Joel Helenius

Improving the quality of manual verification procedures

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technologies

Thesis

8 May 2018

Metropolia

Author Title Number of Pages Date	Joel Helenius Improving the quality of manual verification procedures 37 pages + 1 appendices 8 May 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technologies
Professional Major	Health Technology
Instructors	Sakari Lukkarinen, Senior Lecturer, Metropolia Tulasi Punaganti, Verification Lead

This thesis was done for a company that designs and produces patient monitoring devices. These devices are classified as medical devices, meaning that their design, implementation and documentation are bound by ISO standards and FDA regulations that demand for high quality and preciseness. For this thesis, the focus was on the documentation of instructed manual testing, manual verification procedures.

The purpose of this thesis was to come up with recommendable improvements for manual verification procedures that will reduce the amount of wasteful corrective actions stemming from erroneous documentation or execution of verification. The improvements that were arrived at include ways to make the procedures themselves more robust, but the more impactful recommendations concern practices that originally generated bad documentation.

A large portion of the work towards giving improvement recommendations was first getting accustomed with the various details related to software development and testing, requirements engineering and standards compliance. Once a reasonable amount of background knowledge was accumulated, a method for investigating the current process of verification procedure creation was developed and carried out. The most suitable way of gathering information was found to be the in-depth interviewing of professionals. The interview questions were based on the preliminary understanding of verification engineering gained from relevant literature and work experience as a manual tester.

The improvements proposed in this thesis were formed with a long-lasting impact in mind, with an acknowledgement of lean practices; the idea of getting effective results using few resources. While their implementation might show an increase of effort during the early phases of the workflow, the whole life cycle of verification procedures is estimated to be a lighter burden with an added increase in quality.

Keywords	Requirements engineering, verification, quality management, software testing, standards compliance

Metropolia

Contents

List of Abbreviations

VVT	Verification, validation and testing.
UUT	Unit under test.
FDA	Food & Drug Administration.
HSI	Human-system interface.
CASRE	Computer-Aided Systems Reliability Estimation
CPU	Central processing unit.
API	Application programming interface.
SE	Software engineering.
GHTF	Global Harmonization Task Force.
ISO	The International Organization for Standardization.

1 Introduction

Originally, the manufacturing industry was mainly focused on the design, development, production and maintenance of singular products, despite their complexity. Presently, by contrast, the scope of the manufacturing industry has grown to include services and solutions as well as products, that comprise of a variety of components, technologies, people and machines. These complex entities can be called engineered systems. Additionally, the time allotted to the introduction of new products and services into the market is decreasing steadily. For these reasons, the quality of processes and methods used in the manufacturing industry needs to be high. [1, p. 4.]

The development of a product can be hindered whenever a phase in it is working poorly. An often-disregarded phase in product development is the verification, validation and testing (VVT) of a system. This phase, a systems engineering tool, is used to ensure that the product or service in question is delivered with as few errors as possible, is functional and meets or exceeds the user's demands. The poor execution of the VVT process often results in increases in time and costs committed to the development of the product or service by over 20 percent. Any extra work stemming from this, however, must be carried out since faults found in the product by the end user likely harm the reputation of the system or even the reputation of the system's developer. [1, p- 4.5.]

This bachelor's thesis is done for a company that produces patient monitoring solutions for hospital use. The task is to come up with improvement ideas and solutions to problems that are currently hindering the verification process. A patient monitor is a complex medical device that requires extensive measures for quality assurance. The features are constantly tested during development, after which they are verified and validated as an indication of adherence to design and quality specifications. The manufacturer needs to provide comprehensive documentation regarding testing procedures and test results. Since there are hundreds of features and thousands of requirements that need to be tested, the documentation burden is substantial in such a degree that having multiple authors that all adhere to same documentation guidelines is inevitable. To properly assess and improve these guidelines, the author needs to be well-versed in both VVT and company practices. The author is a manual verification intern in the company with hundreds of hours of experience in system and software verification of one of the company's newest product lines in patient monitoring solutions. The improvement proposals shall

be given with this experience in mind, in addition to acquainting oneself with VVT, ISO/IEC/IEEE standards, requirements engineering and various other relevant topics through professional literature.

2 Requirements engineering

2.1 Presentation of requirements

In spoken language, the term ‘requirement’ can be defined as a desire someone has. In the case of systems development, a requirement is an attribute of the system being built, regardless of whether the attribute has been implemented or not. Requirements can also be negative in the sense that they can be restrictions to the system’s operation. [2, p. 45.]

In the ISO/IEC/IEEE 24765:2017 standard, a requirement is defined as:

> 1. statement that translates or expresses a need and its associated constraints and conditions; 2. condition or capability that must be met or possessed by a system, system component, product, or service to satisfy an agreement, standard, specification, or other formally imposed documents; 3. provision that contains criteria to be fulfilled; 4. a condition or capability that must be present in a product, service, or result to satisfy a contract or other formally imposed specification.

Requirements need to be represented in a specific way. Firstly, they must be easily understood, written in simple sentences without using professional jargon or abbreviations. Secondly, they must be unambiguous, not open for interpretation. They also need to be based on objective facts. [2. p. 46.]

In a sense, misinterpretations or errors in requirements could be all but avoided by writing them using formal logic and mathematics, but they need to be interpretable by all parties, engineers and non-engineers alike. Therefore, writing requirements is done using natural language. Since writing is an artform and not science, it needs to be regulated by guidelines and policies in technical settings. There is no way to ensure the quality of requirements, but it is a skill that can be learned by individuals just like any other. [2, p. 137.]

The basis of good requirement writing skills is grammatical proficiency. Spelling, punctuation and the general command of the used language are essential to any writing activity. In addition, the writing of requirements must adhere to the basic tenets of technical writing. The language needs to be clear and precise, objective, and without the use of figures of speech that could give rise to ambiguity. Finally, requirements must be written in a uniform style, following a standardized format. The formats may change, however, depending on the kind of requirements that are being written. For example, user requirements and system requirements are written differently to accommodate the purpose or intended audience of each requirement type. Some agile development methods use a format called user stories to write requirements. They are short descriptions of a system's functions, generally in the following format: "As a <type of user>, I want to <objective> for <reason>." These kinds of requirements are purposefully easy for the stakeholder in to specify and, in turn, for the developer to identify the source. [2, p. 139-140.]

2.2 Requirements engineering terminology

Requirements can be divided into groups in many ways. One such division relates to the perspective of the requirement. A user requirement, defined from the perspective of the user of the system, construes a functionality or restriction expected from the system for its users. These requirements come from the stakeholders and are usually represented using unrestricted natural language, communicating a specific problem that needs to be addressed. A system requirement, on the other hand, is defined from the perspective of the system itself and, while ultimately stemming from user requirements, are defined independently to present engineers with desired solutions to be implemented into the system. [2, p. 57-58.]

Another division relates to the way a requirement guides the construction of the system. A functional requirement describes a functionality or feature of the system, and so is an active, concrete component in the design of the system. These requirements should be as open as possible regarding how they will be implemented, or which technologies are to be used. For example, a functional requirement might state: "The car shall have brakes." The collection of functional requirements need to be coherent and complete. Coherent in the sense that the set of requirements contain no contradictions among them and complete in the sense that all stakeholders' concerns are addressed. Ensuring that both criteria are fulfilled is difficult, especially for complex systems. [2, p. 47.]

