handsomely."

Even though you would develop a very interesting software, customers will rarely switch their existing platforms just for your software. This might be the case in professional high-end software but even users are probably accustomed to the existing ways of doing things and the switch would be costly not just because hardware and software costs but due to decrease in productivity. Portability of a software can also be seen as a safety guarantee for the developing organization: they are less susceptible to the changes done by the developer of the environment. There are multiple documented cases [37][84] of how developers have put in a lot of work to create an iOS software just to see its access denied to the Apple's App Store and thus to all the iOS devices (except for ones using rooted firmwares which allow software to be installed even though the software has not received Apple's blessing).

Typically possible users of a software use varying devices and operating systems. For example if an organization wants to develop a software for mobile devices which

would potentially be available for 80% of the population, the software would have to support four different mobile operating systems: SymbianOS, iOS, Android and BlackBerry OS [92].

Game industry is also concerned with portability. Most major game titles are currently released on PC computers and Microsoft Xbox 360 and PlayStation 3 consoles. In addition to these, the game may also be released on Mac computers and Nintendo Wii console.

For example Battlefield 3 video game was released in 2011 for three platforms: PC computers and Xbox 360 and PlayStation 3 consoles. On November 6th, 2011 sales figures for each platform were 500,000 U.S. dollars, 2.2 million and 1.5 million respectively [11]. In this case choosing just a single supported platform would have almost halved the sales rate.

3.4 Problems of Portability and Portability Testing

Each additional supported platform requires some additional resources. At least the hardware and the software must be acquired and integrated into the testing process. This can lead to a lot of different hardware and software combinations. jQuery Mobile, which is a JavaScript library for creating mobile user interfaces to web pages, is an example of this: in their test lab they roughly 50 different phones, tablets and e-readers to make sure their library performs properly on each platform [52].

Just providing a working software on a platform might not be enough. For example there are certain differences between Windows, OS X and Linux desktops which the users have grown accustomed to. One such difference is the placement of OK and Cancel buttons in dialog windows. On Windows, OK button is on the left and Cancel button on the right. On OS X these are reversed and on Linux their placement varies between different window managers. To create great software, it is recommended for the software to feel native to its execution platform.

Some features might not be available on all platforms. It is up to the developers to decide whether to go with the lowest common nominator, which could lead to inferior product compared to the competitors, or to create different versions for each platform to take advantage of each platform's features. Different versions introduce different test plans and platform-specific test suites which in turn require additional resources. There are no right answers to these decisions: they require careful balancing.

Portability testing can be a very complex process. As discussed in section 2.3, testing even a trivial program can be difficult in itself and by introducing multiple platforms the labor-intensiveness is multiplied. For example, by supporting browsers

of 80% of the international users, a software would have to compatible with Internet Explorer, Google Chrome and Mozilla Firefox browsers (based on data from November 2011) [91]. Unfortunately, just testing the software with three browsers is not enough because the browser usage is further divided into different versions of these browsers. By supporting the same 80% share of the users, the software would have to be tested with Chrome version 15, Internet Explorer versions 7.0, 8.0 and 9.0 and Mozilla Firefox versions 3.6, 7.0 and 8.0 [91]. Thus the testing would have to performed with at least seven different desktop browsers. Things are even further complicated if mobile browsers should be supported. According to International Data Corporation, the amount of mobile Internet users surpasses the amount of wireline users in United States of America by year 2015 [48]. By supporting the same 80% share of the mobile users, the software would have to support Opera, Android, iPhone, Nokia and Blackberry browsers [91]. In addition to this, mobile browsers also contain version differences [29].

3.5 Chapter Summary

Nature of portability has changed over the years. In the beginning of the computer era there were just a handful of vastly different machines. Then the main issue in portability was supporting the hardware, as the software typically directly interacted with the underlying machine with no abstraction layers in between. Today a lot of developed software doesn't even have access to the machine below.

Different software face different types of portability issues. It is up to developers to find out what these issues are, how affect the workings of the software and how to mitigate portability problems.

Even though ensuring portability is not an easy task, it is important for many reasons: Company doesn't have to bet its business on just a single platform. The pool of potential users is larger and thus the potential economical or other benefits are larger as well.

4 Testing Levels

This chapter presents and compares multiple definitions of different testing levels. The most often-used testing levels are described more in-depth.

4.1 Defining Testing Levels

Software testing is often performed on multiple levels. For example, a developer tests the software at a very low level because often she wants to verify that a specific piece of code performs as expected. However this test level is rarely a meaningful level for customer. At so low a level evaluating the whole software can be very difficult. Instead customer usually tests the software by using its user interface and verifying that the software works as expected from the point of view of an actual user.

Validation and verification testing are often mentioned when discussing software testing. However they are neither testing levels nor methods. Barry W. Boehm [15] defines verification as a process which answers the question "Are we building the product right?" and validation process answers the question "Are we building the right product?".

Testing levels as a term are also used to refer to maturity levels of testing process in an organization. According to a definition by Ammann & Offutt [1], there are five different levels ranging from Level 0 "There's no difference between testing and debugging" to Level 4 "Testing is a mental discipline that helps all IT professionals develop higher quality software". However this categorization is outside of the scope of this thesis because this thesis is more concerned with testing on a more technical level and not on an organizational level.

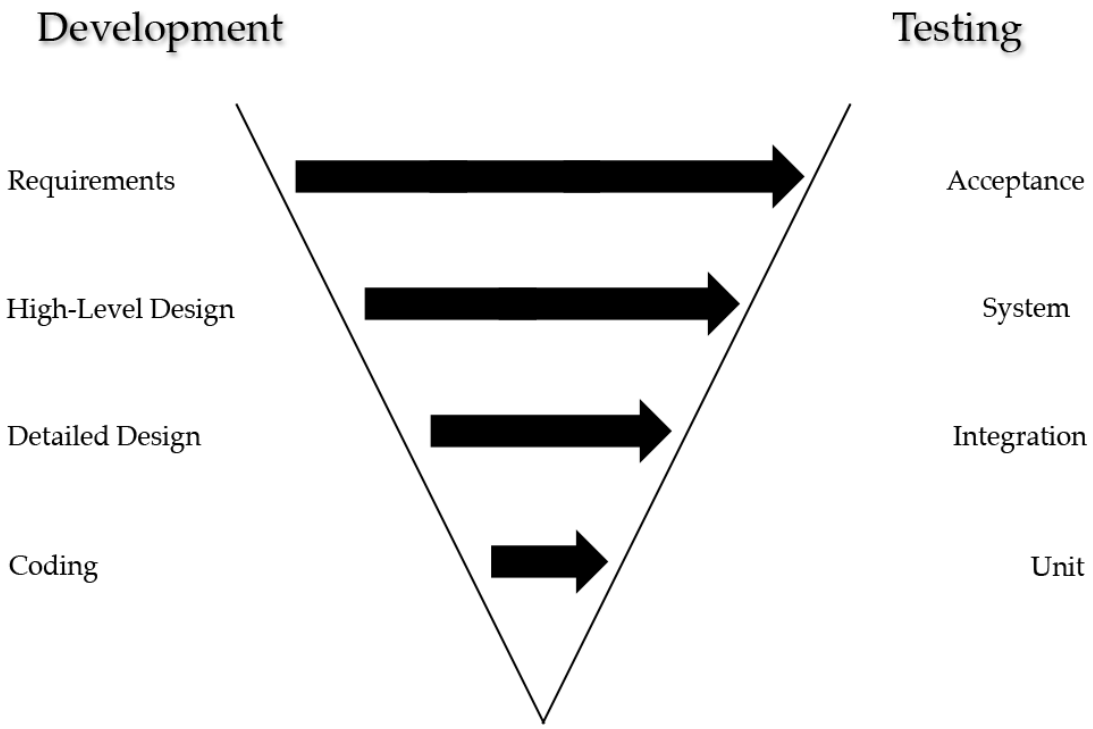
There is no consensus of the exact definition of different test levels. However different definitions contain a lot of common themes.

According to IEEE Computing Society's Guide to Software Engineering Body of Knowledge, there are three different target levels in software testing: unit testing, integration testing and system testing. These levels are defined as follows [44]:

- "Unit testing verifies the functioning in isolation of software pieces which are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units."

- "Integration testing is the process of verifying the interaction between software components."
- "System testing is concerned with the behavior of a whole system."

There are four different test levels according to Craig et al. [22]. Each level of testing has a corresponding development level. Code is tested by unit tests, detailed design by integration tests, high-level design by system testing and software requirements by acceptance testing (figure 4.1). All these levels should not necessarily be used in every software project. Instead test manager should select the used levels based on a number of variables such as complexity of the system, number of unique users and budget. Unfortunately there is no formal formula for this decision.



[22]

Figure 4.1: Test levels and their corresponding development levels.

Ammann et al. [1] present two different kinds of testing levels. First kind is based on software activity and the levels are defined as follows:

- "Acceptance Testing - assess software with respect to requirements.
- System Testing - assess software with respect to architectural design.
- Integration Testing - assess software with respect to subsystem design.
- Module Testing - assess software with respect to detailed design.
- Unit Testing - assess software with respect to implementation."

These levels differ slightly with object-oriented software because the design blurs distinction between units and modules. Testing of a single method is usually called intramethod testing while testing of multiple methods is called intermethod testing. If a test is constructed for the whole class, it is called intraclass testing. Testing

of interaction between different classes is called interclass testing. The first three are variations of unit and module testing and the interclass is testing is a type of integration testing [1].

The rest of this chapter contains more in-depth definitions of different levels. The set of testing levels is based on the definition of Craig et al. as it most closely resembles the testing levels used at the target organization of this thesis.

4.2 Unit Testing

Roy Osherove [79] defines unit test as follows: "A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A 'unit' is a method or function." By isolating the unit from other units, tests do not rely on external variables such as a state of database or a network connection.