Non-functional requirements can be characterized as restrictions imposed on the system under development, for example which programming language is to be used. Many systems are built based on a revision of non-functional requirements when a new, better version of a system is constructed. In such a case, the core functionalities of the system would stay the same but they would be more reliable or the user interface would be changed. Generally, non-functional requirements impact the whole system, whereas functional requirements are specific to singular functions or features. For example, a non-functional requirement might state: “the car must be able to brake to a full stop in 5 seconds from a speed of 100 km/h” In this regard, non-functional requirements are instrumental in system architecture: they affect the implementation of functional requirements throughout the whole system. [2, p. 48-50.]

Non-functional requirements can be further split into three groups: product requirements, organizational requirements and external requirements. Product requirements refer to the product itself, such as its functionalities, reliability or usability. Organizational requirements characterize the strategies and processes of the organization regarding standards and implementation requirements. External requirements come from factors outside the system itself, such as compatibility with other systems, legal requirements and ethical requirements. [2, p. 50.]

Some requirements, even some that are vital to the system’s proper functioning, are implicit: not clearly defined requirements but still implemented into the system because of their perceived necessity. These requirements are not requested by the client and are not always documented. For this reason, sometimes it is important to express which requirements are not wished to be implemented, so that time and resources are not used to implement requirements that are not wanted by the client. An explicit requirement, however, comes from the client and is documented. [2, p. 47-48.]

2.3 Things to consider when defining requirements

Especially in the case of the software industry, because the market pressure to launch a product as soon as possible is significant, many products do not have all their requirements met at launch. Hence, it is important to prioritize requirements and see that the most important ones are met first. This process demands the cooperation of all stakeholders because, for example, the developers do not know which functionalities are the

most important to the client and the client doesn't know which functionalities are the most time consuming or difficult to implement. The prioritization process is also affected by external factors: some features might become more important than others depending on the situation. Agile development methods are particularly effective from this point of view. [2, p. 125-127.]

Requirements can be prioritized using a prioritization technique, of which there are many. Ranging from quite simple approaches, like the self-explanatory top-10 technique to more methodical ways of prioritization, like the analytical hierarchy process, which requires the usage of mathematical matrixes and normalization calculations, these techniques have their weak and strong points. The simplest prioritization method that can successfully perform the task should be selected to get the best cost-benefit ratio. [2, p. 129, 130, 133.]

Some requirements might emerge from unexpected sources. One such case comes from the realm of software testing methodologies. When a software feature is complex enough, a test input division method called equivalence class testing might be used. Using this method requires certain assumptions to be made about the functioning of a software feature. For example, if the same alarm is unexpectedly generated from two different kinds of test inputs and it is not a bug, this behavior must be bound to a requirement. Understandably, this nuance might have not been obvious in an earlier phase of development, i.e. when defining requirements. [3, p. 278-279.]

Understanding or defining a project's scope is important before starting to define requirements. At times, the scope of a project is obvious, but it can be rather elusive, too. For example, defining requirements for a self-contained system is most likely more straightforward than defining them for one that interfaces with other systems. In the latter case, it is probably needed to work with people outside of your project since some of the functionalities might overlap. There is a significant amount of unnecessary work to be avoided by properly scoping the project. Occasionally, something called scope creep might happen. Scope creep means that the requirements of a system change significantly while in the development phase, usually by expanding. This effect can be resisted by defining requirements as they are needed, a method called "just-in-time production". [4, p. 21-22.]

3 Software development activities

3.1 Software development methods

Owing to the rapidly changing circumstances of the software market, software development is especially susceptible to time constraint issues as well as the more inherent problem of quality assurance. For these reasons, choosing the appropriate development model is crucial. For a long time, software was developed using the waterfall model (figure 1), where the transition from one developmental phase to another is strictly sequential and one-directional, starting from the most abstract going towards the most concrete. In the first phase, analysis, the functionalities of the system are defined through requirements engineering. This is done with earlier-mentioned basic tenets of requirements engineering in mind, such as clear and simple language, to properly include stakeholders in the specification of the system. Then, in the design phase, these requirement specifications are turned into software architecture, which determines the internal components and system structure with enough detail to allow their implementation. This phase is notably different from most other industries, since virtually all components are designed for the product under development, further emphasizing the importance of developmental efficiency. After this, the product is rather mechanically implemented and tested to ensure it reflects the design specifications. While this process might seem reasonable at first glance, it poses debilitating problems for developing a complex system, such as a patient monitoring device. For example, all requirements for the system must be agreed upon at the start of the project, with limited knowledge. Errors made in this phase will then trickle down to all following phases, only to be detected much later in the testing phase. [2, p. 33-35, 37.]

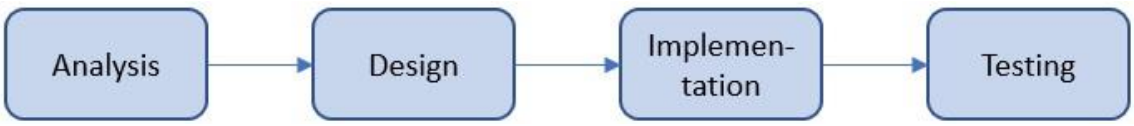
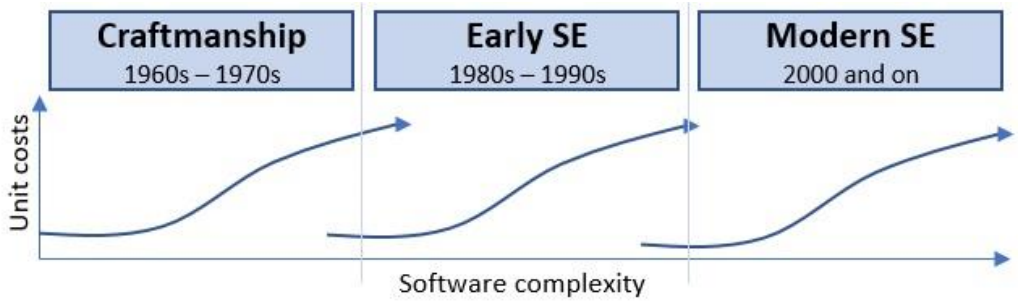


Figure 1. The waterfall process model. [2, p. 33.]

Developing a system by using ready-made building blocks would be highly efficient but, while there have been significant advancements in the field of component-based software development, building a whole system out of recycled software components has

been deemed impractical for most projects. Also, while software development methodology or process improvements can be independently incorporated, the widespread usage of software components would impact the design phase and requirements, and it would require a procurement infrastructure, standards and technologies such as Microsoft's COM+. [5, p. 1-4.] This doesn't mean software development doesn't include the usage of ready-made functionalities, only that software isn't yet built solely based on them [6, p. 14]. As software development has started to shift from craftmanship into something more like a manufacturing process (figure 2), more and more software functionalities have become reusable, while only the strategically sensitive components are still custom-made [6, p. 13-14; 5, p. 2].



	Ad Hoc → Waterfall	Process Improvement	Agile Development
Scope	0% component based; 100% customized	30% component based; 70% customized	70% component based; 30% customized
Process	Unformalized → Formalized	Repeatable	Iterative
Tools	Mostly Proprietary	Proprietary/ Commercial	Commercial and integrated
Team	Predominantly untrained	Trained and untrained	Predominantly trained

Figure 2. Many practices concerning software development have changed throughout the years. [6, p. 14.]

To combat the rigidness of waterfall development, agile development methods offer many improvements spanning multiple areas of software development. In 2001, a group of software development experts got together and discussed ways to improve software development to meet the demands of the modern era. They formed four dichotomies with preferences to ideals that embody the characteristics of agile methods: [7, p. 15.]

> Individuals and interactions over processes and tools. Working software over comprehensive documentation. Customer collaboration over contract negotiation. Responding to change over following a plan. [8.]

These dichotomies, while provocative, provide little information without some background knowledge. The first one, “individuals and interactions over processes and tools”, has much to do with motivation, talent and, ultimately, performance [9, p. 92]. One might think that high performance is achieved through rigorous application of efficient processes, but it has been shown that limiting the autonomy and creativity of employees has a negative impact on their performance through severely lowered motivation. Highly efficient processes also limit the opportunity to experience meaningful failures, a prerequisite to developing talent. The interaction of individuals involved in the project should drive the design process as more knowledge is cumulated, as opposed to designing the whole system before any implementation work. It is likely that individuals directly responsible with the implementation of the system are better equipped to make valid design decisions than any external party. People are also more motivated to work with decisions made this way. [9, p. 94-95, 97-99.]

The second one, “working software over comprehensive documentation”, ties in with the agile principle of continuous delivery of software. The logic behind iterative delivery of software is that, the sooner the developers get feedback from the customer, the sooner they can make a course correction. This is precisely why agile develop methods utilize iterative development: continuous assessment of the system lowers risk. Regarding working software, good design and technical excellence must be upheld throughout the project to avoid something called technical debt, a tendency of problems to grow in their cost to fix as time passes. A software fix, for example, must be ratified by fixing the underlying cause, not by creating a workaround that allows for more software to be built upon defective design. [9, p. 103, 109-110.]

The third dichotomy, “customer collaboration over contract negotiation”, concerns the way multidisciplinary collaboration is handled. Rather than holding formal meetings infrequently, communication should be continuous and open. This way, the risk of misinterpretations, uninformed decision making and scheduling mistakes from both parties is lowered significantly, increasing efficiency. Naturally, open communication must be upheld internally as well. Regular reflection and appropriate adjustments are at the heart of agile methods, both across business boundaries and within teams. [9, p. 112-113.]

Finally, “responding to change over following a plan” corresponds to the way how uncertain or volatile environments are managed. A core agile principle, reacting to change by

intimately involving it in the development process and not seeing it as an external obstacle, is beneficial when developing a complex system, where unanticipated events are commonplace. This attitude differs greatly from the waterfall method, where it is believed that enough competent analysis can produce an infallible design that delivers an accurate product. Realistically, a product evolves during the project as the stakeholders and the development team become more aware of what they need. From this perspective, one can appreciate how important transparency in the process is, since any hindrances in the flow of information affects the effectiveness of designing the product. [9, p. 114, 21-22.]

3.2 Software testing fundamentals & terminology

In the case of medical devices, ensuring that the product functions properly is critically important. When software is present, this responsibility ultimately falls to the verification and validation teams. It is a time-consuming and difficult process that requires a considerable amount of planning to be effective and efficient. The list of things to consider is expansive: requirements, design specifications, trace links, risk management and test procedures to name a few. Possibly the single most important part of software test planning is defining the expected result. What constitutes acceptable behavior for the software should be thoroughly thought out and set in documentation, not assessed by the tester on the spot. [3, p. 254-255.]

Software testing can be defined as:

> An activity in which a system, subsystem, or individual unit of software is executed under specified conditions. The expected results are anticipated from requirements or designs. The actual results are observed and recorded. An assessment is made as to whether the actual results are acceptably equivalent to the anticipated results. [3, p. 256.]

Specified conditions mean the steps the tester needs to take so that a specific feature or requirement is tested in isolation. Expected results mean the specific behavior of the software, known by the author and documented in requirements or design specifications. Actual results mean the observed behavior of the software or device, recorded in a format that requires little or no interpretation to assess whether they adhere to the expected results. The assessment means the decision to either pass or fail the test, ideally being as uncomplicated as possible. [3, p. 256-257.]

Testing software as opposed to hardware has some peculiarities. Whereas mechanical components are subjected to repetitive stress-testing, the point in software testing is to navigate through every possible variation in operation, challenging the software in variety, not in repetition. This means that, in addition to covering normal usage of the software in tests, unanticipated inputs designed specifically to attempt to break the software are included in proper testing of the software. Of course, stress-testing sometimes does also have its place in software testing. [3, p. 257-258.]

In the context of software testing, the terms verification and validation have two distinct definitions. Software verification is done while still in the development phase, and relates to the correctness of the software, i.e. whether the product has been built according to specifications. Software validation is done when the product is finished, to demonstrate that it performs in accordance to the user's needs. [10.] A pair of sentences can frequently be heard repeated to explain the differences between verification versus validation: "did we build the product right?" for verification and "did we build the right product?" for validation.

When a test result is concluded, its reasoning must be evident to a third-party reviewer. Just like in the scientific method, a tester needs to document their methods, expected results and actual results in a way that enables someone else to come to the same conclusion. This is usually done by writing step-by-step test protocols that explicitly instruct the tester to perform certain actions, simultaneously acting as objective evidence if they have been properly written. The FDA has defined requirements for objective evidence, they are as follows:

- The data can be evaluated by independent observers to reach the same conclusions
- The data is documented in a manner that allows recreation of the data or events described
- The documented evidence provides sufficient data to prove what happened
- The documentation was completed concurrently with the tasks
- The documentation is directly traceable to a person
- The data and documentation have been maintained in a way that provides traceable evidence of changes to it
- The data and documentation is maintained in a way that protects them from changes done to them [11.]

In addition to requirements concerning objective evidence, the FDA expects general good documentation practices to be followed. Also, all documents and records need to have a clear audit trail, they need to be complete, dated, signed and reviewed and they must be maintained in approved document management systems. [11.]

Also relevant to test result conclusion is the term test oracle:

> A test oracle is a mechanism for determining whether a system has passed or failed a test. It is used by comparing the output(s) of the system for a given test case input to the out puts expected by the oracle. Test oracles are always separate from the system under test. [1, p. 296.]

To thoroughly inspect a software product, it is usually tested from three distinct standpoints. Firstly, sub-programs are tested to ensure they perform their intended tasks independently without errors. This is called unit testing. Secondly, the various interfaces concerning the software are tested to ensure that data communication within the software and with other software functions properly. This is called integration testing. Finally, the whole system is tested to evaluate the entire software product's performance. This is called system testing. [3, p. 260.] To further ease the understanding of these concepts, consider a testing scenario for an automobile. Testing the material strength of a brake disc would be analogous to unit testing. Testing the braking force of the disc brake when attached to a tire in a controlled environment could be considered integration testing. Going for a test drive in the complete automobile would test the whole system.