Robert C. Martin [62] describes devising a test for a piece of software which he developed for an embedded real-time system which scheduled commands to be run after a certain amount of milliseconds. His test used an utility software in which by tapping a keyboard he scheduled a text to be printed on the screen 5 seconds later. Then he hummed a familiar song and tapped keyboard in rhythm and 5 seconds later he hummed the same song again and tried to make sure that the pieces of text would be printed in perfect synchronization. According to Martin, this not an exemplary way of unit testing: instead it is an imprecise test at best and at worst it is very error-prone and labor-intensive because it requires manual interaction with the software. A good unit test for this program would be one where the unit subjected for testing would be isolated from the computer clock. This kind of unit test would not only remove the ambiguous results of repeated manual tests: it would also allow developer to manipulate time perceived by the tested software to all kinds of arbitrary situations such as testing the timing at the moment when clocks are adjusted due to daylight saving time. Testing this particular situation by relying on the actual computer clock would only be possible twice a year.

Ammann et al. [1] present Pentium Bug case as a real-life example of how inadequately done unit testing can prove very costly if problems are found in the later stages of testing or even after the product release. A Pentium processor developed by Intel is an example of this. The Pentium Bug was a defect discovered by MIT mathematician Edelman and the bug resulted in incorrect answers to particular floating-point calculations. Both Edelman and Intel claimed that this would have been very difficult to catch in the testing but according to Ammann et al. [1] it would have been very easy

to catch with proper unit testing.

As unit tests are small test programs which verify a certain part of a production software, the test programs can be executed by a computer. By employing a unit test framework such as NUnit, test programs can be written in a way which makes it possible to automatically check whether tests succeeded or failed and if they failed, which part of the software malfunctioned [76].

According to a case study performed at Microsoft Corporation Inc. [107] by comparing Version 1 of a product, which was developed with ad hoc and individualized unit testing practices, to Version 2, which was developed with the utilization of NUnit automated testing framework by all team members, test defects decreased by 20.9% while development time increased by 30%. Version 2 also had a relative decrease in defects found by customers during first two years of software use.

Unit testing relates to portability in several ways. Firstly unit tests can be used to learn and verify how external libraries behave. Secondly unit testing can be used verify that basic operations such as adding and subtracting work correctly despite the change in the execution environment.

Unit testing external libraries which may be already thoroughly tested by their development teams might sound redundant at first but it offers several advantages. When a new external library is being investigated for possible inclusion, writing unit tests against its API is a more systematic approach to learning how to use the library compared to simple ad hoc testing. These unit tests can later be used when the library is updated to a new version to verify that the API still works as expected. Without these kinds of unit tests, small changes in the external library may propagate bugs in a very surprising ways.

When a software is ported to another architecture, there may differences in the basic operations. For example, if a software is ported from 64-bit architecture to a 32-bit architecture, mathematical operations with large integers or other number types may overflow. When an integer overflow happens, it may change from architecture to architecture what value will be returned as a result.

4.3 Integration Testing

Integration testing is performed to verify the correct integration of different components. Integration testing ensures that passing of data between components works as expected and that the components work in cohesion. Components may be integrated on multiple levels in hierarchy of the software. An example of integration between components is presented in figure 4.2. Also the integration testing can be done on

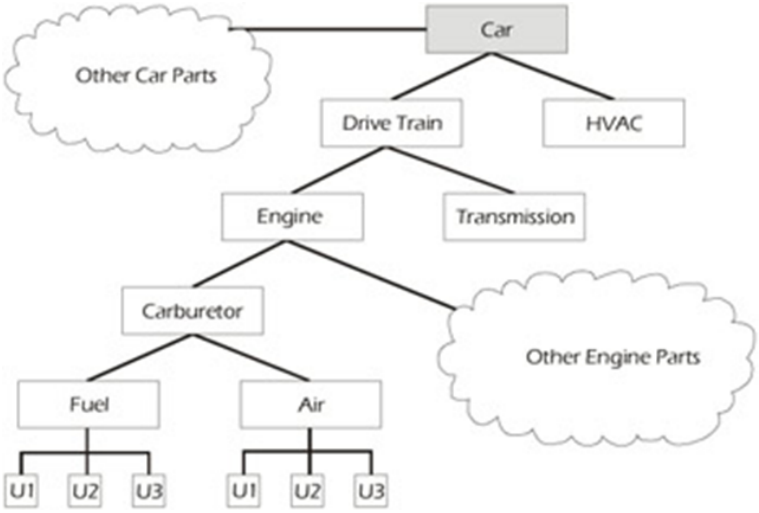


Figure 4.2: "Levels of Integration in a Typical Car" [22]

multiple levels. At the lowest level small components are integrated together. This level of testing is usually performed by the development team. At a higher level the integrated components are much larger and thus may require more testing resources. This level of testing may be done by the developers but it may also be performed by a dedicated test team [100].

An example case of proper unit testing but inadequate integration testing is the story of Mars Climate Orbiter in 1999 [74]. Mars Climate Orbiter was designed to gather data from Martian weather and to be a communications relay for Mars Polar Lander. When Mars Climate Orbiter was entering the Martian atmosphere communications to it were lost. NASA performed an investigation into the reasons of this mishap and located the problem in the communication between two components responsible for certain calculations relating to the atmosphere entry. Both components worked when they had been tested in isolation but when they were integrated the calculations were incorrect. The reason of this incorrectness was that another component used Imperial units and the other used metric units.

Integration testing at its nature is similar to unit testing except that whereas unit testing is concerned with testing a small unit of the software in isolation, integration testing tests the integration of different components. Thus automatic integration tests can usually be implemented with the same tools as automatic unit tests.

Portability requirements may present more requirements for the testing process. Different components of the system may be used on different platforms. Integration of the components on the similar platforms (such as on 64-bit architecture) may perform

as expected but when one of the components is run on different platform (such as on 32-bit architecture), the functionality may be altered. One way to avoid these types issues is to have comprehensive unit tests which are performed on all supported platforms. If the unit test results are equal on all platforms, integrating such a component running on different platforms should work as expected. However, even a unit testing with comprehensive code coverage may not be able to detect subtle differences such as slightly different timing which leads to problems only on specific platforms. Thus if resources permit, performing integration tests on all possible combinations of the integrated components would increase the detection of unexpected problems.

4.4 System Testing

The target of system testing is the entire system in a fully integrated state. The point of view of system testing varies based on the type of the project: for example installation and usability testing are performed from the point of view of a customer but some tests can verify behavior which might go unnoticed by the user but are very important for the correct performance of the system. System testing is usually performed by an independent test organization if one is available [13].

Because system testing should be performed from the perspective of the user, it is inadequate to just perform tests against internal APIs and to verify their functionality. If the software under testing is has some kind of user interface, the testing must also test the said user interface. There are existing testing tools for testing web UIs [88] and desktop UIs [63][77].

To encourage developers to use automated system testing, it is recommended to make test design and coding as easy as possible. If developing automated tests is more of a burden than testing the feature manually, there would no point in using automated tests. Automated system tests can yield some unexpected benefits: if system testing requires hardware, performing automated system tests outside of typical office hours allows the hardware to be free for other use during the office hours.

As system testing is concerned with the overall performance of the software, it is important to perform system tests on all supported platforms. Catching some portability problems such as disparities between different database software might be hard to catch on lower levels.

4.5 Acceptance Testing

Whereas unit, integration and system testing have been concerned about looking for problems, acceptance testing is used to make sure that the software fulfills the requirements. When software development has been contracted, successful acceptance tests are usually required before the customer accepts the product. However, this doesn't mean that acceptance tests would not be used when software is not developed under contract. Acceptance tests should be performed in an environment as close to the production environment as possible. The focus of the tests is usually on typical user scenarios instead of corner cases [13].

People interested in the acceptance test plan and the results of test can include many individuals from developers to business people. All interested parties may not be technically oriented so acceptance test plan should be non-technical [22].

Test-driven development and unit testing are widely used and researched methods in agile software development. Automated acceptance tests (AAT) is a more recent arrival and compliments the previous practices. Manually performing acceptance tests can be tedious, expensive and time consuming and thus is not suitably for agile software development processes. In AAT, acceptance requirements are captured in a format which can be automatically and repeatedly executed [40].

Selenium [88] is an example of such tool. Selenium can be used to automate actions which user performs with a web browsers. Though it can be used for other purposes than just acceptance tests, it can be used for them as well. Portability testing of multiple browsers can be done by executing Selenium test suites with multiple browsers.

However, Selenium tests are generally quite technical and therefore usually difficult to present to the customer as is. Customers can rarely read or especially write the code required for Selenium tests. FitNesse [30] and RSpec [85] are example tools which attempt to make the automated tests more readable to the customer.

RSpec is designed for Ruby programming language and is intended as a Behaviour-Driven Development tool. Behaviour-Driven Development combines Test-Driven Development, Domain Driven Design, and Acceptance Test-Driven Planning. [85] For example, in a banking application one acceptance criteria might be that a new bank account should contain 0 dollars. This test case can be written as an RSpec test detailed in figure 4.3 [17].

FitNesse uses a wiki for creating and maintaing test cases. Pages in the wiki contain test cases which are associated with tables of different inputs and expected outputs. Test tables are then parsed and used to perform tests against the software. When tests are performed, the successful test values are colored green and unsuccessful tests are

```
describe "A new account" do
    it "should have a balance of 0" do
        account = Account.new
        account.balance.should == Money.new(0, :USD)
    end
end
```

Figure 4.3: An example of RSpec test case.

colored red. [30]

eg.Division		
numerator	denominator	quotient?
10	2	5.0
12.6	3	4.2
22	7	=3.14
9	3	<5
11	2	4<_<6
100	4	33

Table 4.1: An example of a FitNesse test table. [30]

Based on case studies performed by Haugset et al. [41] developers felt that by having automated acceptance tests it was safer to make changes in code, fewer errors were found and the need for manual testing reduced. In addition to these there were some higher-level benefits such as it was easier to share competence within team, acceptance tests gave a better overview of the whole project and writing acceptance tests made developers think of what they are doing before doing this. Developers also felt that traditional specifications received from customers were poor in quality and writing acceptance tests based on them was in a way a reflection of what the customer really needed.

By executing automated acceptance tests to ensure portability, resources can be freed from performing same repetitive tests on multiple platforms. These resources can used to for example perform additional tests which may not possible to automate or to lower costs of the overall software project.

4.6 Chapter Summary

There is no single universally accepted definition of testing levels. Different software projects may require some testing levels which are unnecessary in others. By comparing different definitions and emphasizing the testing levels used at Sysdrone, unit, integration, system and acceptance testing levels were chosen for more detailed inspection. Unit testing is concerned with the operation of smallest independent unit in software. Integration testing is used to verify that the integration of two independently correctly working components is working as expected. System testing tests the software as a whole with a lot of technical and detailed test cases. Acceptance testing is usually concerned with the user requirements and thus its test cases are smaller in number and of less technical nature.

5 Test Techniques

In this chapter different test techniques are described and their relation to automatic testing and portability testing are investigated. The list of of test techniques is lifted from SWEBOK [44] but other literature and academic sources have been used to supplement the descriptions.

5.1 Alpha and Beta Testing

The usage of terms "alpha testing" and "beta testing" can vary significantly between different development teams. Most commonly they are defined as follows [22]: Alpha testing is an acceptance test performed in a development environment and hopefully with real users in a realistic environment. Beta testing is an acceptance which is performed in a production environment with real users. Many companies just release a beta version of the software for a group of users and let them have a go at it. This kind of ad hoc testing is not as precise as performing beta test with planned test cases and expected results.

Acceptance tests are sometimes divided into two categories [13]: alpha tests which are performed in-house and beta tests which are done by real customers. These tests can be used to test whether the product is ready for market but they can also be useful in finding bugs which were not found in the usual system testing process.

Alpha and beta testing performed by real users with usage monitoring systems (external or built-in into the tested software) can provide valuable insight into how users really use the software or which functions are the most commonly used.

Game called Starcraft II is one example of how beta testing can be applied to solving complex problems [14]. Starcraft II is a real-time strategy game with three different races which all have different kinds of units. This leads to a tricky balancing between different units and races so that no race or unit is unfairly dominating in gameplay. Finding the right kind of balance by only using internal testers can be very difficult but by releasing a beta version of the game to public the development team can gather massive amounts of data by embedding statistics tools and by asking opinions of the gaming community and pro players. These can be used to measure for example which units are used most or what are the winning ratios between three races.

Alpha and beta testing are especially used to get real users for the software by

releasing alpha and beta versions. Real user feedback cannot be obtained by any automatic testing system. Of course, alpha and beta versions can be subjected to other tests but usually if any test suites have been developed for the particular software, it should already pass those tests completely or at least partially with an expected level of errors before an alpha or beta version is released to the users.

Alpha and beta testing can be a good source of feedback about the portability of a software. Developers may not have access to the same array of devices that the users have or may not have enough resources to perform testing with that many devices. For example, testing an Android mobile software can be extremely difficult even if money is not a problem due to the sheer amount of different devices with varying hardware and software capabilities.

5.2 Configuration Testing

SWEBOK [44] defines configuration testing as follows: "In cases where software is built to serve different users, configuration testing analyzes the software under the various specified configurations." On the contrary, Kaner et al. [54] definition is: "The goal of configuration test is finding a hardware combination that should be, but is not, compatible with the program." Based on SWEBOK definition, portability testing and configuration testing are the same thing, but Kaner et al. definition places configuration testing as a subset of portability testing as portability testing is concerned with a more general portability from one environment to another. These environments may be identical in hardware aspect but differ on the software level. Because of these reasons, configuration testing is not researched in this particular subsection as it is researched overall in this thesis.

5.3 Conformance/Functional/Correctness Testing

SWEBOK [44] defines conformance testing as follows: "Conformance testing is aimed at validating whether or not the observed behavior of the tested software conforms to its specifications." Specification which is used to perform conformance testing may also be a specific standard [102].

Requirements detailed in a specification can be used as a basis when developing automatic test suite although some requirements may not suitable for automatic testing.

Software may behave differently on other platforms. Thus a conformance testing should be performed on all supported platforms.

5.4 Graphical User Interface Testing

Graphical user interface testing is hard because it involves testing both the underlying system and the user interface implementation. User interfaces have a large number of possible interactions. It is not just sufficient to test each view but also to test each sequence of commands which lead to that view. It is also very hard to determine how much of the user interface is covered by each test set. Determining test coverage of conventional systems is done by analyzing the amount and type of code in the software. This does not apply to user interface testing because the most important metric is how many different possible states of the system are tested. Each step of the user interface test has to be verified because if a problem has been encountered, it may not be possible to proceed to the next step. [64] Due to the amount of possible permutations, automation of graphical user interface testing is paramount. Manually testing each permutation and keeping track of which permutations have already been tested is very labor-intensive even with a small amount of permutations.

Testing of graphical user interfaces is not limited to a single level. The testing can be performed on all levels from a unit test level by developer to an acceptance test level by end-user.

User interface consists GUI objects which can be anything from windows and buttons to menus and other types of widgets. Even a simple interface typically contains multiple GUI objects. These objects have a lot of different behaviors from responding to a click to disabling an input field. Testing these behaviors can be very complicated and the resulting test code is usually messy. Some of these behaviors may even be impossible to test. [39] Unit testing of GUI objects can be made a lot easier by not putting code in the actual GUI object class but to create another class, a smart test class, first and the actual GUI object class last. GUI object doesn't any logic in itself but it is used just to display and retrieve data to and from user. This kind GUI object can be represented by a mock object during unit tests so developer can verify that smart class retrieved and displayed data correctly. [28]

Selenium [43] is a software used for in-browser testing. Selenium offers the possibility for writing web software in a test-first manner. These tests can be later used as a regression test suite. Selenium can be used on a system testing and acceptance testing levels.

There are also test software for taking automated screenshots of web pages on multiple platforms. This kind of testing is also offered as a service: one such example can found at http://browsershots.org/. This will be explored more in depth in section 7.2.

5.5 Installation Testing

SWEBOK [44] defines installation testing as follows: "Usually after completion of software and acceptance testing, the software can be verified upon installation in the target environment. Installation testing can be viewed as system testing conducted once again according to hardware configuration requirements. Installation procedures may also be verified."

It is possible to perform a installation without user interaction at least on Windows [65], Mac OS X [2] and various other UNIX-based [27][6][33] platforms. These types of installers can be used automatically and a successful installation can be verified in multiple ways depending on the installed software.

Installation on different platforms requires testing because for example installation on Windows and Linux operating systems are very different processes. This can be performed automatically with the help of virtual machines. [9]

5.6 Penetration Testing

The purpose of penetration testing [97] is to test security of the software against real-world attacks in a safe way. The penetration testers provide the software developers and managers data about the possible security problems before real attackers can exploit these problems. Penetration testing is usually performed on a real system and staff. Penetration testing may even involve physical attacks such as breaching physical security controls, stealing equipment or disrupting communications. Many penetration testers use a combination of vulnerabilities in one or many systems to gain improper access.

Most common vulnerabilities used in penetration testing can be found from these categories [97]: Misconfigurations such as insecure default usernames and passwords. Flaws in the kernel which is the heart of the operating system and thus affect all programs which are executed within it. Buffer overflows where an intruder can input a too large piece of data in memory and thus inject malicious data. Insufficient input validation such as when an attacker employs SQL injection by entering SQL commands in text inputs. Symbolic links where an attacker can point the link to parts of file system which would otherwise be inaccessible to the attacker. File descriptor attacks where type of file can be used in malicious ways by entering an invalid file descriptor. Race conditions where attacker can hijack a program when the said program is running in with elevated privileges. Incorrect file and directory permissions where attacker can gain access to files which should not be made available.

Some of these categories - such as misconfigured security settings - may not be directly related to the developed software. For example an UNIX server which is configured to allow root user to login over SSH or even telnet connection and a poor choice of root user password can be combined to breach the system. Root privileges allow the attacker to perform a variety of different attacks such as reading files which can not be accessed through the developed software or overwriting certain parts of the memory to fool the developed software to behave incorrectly.

However categories such as insufficient input validation are a direct responsibility of the developed software. For example database components should have proper unit tests for detecting SQL injections and when developing web software, HTML form components should be tested against cross-site scripting attacks where attacker inputs malformed data in a text input and manages to execute external JavaScript code.

Software may use security features of the underlying system. Different systems may have different implementations of these security features. Thus penetration testing should be performed on all supported platforms to ensure the safe operation of software on each of them.

5.7 Performance and Stress Testing

Software problems are often not caused by a deficiency in program logic but because the software is executed under a kind of stress which was not anticipated or not properly tested for during the development. Purpose of stress testing is to execute software in a situation where expected resources are not available [18]. These kinds of situations may be such as performing time-critical operations with CPU usage at full capacity or losing network connection in the middle of a database transaction.

SWEBOK [44] defines performance testing as follows: "This is specifically aimed at verifying that the software meets the specified performance requirements, for instance, capacity and response time. A specific kind of performance testing is volume testing, in which internal program or system limitations are tried." In comparison, stress testing is defined as follows: "Stress testing exercises software at the maximum design load, as well as beyond it."

Stress testing is often used to mean an array of different tests such as load testing, mean time between failure testing, low-resource testing, capacity testing and repetition testing. Stress testing itself is used to simulate a load greater than expected to expose bugs under stressful conditions. Load testing instead is used to test software under typical peak or higher loads. Mean time between failure measures average amount of successful operation before an error or crash occurs. Low-resource testing executes

program in an environment with low or depleted resources such as hard disk space or physical memory. Capacity testing is used to measure the maximum amount of users a system can withstand. Repetition testing is used to perform a test suite repeatedly to detect hard-to-find problems such as tiny memory leaks which become a big problem after a huge number of repetitions. [80]

Stress testing can be automated. Automated stress testing tools enable test engineer to instruct which tests should be performed and how many users should simulated during the test. Stress tests typically attempt to mimic normal user behavior but this kind of behavior can be multiplied with the automated stress testing tool. These kinds of tools typically produce a log file which can be used to assess the performance during test. Typical types of errors found in stress tests include memory leaks, problems with performance, locking or concurrency, excess consumption of system resources, and exhaustion of disk space. [25]

Environment can affect software performance. For example, different browsers provide different kinds of JavaScript performance, hardware architecture (such as 32-bit compared to 64-bit) can affect memory requirements and typically newer operating systems have more optimized system functions. Thus it is important to perform performance and stress tests on all supported platforms.