Unit testing and integration testing can be seen as something called white box testing. White box testing requires intimate knowledge of the implementation of the software and is done by testing singular methods or objects within the code using automated testing software. Black box testing, which includes system testing, is done without intimate knowledge of the implementation of the software simply by creating test cases based on the requirements or design specifications imposed on the software. Gray box testing is a combination of these two, where the software is tested against specifications or requirements but with knowledge of the implementation, for example when a test case is designed to consider problems that could have only been made known to the tester by examining the code. [3, p. 277-278; 12.]

Software testing should be done in a way that all possible variations of operation are examined. In the case of unit testing, this is usually done by using software that gives a percentage value of how much of the software has been exercised. An important thing

to note is the difference between the terms exercise and test. A unit testing software can exercise 100 percent of the code with a pass-rate of 100 percent but that only means that all the functions, methods or objects in the code have functioned using some values. The only measurable attribute gotten from this kind of test is path coverage, which tells us that the test's coverage across functions but the quality of tests remains unknown. Path coverage, however, has greater value than branch or statement coverage, since the different branches and statements will be exercised multiple times for each path. Still, the issue of testing is not an arithmetical, but a logical one. Turning a light bulb on and off demonstrates its operation in principle, but turning it on and off for thousands of times and for extended periods of time might better indicate the quality of the light bulb. [3, p. 263-264.]

Integration testing is even less of a straightforward matter. The interfaces of a software product can be highly complex and numerous. While integration testing is usually quite unique for each system, some general rules concerning priorities can be made. For example, safety critical functions of interfaces should be considered a priority. Also, integration testing is prone to be done partly in the hardware realm, sometimes even with analog signals, further complicating testing. For medical devices, a typical integration test is associated with medical device communications protocols, which require sequencing of messages in rapid succession. This is usually achieved by using a test computer between two processors to intercept transmissions. The computer can then test the validity of the transmission or inject errors into it, testing error-recovery. [3, p. 268-270.]

System testing fundamentally deviates from unit testing and integration testing, in that system testing is done not against design specifications, but against system requirements. System testing should address functional aspects related to intended uses, such as performance issues, security, compatibility and documentation accuracy. One might say system verification resembles software validation, but there is still a vital distinction: system verification is done in the development phase to verify the proper implementation of requirements and includes anticipated test results, whereas validation is done to demonstrate that the finished product satisfies the intended user's needs. [3, p. 272-274.]

Apart from a few exceptions, some commonalities are shared by all software testing methods. Namely, their execution is generally developed from three standpoints: test

designs, test cases and test procedures. While test procedures are the product of this activity, the creation of test designs and test cases make the whole process run more smoothly. Test designs are written using simple natural language so that the reviewers and eventual test procedure author can properly understand the logic behind the test. A test design should provide sufficient information to accommodate the creation of properly insulated test cases: the requirements that will be tested, initial conditions or state of the software, a short description of the context of the test, rationale for the test, inputs and expected results and instructions on how to provide objective evidence. Simply put, a test case is a collection of test inputs and their expected results (table 1). A respectable amount of information can be gained from a single test case table and creating it forces the test designer to consider the test cases more carefully. [3, p. 290-292.]

Test inputs	*Expected Result*	*Notes*
1	Result accepted	
5	Result accepted	
10	Result accepted	
0	Result rejected	
11	Result rejected	
001	Result accepted	Unclear from requirement, may reject if only allows 2 digit input
1.1	Result rejected	
.8	Result rejected	
-1	Result rejected	
Null	Result rejected	
a	Result rejected	
A	Result rejected	
%&	Result rejected	
111111 ... 1	Result rejected	If behavior allows, enter 257 1's to error guess buffer overrun up to 256 byte buffer

Table 1. An example of a test case table. [3, p. 292.]

Test procedures combine the test design and test cases into step-by-step instructions that ensure the proper setup, sequence of test inputs, observed results and objective evidence are observed in all test cases. Consequently, there are many critical benefits to using test procedures. First and foremost, when a software defect is found this way, it can be easily recreated by following the same procedural steps again. In contrast, while unscripted testing methods may occasionally be more efficient at finding defects, recreating the conditions where the defect appears might be practically impossible. Secondly, highly detailed test procedures are less demanding on the individual tester's skill or knowledge level, allowing less experienced testers to be effective at testing. This is especially important in highly voluminous testing programs, where extra testers might be

added on short notice. Also, a high level of detail leads to high-resolution testing, which is likely to uncover defects that are not necessarily even related to the actual requirements being tested. The downside to this level of detail is that the procedures are costly to create and maintain, and they are susceptible to errors in general. A typical test procedure consists of tester actions, tester observations and assessment of results. Tester actions are short and simple instructions for the tester, such as “Click the OK button.” Tester observations are intermediate confirmations that the test is proceeding as designed, verifying that nothing unexpected happens in the transitional steps leading to the requirement-verifying observations. In the assessment of results, the tester will determine whether the test has passed. This can be represented by a simple “Pass” or “Fail” checkbox. [3, p. 292-294.]

3.3 Software testing methodology

Often, software functions perform complicated tasks, or tasks that are not predefined. Some software functions have such a large amount of possible inputs that covering all of them in a test case is not feasible. To illustrate this, consider that there are about 3.026 x 10^15 possible combinations for an 8-character password, which is more than the number of stars in our galaxy. For these situations, where testing all possible values is impractical, choosing the right subset of values is critical. This can be done by dividing test inputs into groupings that would most likely produce the same test result. This method is called equivalence class testing. For example, positive real numbers and negative real numbers could be considered two equivalence classes, since they most likely produce the same test results among themselves. Having just two classes is not going to cover all possible test inputs and only looks at the problem from one point of view. The equivalence class dividing process itself will lead to the tester to consider probable weaknesses of the system, in addition to cutting down the amount of redundant testing of similar inputs. [3, p. 276-277.]

Closely related to equivalence testing, boundary value testing is done at the boundaries of equivalence classes and has been shown to be an effective method to finding software errors. To identify boundaries, knowledge of the implementation of the software is required. Computers use bits to store, process and transmit information and while the value 255 might seem unremarkable to some people, it is the highest possible value that can be shown in 8 bits and can be considered a boundary value as such. In addition to cases

like these, more obvious boundary values can be identified. If acceptable inputs for a software function are integers ranging from 1 to 10, these can also be considered boundary values. Generally speaking, boundary values should be tested around the boundaries, so if 10 is a boundary value, the values 9, 10 and 11 should be tested. While these two examples hardly illustrate the realm of possibilities to boundary testing, one might already appreciate the efficiency of combining equivalence class testing and boundary testing to create test cases that reliably catch software errors while keeping the amount of testing to a minimum. [3, p. 279-281.]

For systems that have input and output devices, testing the human-system interface is in order. Both the ergonomics and proper functioning of the interface are important: an interface that enables easy operation and interpretation of the system's conditions will decrease the probability of erroneous use. The system must not accept invalid inputs and it must give comprehensible outputs to the user. To achieve this, the input tolerance of the system is tested. Invalid user inputs should be handled in a way that blocks them from affecting the system and helps the user to correct or remove their erroneous input. First, the system is tested using proper user commands to ensure it meets its specifications in normal circumstances. Then, the system's response to different types of improper user commands is tested. These include using the wrong data type or size, violating input restrictions, skipping mandatory fields, inundating the system with unreasonably long inputs and activating input switches out of sequence. The detection of erroneous user inputs can be done either in real time, or upon committing a value to a field or to an entire form. HSI (human-system interface) outputs are similarly tested by first operating under normal circumstances to verify the system's adherence to specifications, after which they are tested with the same kind of improper user inputs as defined previously. [1, p. 373, 375-377.]

Sometimes, doing software testing through automation has a lot of benefits. It's faster, more reliable and uses less resources. Some tests, like stress, timing or response time testing are not even possible to execute manually. Of course, automated testing is not without downsides. In addition to normal testing activities, the tests need to be developed similarly to any other software, thus being susceptible to quality issues. One test method that is particularly suited for automated testing is the injection of large quantities of random test inputs. This type of test gauges the stability and reliability of the system with a large range of unexpected test inputs while requiring a minimal investment of time and

effort. Yet due to the unplanned nature of automated random testing, a reliable test oracle cannot be used so only the most obvious faults in the system might present themselves to the tester. Typically, automated random testing will expose accumulative software faults such as memory leaks. Automated random testing can be combined with another testing method such as equivalence class partitioning to make it more efficient. [3, p. 302-303; 1, p. 378, 380.]

Testing a system at lower load levels only technically demonstrates that the system meets its specifications, since it might have significant faults at full load. The system's behavior at high loads must be specified and reflect those specifications. For example, the level of CPU load for performing a specific task could be defined in the performance requirements. Typically, performance testing metrics include task response times, how many external systems the system can handle, how many resources the system utilizes and system reliability. [1, p. 382-384.]

Some errors and faults are inevitable in the intended environment of a system. External conditions such as human error or network connection errors and even some internal conditions such as software errors are bound to be a factor in the field use of the system. Consequently, a system must be capable of continuing or resuming operation without loss of data in the event of such an error. Testing this behavior is called recovery testing and is usually done by injecting a fault into the system. Hardware-wise, this can be done by shorting connections or using a software that simulates any such behavior. Strictly software-wise, faults can be injected by techniques called software mutation, where the actual code is altered in ways that cause abnormal behavior. Less subtle ways can also be employed, such as flooding the memory or other resources, aborting applications or disconnecting cables. Regardless of which method is used, resulting behavior is observed and compared to recovery specifications. [1, p. 385-386.]

Whereas performance testing is done to determine the limits of the system's performance in normal conditions, stress testing is done to determine the limits of conditions where the system can still operate normally. These conditions include environmental factors such as ambient temperature, mechanical stress and electromagnetic radiation or overpowering the system's resources. Stress testing is done to determine the system's robustness and elasticity. Robustness means the capability withstand stress and changes in conditions, elasticity means the capability to return to nominal performance after the extra stress is removed. Typical tests include maximum data transfer rates,

maximum channel and data bus usage and maximum resource usage. As always, all these behaviors must be documented in requirements or specifications and they must be met by the system. [1, p. 386-387.]

An unscripted method, exploratory testing is done by simultaneously designing and executing test cases based on educated guesses, in real time, while learning about the system. Consequently, this method enables the tester to concentrate on possible defective behavior as it appears without time-consuming formal preparation, but the method is highly dependent on the individual tester's skill and knowledge level: expected behavior must be known to spot abnormal behavior. It is exactly this prerequisite of knowledge that differentiates exploratory testing from ad hoc testing. While often an effective method to find defects, exploratory testing is not without flaws. It tends to be unstructured and badly documented, giving rise to difficulties recreating defective behavior when it is found, since it might be the product of a highly specific sequence of test inputs. Exploratory testing, then, is best used to compliment other, more formal types of testing methods. [1, p. 445-446.]

When modifications to a system are done, new faults can be unknowingly introduced into the system even when the changes were made to fix another fault. Of course, the affected areas then need to be retested. This practice is called regression testing. In the development and maintenance of complex systems regression testing accounts for a large portion of all testing efforts so there is considerable emphasis on efficiency. This efficiency can be attained by partly reusing test suites from previous versions. To do this, the test engineer must analyze exactly which parts of the system have been modified and how, and what kind of impact do these changes have to the system. Then, a regression test strategy that contains the scope, coverage, success criteria and testing sequence of the regression test is devised. Finally, the test case suite can be updated to include relevant test cases, after which the tests are performed, analyzed and reported. There are multiple different kinds of regression testing strategies. Most of the strategies result in testing only a part of the system using some kind of election logic (figure 3), running the risk of leaving a potentially impacted area of the software untested for changes. Conversely, using the prioritizing and rotating regression testing method, a different part of the test suite is always executed each time a modification is made to the system, until the whole test suite is fully executed. In this method, the subsets of test cases are prioritized by their tendency to detect faults, for example. [1, p. 447-450.]

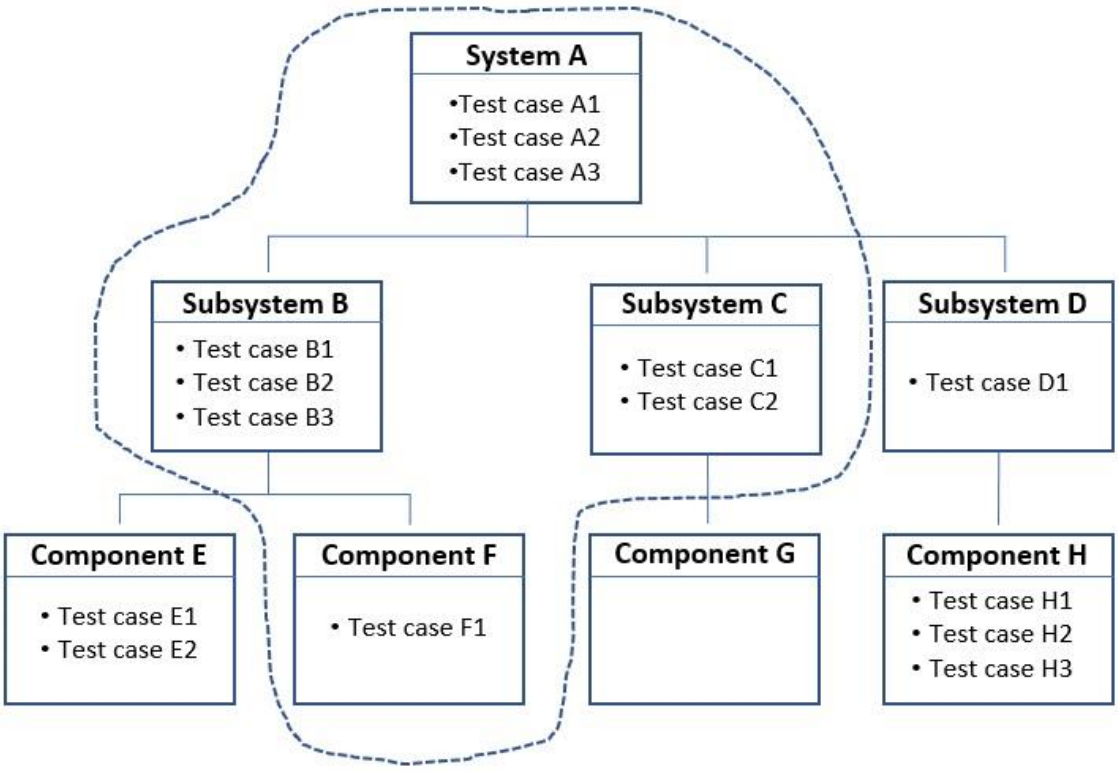


Figure 3. Most regression testing methods omit parts of the system from the test coverage. [1, p. 451.]