5.8 Recovery Testing

Many systems are required to have some degree of fault-tolerance. By being fault-tolerant, system has to be able to recover from certain hardware or network faults, software or human errors or loss of data without causing the system to cease operating or crash. During recovery testing, a system is intentionally injected with some type of system failure to verify that the system recovery is properly performed. Recovery process may or may not be automated. In case of automatic recovery process, a test should validate that the recovery was properly performed. [26] Specific objectives of recovery testing include the following [81]:

- "Adequate backup data is preserved.
- Backup data is stored in a secure location.
- Recovery procedures are documented.
- Recovery personnel have been assigned and trained.
- Recovery tools have been developed."

Some aspects of recovery testing - such as assigning and training recovery personnel - cannot be used in automatic testing process. However, certain fault scenarios can be triggered automatically. For example, if a software recording data from medical devices loses a database connection, it may be required to be able store medical data for a certain time period in a secondary storage location. This type of scenario can be used in a automated system or acceptance test where a database is programmatically closed or its network traffic blocked and then programmatically restored after the specified timespan. Automatic test can then verify that all produced data can be found at the database after the test has been concluded.

Different environments of software may affect the recovery procedures so recovery testing should also be performed on all supported platforms.

5.9 Regression Testing or Back-to-Back Testing

Regression testing and back-to-back testing are somewhat related. Back-to-back testing involves two different versions of the software which are ran through the same set of tests and their test results are compared. Regression testing is a way of testing subsequent versions of software against the same set of tests and verifying that all the previously passed test cases are still passed. [44] Majority of all testing effort in commercial software development is regression testing. Regression testing is very important because even small changes to one part of the system may cause unwanted changes in a completely different part of the software. [1]

Regression tests must be automated. If regression tests are not automated, it is equivalent to no regression testing. Large components or systems usually have large regression test suites. [1] Performing the equal amount of testing manually would be a huge waste of resources because these test suites must absolutely be performed on each software release. Performing all tests manually may take several weeks with a small test team. So for example if a software project has a release cycle of two weeks, test team may not even have enough time to perform all the previous regression tests.

As small changes may cause unexpected changes, making a seemingly innocuous change to code and testing it on one specific platform may introduce a bug on another platform. Because of this, regression tests must not only be automated but they must be performed on all supported platforms to ensure the correct behavior of the software.

5.10 Reliability Achievement and Evaluation

Software reliability engineering is concerned with reliability which is defined as "the probability of failure-free software operation for a specified period of time in a specified environment". Reliability is one of the attributes of software quality. Other attributes are functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation. Software relibility engineering includes measuring reliability, identifying attributes and metrics which affect reliability and applying these to increase reliability of the software. Reliability can be affected by multiple factors such as product design, development process, system architecture and software operational environment. [58]

Software reliability engineering can be divided into six steps. First it is important to specify how software is used and what kind of environmental conditions affect that. Secondly *quality* should be defined in cooperation with the customer. Quality of course includes reliability but also includes other factors such as delivery date or costs. Thirdly quality definition and product usage data can be used to design and implement the product. Fourthly reliability of third party software should be measured and included in the acceptance requirements. Fifthly reliability should be tracked during testing and this information can be used when deciding on releasing the product. And lastly reliability should be monitored in real world conditions and its results should be used to guide the product development further.

A case study performed at AT&T researched the application of software reliability engineering. Software engineers defined an *operational profile* based on customer modeling. This operational profile was used in automatically generating test cases. Software underwent a system test done in increments and clean-room development techniques together with feature testing based on the *operational profile*. Testing was done to reliability objectives. The quality improved dramatically: a factor-of-10 reduction in customer-reported problems, a factor-of-10 reduction in program maintenance costs, a factor-of-2 reduction in the system test interval and a 30 percent reduction in new product introduction interval. [58]

Besides using software reliability engineering methods to automatically generate test cases based on a real-world operational profile, software reliability can also be included in automatic testing process by adding a measurement of reliability rates in the automatic test cases.

Different environments, both hardware and software, may have different reliability rates. These reliability rates reflect on the reliability rate of the software under testing. Due to this, software reliability engineering should be performed on all supported

platforms.

5.11 Test-Driven Development

Test-driven development [106] is a software development practice. Earliest references to TDD are from as early as 60s when it was used at NASA. TDD is a critical part in Extreme Programming where code implementation is not preceded by any formal designs.

Test-driven development or in its commonly abbreviated form TDD is a software development process. Robert C. Martin defines the laws of process as follows [62]:

- "First Law: You may not write production code until you have written a failing unit test.
- Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- Third Law: You may not write more production code than is sufficient to pass the currently failing test."

These laws should be performed in a tight loop which is about two minutes long and almost always ends in success. Based on Robert Martin's experience [60] by employing TDD 90% of the production will be covered by unit tests. A major benefit of TDD is that if the code is not covered by tests, developers are afraid to change it because they might break it. If TDD is employed, any change may be tested against the existing test suite and if no tests break, it is unlikely that anything was broken. Unit tests can also be used as a documentation of how the software works.

Traditionally TDD is focused on unit tests but variations of TDD such as storytest-driven development (STDD) and acceptance test-driven development (ATDD) have also emerged. Both STDD and ATDD rely on automated test frameworks such as Fit, FitNesse and FitLibrary [42].

Automatic tests created by using test-driven practices can and should be performed on all supported platforms.

5.12 Usability Testing

Usability testing [86] is "a process that employs people as testing participants who are representative of the target audience to evaluate the degree to which a product meets

specific usability criteria". According to this definition, automation of usability testing is impossible. There are however services such UserTesting.com which help the user testing process.

UserTesting.com [99] provides a crowdsourcing service that provides a selection of test users from different demographics. The user of the service can watch as the test users use the software and provide commentary about their intents and frustrations. Unfortunately these results still require manual evaluation.

5.13 Chapter Summary

Alpha and beta testing is a valuable tool to get information from a lot of users. Due to its nature, it cannot be automated but by using diagnostics tools, a lot of data can be obtained from the real-world usage and possible problems related to portability.

Configuration testing is not precisely defined but most of its definitions relate to testing software in different configurations. This is similar to the portability testing but most definitions focus only on the hardware configurations.

Requirements detailed in a specification can be used as a basis when developing automatic test suite although some requirements may not suitable for automatic testing.

Due to the amount of possible permutations, automation of graphical user interface testing is paramount. Testing of graphical user interfaces is not limited to a single level. There are design patterns and 3rd party tools which help developers to create automatic test suites for graphical user interfaces.

Installation is particularly dependent on the underlying environment. Its testing can be automated. Virtual machines are useful when creating a testing environment for installations.

The purpose of penetration testing is to test security of the software against real-world attacks in a safe way. Some of the most common vulnerabilities encountered in penetration testing are misconfigurations, kernel flaws, buffer overflows, insufficient input validation, symbolic links, file descriptor attacks, race conditions and, incorrect file and directory permissions. Some aspects of the penetration are not directly related to the software development and some are concerned with the actual humans interacting with the system. These cannot be automated but there are many aspects which can be. Environment also can affect the safety of the software so penetration testing should be done on all supported platforms.

Purpose of stress testing is to execute software in a situation where expected resources are not available. Stress testing is often used to mean an array of different tests such as load testing, mean time between failure testing, low-resource testing, capacity

testing and repetition testing. Stress testing can be automated. Environment can affect software performance and thus stress testing should be performed on all supported platforms.

A system is injected with some type of failure recovery testing. To succeed the tests, the system must recover from the failure. Recovery testing involves educational and management work which cannot be integrated into automatic tests but some failure injections and recovery verifications can be automated.

In back-to-back testing two different versions are run through the same set of tests and their test results are compared. Regression testing is a way of testing subsequent versions of software against the same set of tests. Majority of all testing effort in commercial software development is regression testing. If regression tests are not automated, it is equivalent to no regression testing.

Software reliability is the probability of failure-free software operation for a specified period of time in a specified environment. A case study performed at AT&T suggests that software reliability testing can result in a factor-of-10 reduction in customer-reported problems, a factor-of-10 reduction in program maintenance costs, a factor-of-2 reduction in the system test interval and a 30 percent reduction in new product introduction interval.

Besides using software reliability engineering methods to automatically generate test cases based on a real-world operational profile, software reliability can also be included in automatic testing process by adding a measurement of reliability rates in the automatic test cases.

Different environments, both hardware and software, may have different reliability rates. These reliability rates reflect on the reliability rate of the software under testing. Due to this, software reliability engineering should be performed on all supported platforms.

Test-driven development is a tried and tested software development practice. By employing TDD, software will have a comprehensive test suite. Created tests are automatic.

Usability testing cannot be automated and thus falls outside the scope of this thesis.

6 Case Study: Sysdrone Oy

This chapter describes software development process, typical software project types and the requirements for validating and testing health technology software.

6.1 Software Development for Health Technology Industry

Sysdrone Oy is a software development company located in Jyväskylä, Finland and currently employs 14 people. Most of the staff are software designers and software architects.

Sysdrone doesn't have any internal product development: instead it provides software development teams and project management for software projects. Because of this, Sysdrone develops a lot of different kinds of software for different kinds of environments. The company specializes in health technology software. Most of the projects are developed with Microsoft's C# language but due to different kinds of customer needs, some projects have also been developed with languages such as C, Delphi Object Pascal, Perl and PHP.

Some of these software projects have been web or other kinds of server projects whereas in others the software has been installed on the computers of Sysdrone's client organizations or even directly on the computers of the clients of Sysdrone's client organizations.

Software testing is a pervasive requirement in software development at Sysdrone. Requirements for the correct performance of the software especially in the context of health technology and heavy industry software require comprehensive and systematic testing. Especially the software related to healthcare technology has some testing requirements specified by either local or EU laws.

Even though Sysdrone doesn't manufacture medical devices itself, both the Council of the European Communities and Food and Drug Administration of the United States of America include the related software as part of the medical device [32][21]. Thus Sysdrone doesn't manufacture any actual medical devices, it is still required to comply with all the requirements defined for actual medical devices.

There are many different standards which are used to evaluate medical devices and software. ISO 13485 is one such standard and is used in the European Union. ISO 13485 is based on ISO 9001 but places more emphasis on meeting regulatory and

customer requirements, risk management and effective processes. Unfortunately just ISO 13485 certification is not sufficient for the product to allowed for example in the United States of America: FDA has its own Quality System Regulation. However, ISO 13485 certification aligns management system of an organization to requirements of FDA's QSR and other such regulatory requirements in other countries. [104]

ISO 13485 is meant to improve consistency. Consistency helps to minimize errors. Well-documented processes, integrated quality controls and automated quality management systems lead to consistency. According to an article in Quality Magazine: "The standard is based on eight quality management principles: customer focus, leadership, involvement of people, process approach, system approach to management, continual improvement, fact-based decision-making and mutually beneficial supplier relationships." Failure to provide consistent functionality can lead to massive financial and personal damages. One such example is from 2010 when Medtronic paid 268 million U.S. dollars to settle lawsuits and claims related to their faulty defibrillation devices. The company estimated that 13 people may have died due to these failures. [105]

6.2 Current Testing Systems

Software development process at Sysdrone "is based on Scrum development process. Scrum is an iterative incremental process of software development commonly used with agile software development. Sysdrone uses slightly modified version of Scrum in order to fulfill the requirements of Medical devices directive." [94]

According to Sysdrone's software development process documentation software development has been divided into sprints. Sprint is an development iteration of 2 weeks or sometimes up to a month. Each sprint is started by having a meeting with a client and discussing which backlog items are selected from the product backlog for the sprint backlog. Product backlog is "a prioritized list of backlog items that are user requirements or risk mitigation requirements or bug fixes." Sprint backlog is "a list of backlog items to be completed during a single sprint." [94] Each backlog item contains a description of what kind of new feature or change to an existing feature should be done and what are the acceptance criteria for the user story. These acceptance criteria are used as a starting point for acceptance and validation testing plans which are written before the development has been started.

Process is divided into three phases: initial analysis, iterative development and release phase. Testing has been integrated into each of these phases. Test plans are written based on the task descriptions during the initial analysis. In the implementation

phase, programming tasks are implemented using test-driven development. If any of the newly created tests or previously implemented tests fails, it is number one priority to fix the tests. In the release phase acceptance tests are performed manually and the test steps and test results are written to test documents. [94]

When a development for a user story is started, the development is split into several tasks. Tasks should usually be the smallest possible meaningful steps to further the development. One example of a task could be adding an HTML form for adding a new patient to the system. When a task is completed, it is handed over to another developer who reads the program code or other similar result of the task and verifies that it is developed correctly. Code is further reviewed usually once per sprint in a more formal group code review session. According to Capers Jones, bug detection efficiency of informal code reviews ranges from 20% to 35% and efficiency of formal code inspections range from 45% to 70% [51].

All program code is developed by using test-driven development (as described in section 5.11). The resulting tests are also used in the peer review phase to verify that the software works as expected in another environment in addition to the development environment of the developer of the particular task. These tests are automatically added to the testing suite of the software.

Sysdrone uses continuous integration. According to Martin Fowler, "Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly." [31]

Automatic testing suite is divided into two testing processes:

First testing process is called instant build which is performed instantly after the continuous integration system detects a new commit to the version control system. Instant build only executes unit tests which can be performed quickly. If instant build would take a lot of time, developers couldn't get feedback from the broken tests immediately after committing new change to codebase. If a test fails, an email is sent to the developer and build monitor displays red error message which specifies the title and date of the breaking commit. Build monitors are located near the developers at the office. Build monitors can also be used by project managers or CEO to verify at a single glance that the software works as expected. Some tests which are not part of the instant build may also fail and developers won't get feedback from these immediately but it is a necessary compromise between smoothly flowing programming effort and

getting instant feedback from all tests.

Second testing process is called nightly build which is performed each night. Nightly build executes all possible tests related to the project. The same procedure is used in nightly build as in the instant build if a test fails. Build monitor and error emails indicate if the problem is with nightly or instant build tests.

Each step in the development cycle contains a *definition of done*. Definition of done is a list of requirements which have to satisfied before continuing to next step. One of the most important requirements is planning and performing tests. Here is an example of how definitions of done are specified for one of the projects at Sysdrone:

- Analyze phase is done when developers have written acceptance and validation tests plans for a user story.
- Develop phase is done when all implementation tasks are done and functionality can be demonstrated. Each programming task is developed with TDD and after the completion of a task, all unit tests are automatically performed. If an error is encountered, all development will be stopped until the error is analyzed and steps have been planned on how to solve the error.
- Verify phase is done when all tests are implemented and integration tests pass in development server.
- Deploy phase is done when all tests pass in a server which is as similar as possible to production server.

In addition to the unit tests created by developing with TDD, there are two types of tests: integration tests and validation tests. Validation tests are performed by the customer and are used to pass responsibility from Sysdrone to the client. Validation and integration tests are usually similar but whereas validation tests are quite simple and easy to perform even manually, integration tests are much more comprehensive.

There are two differences in terminology used at Sysdrone when compared to the more commonly used terminology:

- Whereas integration tests in the more strict definition test integration between two or more components of the software, integration tests at Sysdrone contain all tests between but not including unit test and acceptance test levels.
- Validation tests are similar to acceptance tests as detailed in this thesis but the chosen term *validation* tests was selected to emphasize the validation required in healthcare products.

6.3 Supported Software Types

Software developed at Sysdrone can be divided into three general categories:

- server software,
- desktop software and
- web-based software.

Server software is typically written in either C# language with ASP.NET framework or PHP language and executed on either Windows or Linux servers. Server software may or may not also provide a web interface.

Desktop software is typically written in C# language with .NET framework but may also be written in other languages such as Delphi Object Pascal and is mainly used on Windows operating system. .NET-based languages may be run with Mono framework on operating systems ranging from Android to Mac OS X and even on game consoles such as Nintendo Wii and Sony PlayStation 3 [71] but this cross-platform support hasn't been requested by current or previous customers.

Web-based software is typically written in C# language with ASP.NET framework or PHP. More recently client side of the software has been developed using HTML5. These kinds of software is usually accessed with a variety of different devices and operating systems ranging from Windows desktops to Android and iOS mobile devices. Web-based software may or may not interact with server software developed by Sysdrone.

6.4 Requirement Specifications

Requirement specifications for testing environment was composed based on discussions with software developers and especially project manager Ilkka Laitinen at Sysdrone.

Automatic portability testing system should be built using virtual machines. Because automatic portability testing suites are typically performed during nightly tests, having separate physical computers for each testing environment would be detrimental due to increased hardware, energy and physical space requirements. Sysdrone already uses project-specific virtual machines in development workstations so the company has previously acquired skills and knowledge related to building a system composed of virtual machines. This would also enable individual developers to install multiple different testing environments to their own development workstations for possible manual testing.

The automatic portability testing system should provide an easy way to choose desired testing environments for a specific project from a list of all possible different testing environments.

Reports from different automatic portability testing environments should preferably be provided as a single test report so developers or product manager don't have to access each of the selected testing environments manually to read the test reports.

The amount of testing environments should be kept relatively small because testing environments increase the amount of required computer maintenance (such as installing system updates), the amount of required server capabilities and in some cases the amount of licensing costs. In practice the costs of multiple testing environments and the benefits of higher defect-detection rate should be balanced. All newly added testing environments may not contribute significantly to the defect-detection rate and thus each testing environment should be carefully considered.

6.5 Chapter Summary

Sysdrone's software development process is based on Scrum with slight modifications to fulfill the requirements related to medical devices. Testing is tightly integrated into the software development process. Many of the tests are continuously and automatically performed in addition to a smaller number of manual acceptance tests. Currently tests are only performed on a single platform.

Software developed at Sysdrone can be roughly divided into three categories: server software, desktop software and web-based software. Testing environments for the software categories are describe in chapter 7.

7 Automatic Portability Testing Environments

This chapter includes plans for automatic portability testing environments for different kinds of software projects typically developed at Sysdrone.

7.1 Common Aspects

Even though portability testing increases the required testing effort by introducing many different platforms, fortunately most of this increased testing effort just requires more processing time from computers. If the required behavior is the same on all platforms, an existing comprehensive unit test suite for a single platform can be used when new required platforms are added. However, some platforms may require or utilize custom behavior which requires additional tests. An example of this could be a web site which has both desktop and mobile versions where the mobile version might not have all the features of the desktop version.

Sysdrone uses TeamCity as a continuous integration server. TeamCity supports feature called Build Grid. Build Grid consists of Build Agents which are computers with installed agent software capable of running builds. Build Agents can also run tests against the built software. Build Agents can be installed on multiple platforms with different pre-configured environments. Builds on multiple platforms can be executed simultaneously to speed up the build and testing process. Each build agent automatically checks out the most recent version from version control system so the maintenance requires less manual effort. [49] Build Agents could be run on actual physical computers but they can also be run in virtual machines which are abundantly available at Sysdrone.

7.2 Web-based Software Project

This section focuses on the part of the software project which is accessed by a web browser. Browser compatibility issues have been common for a long time. Some of the issues were detailed in section 3.4. In summary, to support 80% of the desktop users, developers would have to test the software with seven different versions of three major browsers. On mobile devices, the number of browsers to support 80% is five.

Sysdrone already employs Selenium as a system and acceptance testing tool for web-based software projects. However the tests are only run on a single platform. For existing projects, existing test suites can used to verify the functionality on new platforms in addition to the current testing platform.

Most of the major browsers such as Mozilla Firefox and Google Chrome are available on all major platforms. However some browsers such as Apple Safari are only available on certain platforms. In addition to this some browsers implementations between different platforms. For example, Apple Safari is available on Apple OS X and Microsoft Windows and the implementations are mostly same but there are some minor differences [90]. Some of these platforms can be installed on a virtual machine but some licensing issues may prohibit this on at least Apple OS X. OS X can legally be installed only on a Mac computer [8].

Selenium tests can also be executed on mobile platforms. Apple iOS and Google Android devices support remote access to their mobile browser. The browsers are controlled via RemoteWebDriver interface. This interface also allows tests to retrieve screenshots of the tested application. [87] Currently support is only available on Apple iOS and Google Android devices but the support is reportedly coming to Microsoft Windows Phone 7 and Research In Motion BlackBerry devices[47].