After the normal defect testing phase has been completed, the reliability of the system is verified. Reliability testing is different from other types of testing in that it should be based on a statistically significant sample size and it produces, in addition to less-defective software, a reliability estimate that is based on a mathematical model. The test data is also created somewhat differently. Whereas normally, the system is tested with maximum coverage, reliability testing requires the creation of a typical user input profile. This profile contains both normal or expected user inputs as well as reasonable abnormal inputs, which is reflected in the test data. After a statistically significant number of defects have been found, an appropriate statistical model is chosen to calculate the probability of future defects. Hardware faults are often due to components wearing out and they can be replaced when nearing their estimated end of life before they fail. Software faults cannot be anticipated with similar certainty. It is known, however, that any given system's reliability tends to grow over time. Also, whereas hardware faults are usually fixed by changing a component, software faults are often fixed by changing the system itself, i.e. changing its code. New revisions of a software system are prone to introducing new software faults to the system (figure 4). The system's reliability index can be calculated

with the Computer-Aided Systems Reliability Estimation (CASRE) tool. It is a freely available tool developed by the Jet Propulsion Laboratories in the United States. [1, p. 402-403, 405, 410.]

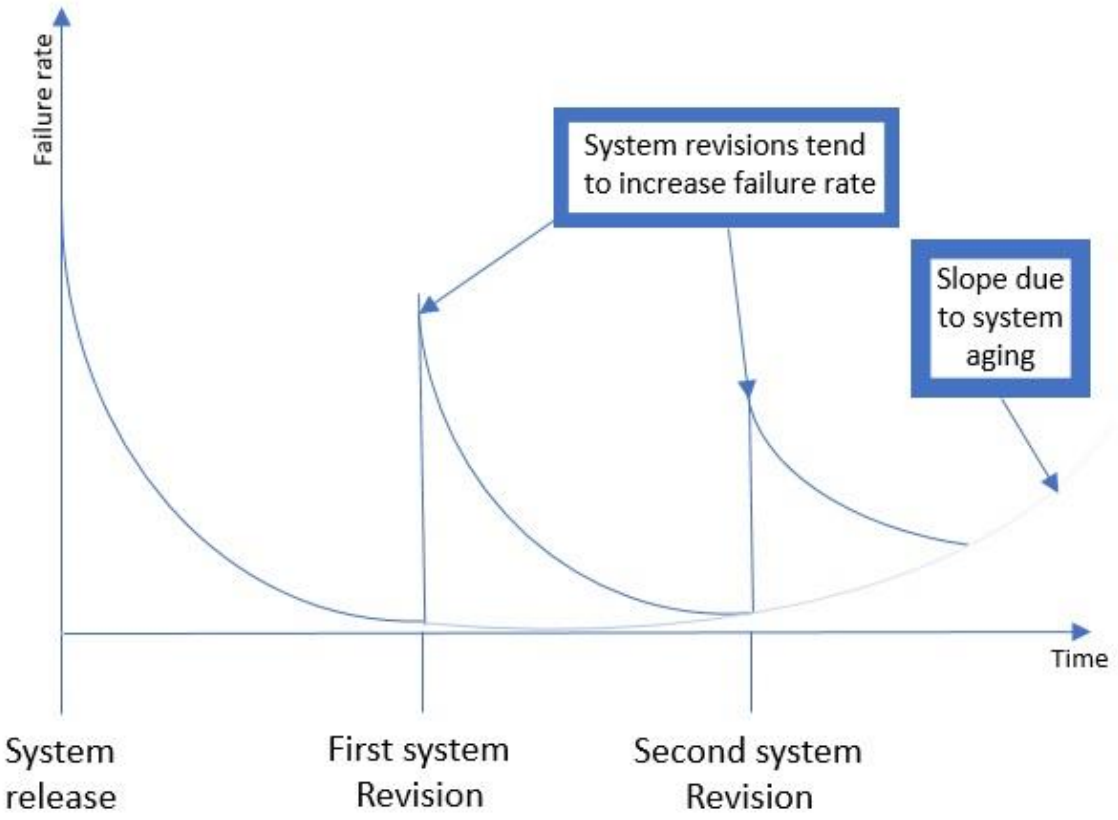


Figure 4. Failure rate of a system through multiple revisions. [1, p. 410.]

3.4 Documentation and compliance

For this thesis, the software in question is that of a medical device. As such, it is expected to endure extra scrutiny concerning standards compliance. The safety of a medical device must be demonstrated with acceptable confidence. The responsibility of the device's safety is shared between all stakeholders and must be accounted for throughout the device's life span. [13, p. 3.]

Since the safety of a medical device cannot be considered in absolute terms, it is done through risk management. Risk is comprised of the hazard (i.e. potential for danger), the probability of occurrence of the hazard and the severity of the hazard. Assessing risk begins with identifying hazards, which are then evaluated, estimating the risk of each

hazard. This estimation can be based on computation and evidence but in practice, for medical devices, it is usually based on the experience of health care professionals in addition to safety design engineering. Right now, classifications of medical devices differ between the United States, Canada and the European Union, but the Global Harmonization Task Force (GHTF) is seeking to unify the classifications. Attributes that affect the classification of a medical device include matter such as the invasiveness of the device or affected body system. The more potential for a hazard, the stricter the regulations for that device become. [13, p. 3-4.]

The safety of a medical device must also be ensured through its performance and effectiveness. A device that does not produce the intended medical effect is not clinically effective and, hence, not safe to use. Performance-wise, a medical device whose operation is unnecessarily difficult is also not safe to use, even it technically could perform its intended function. Therefore, the performance of a medical device goes hand in hand with its safety. [13. p. 4.]

Standards are a practical way of enforcing medical device regulations. They are documented agreements that contain specifications or criteria that, when observed, instruct the proper manufacturing of a product to fit its purpose. Standards contain specifications ranging from device dimensions, calibration procedures and design characteristics to measurement accuracy, energy output limits and quality management systems. While most standards combine different kinds of specification types and are specific in nature, some standards, such as the ISO 9000 and ISO 14000 series, are generic standards that can be applied to any organization. Medical devices of different manufacturers do not have interchangeable parts, which is one of the problems that the GHTF seeks to rectify through standardization. The ISO (The International Organization for Standardization) standard specific to medical device quality assurance is the ISO13485:2016. [13, p. 18; 14.]

Some regulations necessitate the generation of documentation and records for all activities related to products. The product-related activities of the modern world can be quite complex and therefore produce a lot of documentation, which needs to be managed. A company’s records can be in both paper and electronic form, further complicating documentation management. In fact, inadequate documentation is common basis for getting a warning letter from the FDA. [15, p. 32, 47.]

Documents are managed with a document management system, with functionalities such as routing, version control, archiving, electronic signatures and searching. Document routing means the flow of documents between persons for review or approval. Generally done electronically, document routing can also be automated to achieve a high level of efficiency. In many instances, companies decide to subscribe to a software as a service -type of document management system, outsourcing it. This poses extra tasks to ensure compliance and security. Since documentation must be signed, electronic documentation calls for electronic signatures. In practice, most companies employ a hybrid of paper and electronic document managing system: the documents are generated electronically, but then printed and signed. [15, p. 50-53.]

4 Analysis methodology

4.1 Problem statement

Since the task is to give out improvement recommendations, it comes naturally that the current process must be thoroughly inspected. In order to get sensible data, an investigative method must be first developed. Most of the problem areas are found directly in one particular document related to verification activities. This document is the test procedure formulated from system requirements, which tests the functionalities of the whole system. The system in question is a multiparameter patient monitoring device, which houses measurements ranging from hemodynamic monitoring to neurophysiological status. Creating a system-wide test procedure covering all the functionalities and their accompanying calculations, alarms, user interface functions and settings is a complex and arduous task, so the entire chain of actions must be reviewed.

Rather than only looking at finished test procedures, much effort will go to going through documentation and interviewing relevant persons for information related to the verification test procedure creation process. This way, sufficient knowledge about the process will be acquired and, ideally, it will become possible to improve the practices that produces defective test procedures instead of fixing isolated incidents. Interviews are likely to be a key source of information, because they can expose the discrepancy between planned and actual practices. In a complex process with time constraints, some steps are probably going to be misunderstood or somewhat ignored. So far, three potential causes for mistakes in the system verification procedure can be identified: systematic

defects born from a bad process, systematically bad implementation of an otherwise viable process, and individual mistakes stemming from lapses in knowledge.

Systematic defects from a bad process would be the most desirable scenario in terms of effectiveness of a correction. Fixes made here would have a lasting, widespread effect since the fix would be systematic as well. Also, the problem would likely be fairly easy identify and isolate. However, defects in the process itself are unlikely this far into the development cycle and would be laborious to fix. Systematically bad implementation of a process refers to an advanced stage of disregard towards good practices. This would be the worst-case scenario since it would require extensive changes that would be difficult to both identify and manage. Errors arising from minor problems in work instructions communication are the most likely candidate for implementable improvements. These changes would be light-weight, quick fixes that wouldn't affect the procedures' creation process that is already in place.

Along with documentation such as work instructions, another important point about the verification procedures is how they're handled in the requirements management tool. Requirements, trace links, design specifications, test procedures and test results must be documented in a way that satisfies the FDA's regulations [11]. This creates the need for the procedure creation process discussed earlier. The progress and results of the process will be concretely visible in the requirements management tool.

4.2 Investigation

A logical starting point to understand the current process is to identify all of the steps involved in it. Each step is complex on its own and is usually accompanied by transitionary tasks such as version control, document approval and planning activities. Any confusion about the tasks will create inconsistent results. While the software and documentation will be technically compliant, inconsistencies might lower the confidence a client has in the product.

Owing to the type of information this analysis will produce, methods fit for qualitative research shall be used. One such method is in-depth interviewing. It is a research technique where intensive interviews with individuals are conducted to explore their knowledge on a specific topic [16, p. 3]. The process of finding suitable interviewees is

central to the outcome of the analysis: they need to be knowledgeable, and since the topic is highly specialized, their specific knowledge base must fit the purpose of this task. The emerging themes in the first interview are tested with following interviews and the most commonly mentioned issues are considered higher priority. [17, p. 17.]

An initial list of questions must be generated (appendix 1). This is done based on preliminary information gained through professional literature on requirements engineering, software development activities and documentation, work experience in software verification and initial conversations with experts. The initial list of questions serves only as a backbone to the interviews and most likely many of the truly useful topics will emerge spontaneously. In fact, strictly following the questions that have been generated in advance might have a negative impact on successfully gathering useful data in this context. Useful data in this case would be either contrasting or complementary views on the same themes or issues, that were not necessarily anticipated. [17, p. 18-19.]

The data from the interviews will be gathered by recording the audio of each interview. This way, minimal effort will be spent on taking notes, while maximal effect on information retention will be gained [17, p. 18]. This is highly important, because the established way of interviewing will not produce standard answers to standard questions, and the topic includes a large amount of delicate information.

An in-depth interview has many advantages to other types of data collection methods for this particular application. First and foremost, since it is an open method, the data gained is not bound to the coverage of a list of questions as it would be when using a survey [16, p. 3]. This way, the interview serves two purposes: it helps teach the interviewer the general practice of verification procedure creation in addition to uncovering the more specific grievances in its execution. Both are equally important, since the improper functioning of a process can only be identified when the proper functioning of it is understood. Secondly, this method is more comfortable and engaging to the interviewee [16, p.3], further improving the likelihood of good information.

An in-depth interview does have some downsides. Typically, interviews are conducted by outsiders so establishing rapport, a relaxing and encouraging atmosphere that inspires trust, is important to ensure the interviewees provide the interviewer with honest and useful information [17, p. 19]. This is mitigated by the fact that, in this case, the

interviewer and interviewees are professionally acquainted, and the topics are impersonal. The interviewees are likely to give a full account of events and information without much probing. Another issue relates to the objectivity of the information given by the interviewee and the neutrality of the interviewer. Since the interviewees are directly involved in the process of creating the verification procedures, they can be biased. The interviewer needs to be neutral during the interview itself by asking non-leading questions and during the analysis phase by not contaminating the data with personal opinion or interpretation. Finally, owing to the small sample size and specific nature of this analysis, knowledge gained here is not likely to be generalizable. [16, p. 3-4; 17, p. 19-20.]

5 Results and analysis

5.1 Verification procedure creation process

Creating a verification procedure starts from understanding the user requirements. As expected, most of the user requirements have been defined during an earlier project since the system in question is a newer version an existing one. From the user requirements, a set of initial system requirements can then be defined mainly by lead system designers. These requirements, similarly defined earlier, cover all of the user requirements but contain enough technical detail to aid in implementation work and, consequently, the number of the system requirements is roughly ten times that of the user requirements. After that, the system requirements are maintained by the teams reactively: requirements that produce complications for development or testing activities are fixed.

During a project, new features might be introduced mid-development. In these cases, new requirements can be added in a modular fashion as needed. For example, if a user requirement states that all physiological measurements must have an alarm, that requirement is not likely to change when a new physiological measurement type is added, but lower level requirements will have to define the behavior of the new alarm. Additionally, test procedures must be updated to accommodate, and regression testing must be executed when the new feature is implemented. While this particular case is not the most common, it illustrates the cascading effect that any changes to the system has to verification activities.