Screenshots can also be taken from the desktop browsers as well as mobile browsers. Screenshots are especially helpful in error situations but they can also be used to verify that not only the web application works as expected, it also is displayed correctly. Manually comparing screenshots can be labor-intensive in the long run so in the future it might be viable to investigate if an automatic comparison tool can be developed or integrated in to the automatic portability testing environment. This comparison tool could automatically compare images and trigger an alarm for example in a situation where the images of same view are less than 95% similar.

Selenium testing would be most comprehensive if all the tests were run on all combinations of available platforms and browsers. However due to time and money constraints many software projects have a limited selection of supported browsers. This should be accounted when building the automatic portability testing environment so that time is not spent on building support for platforms which are not required in any software projects at Sysdrone.

Selenium testing can be used to verify functionality of the software on acceptance and system testing levels. Testing all detailed functionality of the software project at these levels is labor-intensive (see section 4.2). Unit and integration testing levels are also relevant even on the client side (e.g. browser) of web-based software projects.

As JavaScript accounts for ever-increasing percentage of the codebase in web-based

projects, it too should be properly tested. There are many unit testing frameworks for JavaScript. One of the popular unit testing frameworks for JavaScript is Jasmine. Unit testing JavaScript is similar to unit testing software written in other languages. Because JavaScript support of the browsers differs between different browsers and their versions, unit testing should be performed on all supported browsers. To ensure that there is no regression in JavaScript code in the future versions of the software, JavaScript unit tests should also be integrated in the common build and testing processes. There are a variety of options[95][108] of how to integrate Jasmine tests to TeamCity.

Web-based software project usually also include server-side software. That part is discussed in section 7.4.

7.3 Desktop Software Project

Desktop software developed at Sysdrone is typically executed on Windows operating system. Because Sysdrone uses almost exclusively .NET desktop software development tools without Mono multiplatform extensions, this thesis will focus on the Windows portability issues.

The most apparent portability issue concerns different Windows versions. Microsoft current support lifecycles cover XP (until April 2014 [66]), Vista (until April 2017 [67]) and 7 (until January 2020 [68]). If the desktop software should support Windows Server versions as well, supported versions range from Windows Server 2003 to the latest one [69]. Due to customer requirements, some obsolete Windows versions might have to be supported as well. Further complexity arises from the availability of Service Pack updates. Most software should perform correctly even after Service Pack update but there are documented problems with at least some software and Service Pack update combinations [70]. The amount of versions and Service Pack updates leads to a lot of different combinations. Each new combination requires additional resources not only due to test suite execution time but due to the maintenance of test environments. Therefore it might not be ideal to perform all tests on all platforms but instead negotiate with the client about a list of supported versions. Unsupported versions can be periodically checked if customer wants information about whether the software *can* be used on certain platform even if that platform is not officially supported.

Some desktop software might be required for other desktop platforms. Depending on the nature of the required platforms and chosen programming languages and libraries, this might be relatively simple effort or require a lot of additional resources. In the case of Sysdrone, most of the software is developed on .NET platform. .NET software can be run on other operating systems beside Windows by using Mono framework

as discussed in section 6.3. As the automatic test suites for .NET software at Sysdrone are typically written in .NET languages (and more specifically, with NUnit framework) as well, they can be executed on all platforms supported by Mono [72]. Different programming languages and libraries might need platform-specific testing efforts but as currently there are no specific platforms and programming languages which should be supported at Sysdrone, discussing those platforms' and programming languages' needs would overtly speculative and currently unnecessary.

So far, all desktop software projects at Sysdrone have been for x86 hardware architecture. Recently, both 32- and 64-bit hardware support has become a requirement. So far, all the automatic tests have been run on one hardware architecture 32-bit and 64-bit hardware architecture has been tested manually and in a limited fashion. However, there haven't been any documented problems with this approach as .NET software can be compiled as platform agnostic [89]. If in the future a more comprehensive testing is desired, automatic test suites can be set up for execution on both 32- and 64-bit virtual or physical machines without any added human effort. Of course, computer resource needs will be increased.

7.4 Server Software Project

Server software at Sysdrone is typically written either for .NET platform or in PHP. .NET framework and its portability and testability were detailed in section 7.3. PHP is available on a wide selection of operating systems [82]. Test suites for PHP software are typically written also in PHP at Sysdrone so all the test suites can be executed on all the platforms that the actual software can be run. Selenium tests detailed in section 7.2 are used in most acceptance tests but for example REST APIs can be tested simply with PHPUnit without using Selenium.

As described in section 7.3, Windows operating systems can be updated with service packs which may introduce portability issues. These should be tested for with different service pack versions. Server software developed at Sysdrone is also developed for Linux servers. Linux is available in a wide array of different distributions such as Red Hat or Ubuntu. Although same software is typically available on different distributions, some of core software can differ. These can be tested as well with different virtual machines running on different distributions.

Some server software can also be developed for both Windows and Linux operating systems. In case the software is portable, the test suites are typically written portable as well. The same test suites can be then tested on different virtual machines with different operating systems. If there are some platform-specific test suites, it is possible

to specify these to particular virtual machines. Test results can be combined as a one report to ease the verification process.

The possible issues with 32- and 64-bit implementations are similar to the ones encountered with desktop software projects. For more details, refer to section 7.3.

7.5 Chapter Summary

Performing automatic test suites on multiple platforms would not require additional test planning or writing as the functionality is typically the same on all supported platforms. Different platforms can be added to TeamCity's agent pool so all the required platforms can be tested automatically.

Web-based software projects can and should be tested on unit, integration, system and acceptance test levels. Tools such as Jasmine can be used on unit and integration levels. Tools such as Selenium can be used on system and acceptance test levels. Tests should be performed on multiple browsers on different platforms to comprehensively verify the functionality of the software.

Desktop software projects typically use .NET framework for both the actual software and test suites. Thus the existing test suites can be executed on all platforms where the software will be executed. Mono framework can provide support for other platforms beside Windows. Virtual machines can be used to verify the functionality on different hardware architecture such as between 32- and 64-bit computers.

In a lot of cases the server software developed at Sysdrone are similar to desktop software projects. However desktop software is not typically developed for Linux operating systems whereas server software can run on both Windows or Linux or just either one.

8 Summary

The most important question of this thesis was: "How can Sysdrone use automatic testing to detect and mitigate portability issues in its software development process?" At Sysdrone testing is very important because of the extensive requirements due to laws and standards related to medical devices. Testing is already performed extensively and is automated to a high degree. This is a very good situation for introducing portability testing.

There are a number of steps Sysdrone can do to enhance their testing processes:

Use TeamCity to control a number of virtual machines with different operating systems. These virtual machines have their own build agents which build the software and perform automatic test suites. As done already today, unit tests which are fast to execute can be run upon each commit and longer acceptance tests can be run nightly. The virtual machines can be used to perform both Windows and Linux test suites developed with C# and PHP languages used typically at Sysdrone.

One of the most usually encountered portability problems are differences between various browser vendors and their software versions. Sysdrone already uses Selenium to perform acceptance tests but these are currently only performed with Mozilla Firefox browser. These same test suites can performed on multiple different browsers with Selenium. These browsers also include mobile browsers on mobile platforms such as iOS and Android.

In addition to running existing Selenium acceptance tests with different browsers, Sysdrone would also benefit from unit testing JavaScript code. Acceptance tests are also important but testing various detailed functionality with acceptance tests is overly complicated and much slower. There are various JavaScript unit testing frameworks but one of the most popular is Jasmine. Unit tests written with these frameworks should also be executed on all supported platforms. Preferably they are executed upon each commit as are other unit tests.

First supporting question was: "What is automatic testing, why is it important and how it can be done?" Automatic testing is a well researched subject and there weren't any new discoveries made during the writing of this thesis. Fortunately automation of tests is already well understood at Sysdrone so there wasn't need to argue in favor of increasing the automation.

Second supporting question was: "What is portability, what kind of problems are

related to it and how can these issues be detected automatically?" Portability is a complex concept. A lot of the books and scientific articles related to portability were concerned about the hardware portability. Hardware portability has become less of problem recently as higher-level programming languages have become increasingly popular and they help abstract out the hardware differences. Many portability issues of today are of software nature. One of the most frequently encountered is the issue of browser incompatibility.

At the beginning of writing this thesis I expected that there would be a lot of research done into practical portability issues. Unfortunately finding such research was surprisingly difficult. However during the writing I found out that it isn't required that these portability issues are deeply understood in theory before they can be detected. Testing portability issues proved to be easier than I expected because the existing test suites can be simply executed on all supported platforms. This is good news also because it doesn't require that much of additional resources. After the automatic portability testing environments have been setup for the first time, they can be used in many different projects without a lot of setup overhead.

Typically manual acceptance tests are executed on at least most of the supported platforms. This however is insufficient because some portability issues happen at a very low level and these can be hard or at least labor-intensive to detect in acceptance tests. It is beneficial to perform all the available tests on all supported platforms to detect the problems at an early stage and fix them as quickly as possible.

In retrospective, the subject of this thesis was overly broad. If I were to start the thesis again, I would constraint the subject to a more limited area such as portability issues in web browsers. More limited subject would have allowed me to delve deeper into the issues. However even with this broad subject, writing the thesis was very educational. Even though I already had some experience of automated tests, my knowledge is now both broader and deeper.

9 References

[1] Ammann, P. & Offutt, J., 2008. *Introduction to software testing*, p. 5, 215. New York, New York: Cambridge University Press.

[2] Apple, Mac OS X Developer Library, *installer(8) Mac OS X Manual Page*, available at <URL: `https://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man8/installer.8.html`>, 10.2.2012.

[3] AppleInsider, *Apple confirms switch to Intel*, available at <URL: `http://www.appleinsider.com/articles/05/06/06/apple_confirms_switch_to_intel.html`>, 16.12.2011

[4] AppleInsider, *Apple unveils Mac mini Core Duo*, available at <URL: `http://www.appleinsider.com/articles/06/02/28/apple_unveils_mac_mini_core_duo.html`>, 16.12.2011.

[5] AppleInsider, *A closer look at Apple's Core 2 Duo MacBook Pro*, available at <URL: `http://www.appleinsider.com/articles/06/10/31/a_closer_look_at_apples_core_2_duo_macbook_pro.html`>, 16.12.2011.

[6] APT-RPM, *apt-get(8) - Linux man page*, available at <URL: `http://linux.die.net/man/8/apt-get`>, 10.2.2012.

[7] Ariel, A. & Berger, N.A., 2006. *Plotting the globe: stories of meridians, parallels, and the international*, p. 142. Westport, Connecticut: Praeger Publishers.

[8] Ars Technica, *Appeals court: Apple can continue to restrict OS X to Mac hardware*, available at <URL: `http://arstechnica.com/apple/2011/09/appeals-court-apple-can-continue-to-restrict-os-x-to-mac-hardware/`>, 27.05.2012.

[9] Attivio, Inc., *Attivio Technical Series: Automating Installation Testing*, available at <URL: `http://www.attivio.com/blog/56-java-development/223-technical-series-automating-installation-testing.html`>, 10.2.2012.

[10] Australian Strategic Policy Institute, *Is The JSF good enough?*, available at <URL: `http://www.aspi.org.au/publications/publication_details.aspx?ContentID=56&pubtype=6`>, 10.11.2011.

[11] Battlefield 3 Blog, *Battlefield 3 Xbox 360 sales top PlayStation 3 and PC combined*, available at <URL: http://bf3blog.com/2011/11/battlefield-3-xbox-360-sales-top-playstation-3-and-pc-combined/>, 5.1.2012.

[12] Binu, A., 2010. *Problem Solving and Computer Programming Using C*, p. 6-7. New Delhi, India: University Science Press.

[13] Black, R., 2002. *Managing the Test Process: Practical Tools for Managing Hardware and Software Testing*, p. 7-8. Wiley Publishing. ISBN 0-471-22398-0

[14] Blizzard Insider, *StarCraft II Beta Retrospective*, available at <URL: `http://us.battle.net/sc2/en/blog/39345`>, 9.12.2011.

[15] Boehm, B.W., 1981. *Software engineering economics*, p. 37. Upper Saddle River, New Jersey: Prentice Hall.

[16] Brown, P.J., 2003. *Encyclopedia of Computer Science, 4th Edition*, p. 1633-1634, available at <URL: `http://dl.acm.org/citation.cfm?id=1074100.1074809`>. Chichester, UK: John Wiley and Sons Ltd. ISBN: 0-470-86412-5

[17] Chelimsky, D., Astels, D., Dennis, Z., Hellesøy, A., Helmkamp, B. & North, D., 2010. *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*, p. 153. The Pragmatic Programmers.

[18] Chemuturi, M., 2010. *Mastering Software Quality Assurance: Best Practices, Tools and Technique for Software*, p. 273. J. Ross Publishing. ISBN-13: 978-1604270327

[19] CNN, *This Week At War - Feb 24, 2007*, available at <URL: `http://transcripts.cnn.com/TRANSCRIPTS/0702/24/tww.01.html`>, 10.11.2011.

[20] Committee for Advancing Software-Intensive Systems Producibility, National Research Council, *Critical Code - Software Producibility For Defense*, p. 87-91. Washington, District of Columbia: The National Academies Press.

[21] Council of the European Communities, *Council Directive 93/42/EEC of 14 June 1993 concerning medical devices*, available at <URL: `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31993L0042:En:HTML`>, 20.1.2012.

[22] Craig, R.D. & Jaskiel, S.P., 2002. *Systematic software testing*, p. 101. Norwood, Massachusetts: Artech House Publishers.

[23] da Cruz, F., *Columbia University Computing History: Programming the ENIAC*, available at <URL: http://www.columbia.edu/cu/computinghistory/eniac.html>, 30.12.2011.

[24] Dijkstra, E.W., *Notes on Structured Programming*, available at <URL: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>, 10.11.2011.

[25] Dustin, E., Rashka, J. & Paul, J., 1999. *Automated Software Testing: Introduction, Management, and Performance*, p. 40-41. Addison-Wesley Professional.

[26] Engel, A., 2010. *Verification, Validation and Testing of Engineered Systems (Wiley Series in Systems Engineering and Management)*, Chapter 5.4.3. Wiley-Blackwell. ISBN-13: 978-0470527511

[27] Ewing, M., Johnson, J. & Troan, E., *rpm(8) - Linux man page*, available at <URL: http://linux.die.net/man/8/rpm>, 10.2.2012.

[28] Feathers, M., *The Humble Dialog Box*, available at <URL: http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf>, 9.2.2012.

[29] Firtman, M., *Mobile HTML5 - compatibility tables for iPhone, Android, BlackBerry, Symbian, iPad and other mobile devices*, available at <URL: http://mobilehtml5.org/>, 16.12.2011.

[30] FitNesse, *Two Minute Example*, available at <URL: http://fitnesse.org/FitNesse.UserGuide.TwoMinuteExample>, 15.12.2011.

[31] Fowler, M., *Continuous Integration*, available at <URL: http://martinfowler.com/articles/continuousIntegration.html>, 20.6.2011.

[32] U.S. Food and Drug Administration, *Is The Product A Medical Device?*, available at <URL: http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/Overview/ClassifyYourDevice/ucm051512.htm>, 20.1.2012.

[33] The FreeBSD Project, *PKG_ADD(1) FreeBSD General Commands Manual*, available at <URL: http://www.freebsd.org/cgi/man.cgi?query=pkg_add&sektion=1>, 10.2.2012.

[34] Gabbrielli, M. & Martini, S., 2006. *Programming Languages: Principles and Paradigms*, p. 414, 417. Springer London Dordrech Heidelberg New York. ISBN: 978-1-84882-913-8.

[35] Garen, K., *Software Portability: Weighing Options, Making Choices*, available at <URL: http://www.nysscpa.org/cpajournal/2007/1107/perspectives/p10.htm>, 5.1.2012.

[36] Gore, A. *Presentation given at Nordic Business Forum in Jyväskylä, Finland*, 30.9.2011.

[37] Guardian, The, *Apple backtracks on iPhone sex ban*, available at <URL: http://www.guardian.co.uk/technology/blog/2009/may/24/iphone-ban-eucalyptus>, 16.6.2012.

[38] Hakuta, M. & Ohminami, M., *A Study of Software Portability Evaluation*, available at <URL: http://www.mendeley.com/research/study-software-portability-evaluation/>, 30.12.2011.

[39] Hamill, P., 2004. *Unit Test Frameworks*, p. 51-52. O'Reilly Media. ISBN-13: 978-0596006891.

[40] Hanssen, G.K. & Haugset, B., 2009. *Automated Acceptance Testing Using Fit*, FIND COMPLETE REFERENCE INFORMATION FROM IEEEXPLORE.

[41] Haugset, B. & Hanssen, G.K., 2008. *Automated Acceptance Testing: A Literature Review and an Industrial Case Study*, published in Agile '08 Conference on 4-8 Aug. 2008. Available at <URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4599450>. ISBN: 978-0-7695-3321-6

[42] Hayes, J.H., Dekhtyar, A. & Janzen, D.S., 2009. *Towards Traceable Test-Driven Development*, published in Traceability in Emerging Forms of Software Engineering ICSE Workshop on 18 May 2009. ISBN: 978-1-4244-3741-2

[43] Holmes, A. & Kellogg, M., 2006. *Automating Functional Tests Using Selenium* Published in Agile Conference, 2006. Print ISBN: 0-7695-2562-8. Available at <URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1667589>, 9.2.2012.

[44] IEEE Computer Society, *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, Chapter 5, available at <URL: http://www.computer.org/portal/web/swebok/html/ch5#Ref2>, 10.11.2011.

[45] IEEE Computer Society, 1990. *IEEE Standard Glossary of Software Engineering Terminology*. E-ISBN: 0-7381-0391-8. Available at <URL: http://ieeexplore.ieee.org/servlet/opac?punumber=2238>, 26.1.2012.

[46] Illing, G. & Peitz, M., 2006. *Industrial Organization and The Digital Economy*, p. 42. Cambridge, Massachusetts: MIT Press. ISBN-13: 978-0-262-09041-4.

[47] InfoWorld, *Selenium test suite to add mobile apps*, available at <URL: http://www.infoworld.com/d/application-development/selenium-test-suite-add-mobile-apps-011>, 20.5.2012.

[48] International Data Corporation, *Press Release: More Mobile Internet Users Than Wireline Users in the U.S. by 2015 - Sep 12th, 2011*, available at <URL: http://www.idc.com/getdoc.jsp?containerId=prUS23028711>, 16.12.2011.

[49] JetBrains, *TeamCity - Features*, available at <URL: http://www.jetbrains.com/teamcity/features/index.html#Build_Infrastructure>, 18.5.2012.

[50] Johnson, S.C. & Ritchie, D.M., *Portability of C Programs and the UNIX System*, available at <URL: http://cm.bell-labs.com/who/dmr/portpap.html>, 5.1.2012.

[51] Jones, C., 1996. *Software defect-removal efficiency*, p. 94. Published in Computer Volume 29 Issue 4 by IEEE Computer Society. ISSN: 0018-9162. Available at <URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=488361>, 26.1.2012.

[52] jQuery Mobile, *jQuery Mobile 1.0.1 Released*, available at <URL: http://jquerymobile.com/blog/2012/01/26/jquery-mobile-1-0-1-released/>, 16.6.2012.

[53] The Linux Foundation, *Linux Kernel Development*, available at <URL: http://www.linuxfoundation.org/docs/lf_linux_kernel_development_2010.pdf>, 3.11.2011.

[54] Kaner, C., Falk, J.L. & Nguyen, H.Q., 1999. *Testing Computer Software, Second Edition*. New York, New York: John Wiley & Sons, Inc. ISBN: 0471358460

[55] Kaner, C., *Exploratory Testing - Keynote at QAI November 17, 2006*, available at <URL: http://www.kaner.com/pdfs/ETatQAI.pdf>, 11.11.2011.

[56] Lockheed Martin, *F-22 Raptor*, available at <URL: http://www.lockheedmartin.com/products/f22/>, 10.11.2011.

[57] Louridas, P., 2006. *Static Code Analysis*, published in IEEE Software Volume 23 Issue 4. ISSN: 0740-7459. Available at <URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1657940>, 27.1.2012.

[58] Lyu, M.R., 1996. *Handbook of Software Reliability Engineering*, chapters 1 & 6. IEEE Computer Society Press & McGraw Hill. ISBN: 0-07-039400-8. Also made freely available by the author at <URL: `http://www.cse.cuhk.edu.hk/~lyu/book/reliability/`>, 17.2.2012.

[59] Maraia, V., 2005. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley Professional.

[60] Martin, R.C., 2007. *Professionalism and Test-Driven Development*, published in IEEE Software Volume 24 Issue 3. Available at <URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4163026`> ISSN: 0740-7459.

[61] Martin, R.C., 2007. *TDD with Acceptance Tests and Unit Tests*, available at <URL: `http://blog.objectmentor.com/articles/2007/10/17/tdd-with-acceptance-tests-and-unit-tests`>, 3.6.2012.

[62] Martin, R.C., 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*, p. 121-122. Boston, Massachusetts: Pearson Education Inc.

[63] McCaffrey, J., *Automating UI Tests In WPF Applications*, available at <URL: `http://msdn.microsoft.com/en-us/magazine/dd483216.aspx`>, 3.6.2012.

[64] Memon, A.M., Pollack, M.E. & Soffa, M.L., 1999. *Using a Goal-Driven Approach to Generate Test Cases for GUIs*, available at <URL: `http://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/test-gui.pdf`>, 9.12.2011.

[65] Microsoft, *Standard Installer Command-Line Options*, available at <URL: `http://msdn.microsoft.com/en-us/library/windows/desktop/aa372024(v=vs.85).aspx`>, 10.2.1012.

[66] Microsoft, *Microsoft Support Lifecycle*, available at <URL: `http://support.microsoft.com/lifecycle/?c2=1173`>, 3.6.2012.

[67] Microsoft, *Microsoft Support Lifecycle*, available at <URL: `http://support.microsoft.com/lifecycle/?c2=11732`>, 3.6.2012.

[68] Microsoft, *Microsoft Support Lifecycle*, available at <URL: `http://support.microsoft.com/lifecycle/?c2=14019`>, 3.6.2012.

[69] Microsoft, *Microsoft Support Lifecycle*, available at <URL: `http://support.microsoft.com/lifecycle/?c2=1163`>, 3.6.2012.

[70] Microsoft, *Some programs have compatibility issues with Service Pack 1 for Windows 7 and for Windows Server 2008 R2*, available at <URL: http://support.microsoft.com/kb/2492938/>, 3.6.2012.

[71] Mono Project, *Support Platforms*, available at <URL: http://www.mono-project.com/Supported_Platforms>, 26.1.2012.

[72] Mono Project, *NUnit*, available at <URL: http://www.mono-project.com/NUnit>, 10.6.2012.

[73] Myers, G.J., 2004. *The Art Of Software Testing, Second Edition*. Hoboken, New Jersey: John Wiley & Sons, Inc.

[74] National Aeronautics and Space Administration, 1999. *Mars Climate Orbiter Mishap Investigation Board Phase I Report*, p. 6 & 13, available at <URL: http://sunnyday.mit.edu/accidents/MCO_report.pdf>, 2.12.2011.

[75] National University of Singapore, *Assembly Examples*, available at <URL: http://staff.science.nus.edu.sg/~phywjs/CZ101/assembly-examples/programs.html>, 5.1.2012.

[76] NUnit, *NUnit Quick Start*, available at <URL: http://www.nunit.org/index.php?p=quickStart&r=2.6, 23.2.2012.

[77] NUnitForms, *NUnitForms - windows.forms unit testing*, available at <URL: http://nunitforms.sourceforge.net/>, 3.6.2012.

[78] Oglesby, D., Schloegel, K., Bhatt, D. & Engstrom, E., 2001. *A Pattern-based Framework to Address Abstraction, Reuse, and Cross-domain Aspects in Domain Specific Visual Languages*, available at <URL: http://w3.isis.vanderbilt.edu/OOPSLA2K1/Papers/Oglesby.pdf>, 23.12.2011.

[79] Osherove, R., 2009. *The Art Of Unit Testing with Examples in .NET*, p. 4. Greenwich, Connecticut: Manning Publications Co.

[80] Page, A., Johnston, K. & Rollison, B.J., 2008. *How We Test Software at Microsoft*. Microsoft Press. ISBN-13: 978-0735624252

[81] Perry, W.E., 2006. *Effective Methods for Software Testing, Third Edition*. Indianapolis, Indiana: Wiley Publishing, Inc. ISBN-13: 978-0-7645-9837-1

[82] PHP Group, The, *PHP: platforms*, available at <URL: https://wiki.php.net/platforms>, 11.6.2012.

[83] Pirsig, R.M., 1974. *Zen and the Art of Motorcycle Maintenance*, Chapter 20. New York City, New York: William Morrow and Company.

[84] RegHardware, *Apple bans Page 3 from iPhone app*, available at <URL: `http://www.reghardware.com/2009/05/05/the_sun_app_ban/`>, 16.6.2012.

[85] RSpec, *RSpec.info: Home*, available at <URL: `http://rspec.info`>, 15.12.2011.

[86] Rubin, J. & Chisnell, D, 2008. *Handbook of Usability Testing: How to Plan, Design and Conduct Effective Tests, 2nd Edition*, p. 21. Indianapolis, Indiana: Wiley Publishing Inc.

[87] Selenium, *WebDriver support for Mobile browsers*, available at <URL: `http://code.google.com/p/selenium/wiki/WebDriverForMobileBrowsers`>, 20.5.2012.

[88] SeleniumHQ, *Selenium - Web Browser Automation*, available at <URL: `http://seleniumhq.org/`>, 15.12.2011.

[89] Seth, G., *Moving from 32-bit to 64-bit application development on .NET Framework*, available at <URL: `http://blogs.msdn.com/b/gauravseth/archive/2006/03/07/545104.aspx`>, 10.6.2012.

[90] Stack Overflow, *stopping key event bubbling in safari 4 windows*, available at <URL: `http://stackoverflow.com/questions/1678273/stopping-key-event-bubbling-in-safari-4-windows`>, 27.05.2012.

[91] StatCounter, *Top 5 Browsers on Nov 2011*, available at <URL: `http://gs.statcounter.com/#browser-ww-monthly-201111-201111-bar`>, 16.12.2011.

[92] StatCounter, *Top 8 Mobile OSs from Jan to Dec 2011*, available at <URL: `http://gs.statcounter.com/#mobile_os-ww-monthly-201101-201112`>, 12.1.2012.

[93] Stern, N. 1981. *From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computer*. Bedford, Massachusetts: Digital Press.

[94] Sysdrone Oy, *Software Development Process description*, an internal document, 5.12.2011.

[95] Tar Pit, The, *Having Jasmine tests results in TeamCity via node.js (on windows) invoked from powershell*, available at <URL: http://blog.goneopen.com/2011/09/having-jasmine-tests-results-in-teamcity-via-node-js-on-windows-invoked-from-powershell/>, 18.5.2012.

[96] University of Manchester, *Early Electronic Computers*, available at <URL: http://www.computer50.org/mark1/contemporary.html>, 5.1.2012.

[97] U.S. Department of Commerce, National Institute of Standards & Technology, 2008. *Technical Guide to Information Security Testing and Assessment*, available at <URL: http://csrc.nist.gov/publications/nistpubs/800-115/SP800-115.pdf>, 2.12.2011.

[98] U.S. Department of Commerce, National Institute of Standards & Technology, 2002. *Planning Report 02-3: The Economic Impacts of Inadequate Infrastructure for Software Testing*, available at <URL: http://www.nist.gov/director/planning/upload/report02-3.pdf>, 17.11.2011.

[99] UserTesting.com, *UserTesting.com - Low Cost Usability Testing*, available at <URL: http://www.usertesting.com/>, 9.12.2011.

[100] Sestoft, P. 2008. *Systematic software testing, Version 2*, available at <URL: http://www.itu.dk/~sestoft/papers/softwaretesting.pdf>, 17.11.2011.

[101] United States Air Force, *FY 2011 Budget Estimates*, p. 15, available at <URL: http://www.saffm.hq.af.mil/shared/media/document/AFD-100128-072.pdf>, 10.11.2011.

[102] Wakid, S.A., Kuhn, D.R. & Wallace, D.R., 2002. *Toward Credible IT Testing and Certification*, published in IEEE Software Volume 16 Issue 4. ISSN: 0740-7459. Available at <URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=776947>, 3.2.2012.

[103] Whittaker, J.A., 2000. *What Is Software Testing? And Why Is It So Hard?*, published in IEEE Software Volume 17 Issue 1. ISSN: 0740-7459. Available at <URL: http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=819971>, 27.1.2012.

[104] Wichelecki, S., *Understanding ISO 13485*, published in Quality Magazine on January 2, 2008. Available at <URL: http://www.qualitymag.com/Articles/Feature_Article/BNP_GUID_9-5-2006_A_1000000000000225133>, 26.1.2012.

[105] Willett, N., 2011. *Quality Management: The Need for ISO 13485*, published in Quality Magazine on January 1, 2011. Available at <URL: http://www.qualitymag.com/Articles/Feature_Article/BNP_GUID_9-5-2006_A_1000000000000964284>, 18.6.2012.

[106] Williams, L., Maximilien E.M. & Vouk, M., 2003. *Test-driven Development as a Defect-Reduction Practice*, published in Software Reliability Engineering International Symposium on 17-20 November 2003. ISBN: 0-7695-2007-3

[107] Williams, L., Kudrjavets, G. & Nagappan, N., 2009. *On the Effectiveness of Unit Test Automation at Microsoft*, published in 20th International Symposium on Software Reliability Engineering on 16-19 November 2009. ISSN: 1071-9458, E-ISBN: 978-0-7695-3878-5, Print ISBN: 978-1-4244-5375-7. Available at <URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5362086`>, 23.2.2012.

[108] You Can't Write Perfect Software, *Running Jasmine Tests in a Continuous Integration Build (with TeamCity)*, available at <URL: `http://benshepheard.blogspot.com/2011/05/running-jasmine-tests-in-continuous.html`>, 18.5.2012.