Special attention has been given to making system requirements clear and understandable. Quite recently, system requirements went through minor fixes and tweaks to make them less ambiguous. Such work is valuable, since the quality of system requirements is essential to all subsequent activities: the system verification procedure, software requirements and design details are all dependent on system requirements.

The system verification procedure is produced to verify the functionalities defined in the system requirements. Producing it is the collective effort of many teams, each specializing in the development and testing of a portion of the whole system's functionalities. Each team writes the portion of the system verification procedure that corresponds to their specialization. The process starts with identifying requirements relevant to that team's responsibilities, grouping them into coherent segments and creating test cases, which are then turned into test procedures. For example, all relevant alarms are tested in one section while user interface functions are tested in another. This way, changes to the test setup mid-testing is kept to a minimum, optimizing the use of time. The grouping is also done by the testing method that is going to be used. Automated testing is the desired method, since the execution and re-execution of a well-designed automated test consumes virtually no resources. Manual testing is used when automated testing is not viable, and in the rare cases that neither of these methods can be used, the code itself will be inspected. Of these three, the most common is procedural manual testing, increasing the importance of the quality of manual verification procedures.

A series of test cases becomes a test procedure when sequential instructions that guide the tester into creating the desired testing conditions are added. For example, the specified conditions for testing a physiological alarm include using default settings for the monitoring device, connecting specific instruments to it and producing the alarm using a specific method. A high level of precision is needed to write a verification procedure that reliably creates the specified conditions for test case after test case.

Most of the time, test procedures are created by recycling an old test procedure. This saves a significant amount of time, and since the procedures do not generally undergo large-scale changes between runs, recycling procedures is quite practical. In fact, creating test procedures from scratch each time is not realistic in this scale. There are some generic tasks involving the recycling of test procedures, such as managing trace links and identifier attributes in the requirement management software. Sometimes, additional

tasks such as merging multiple procedures or changing testing the testing method cause more comprehensive changes to the recycled procedure.

The most significant recent change to test procedures is the way objective evidence is gathered. Until recently, it was mostly the tester's responsibility to find a way to document the objective evidence of the software's behavior. This way, the test procedure itself was simpler to maintain, but test execution and documentation relied more on the individual tester's skill and knowledge. Now, objective evidence is recorded into the procedure steps themselves with procedural steps that are designed to force the right information to be documented. Also, because test coverage was increased, and one result step can only be linked to one requirement, the number of individual test cases in the new type of test procedures increased considerably. These two factors have increased the total amount of steps in a test procedure by a large margin.

Whereas system requirements describe the functionalities of the system as a whole, software requirements go deeper into the details of implementation. Multiple software requirements may have a trace link to the same system requirement, because many features can derive from the same system requirement, or the same functionality can be found from multiple locations within the software. Software requirements are grouped into logical compartments, and each team is appointed a few of these compartments.

A software verification procedure verifies software requirements. Unlike the system verification procedure, where the whole system is tested, a single software verification procedure tests one of the compartments mentioned earlier. However, a software verification procedure is much more detailed and comprehensive than the system verification procedure. One might say that the system verification procedure's complications stem from its sheer volume and variability of functionalities, while the software verification procedure's complications come from the high level of precision and test coverage.

Design details, which are the most detailed specifications of the system, are not tested using a test procedure. They contain information such as font sizes, text locations and other user interface functions, which do not require a complicated test setup.

The quality for all procedures is ensured by dry running them. Dry running a test procedure means executing the tests as one would during verification, without commitment to the test results. This way, problems that would be found during official verification are

found earlier, when the process of fixing them is more light-weight. Changes to the test procedure during official verification are made through a rigid chain of extra documentation and approvals to ensure practices that satisfy the FDA's regulations, which takes more time.

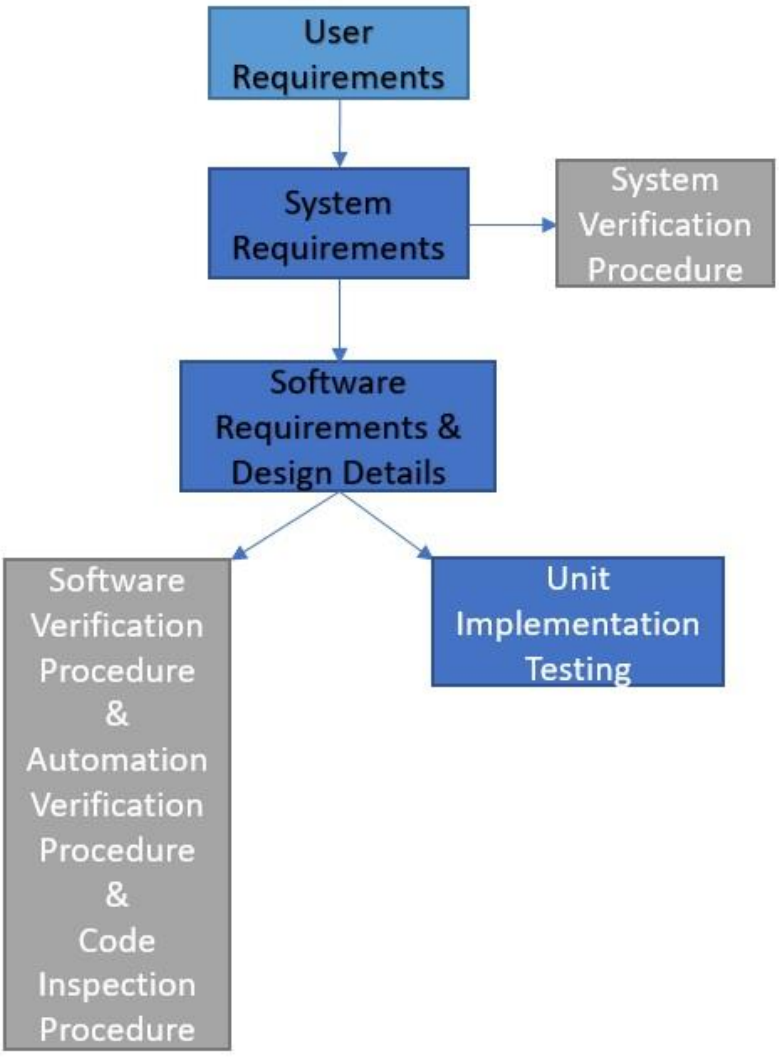


Figure 5. The verification procedure creation process.

5.2 Verification & procedure creation problem areas

During the third interview, recurring themes started to emerge, meaning that an appropriate sample size had been reached, so the total number of interviews was left at three. Valuable information was gained about the status of manual verification procedure creation process, which surprisingly did have systematic problems. Starting from the easier

types of problems to fix, going towards problems that require more heavy-weight adjustments, these problems will be explained with enough detail to serve as rationales for the improvement recommendations discussed later.

The first problems concern the system requirements. They are not methodically checked for issues, which leads to problems that reveal themselves later than would be optimal. For example, when the system verification procedure is created, test coverage, the test method and other types of filtering are done to ensure that the procedure covers all relevant requirements but the system requirements themselves are not inspected for relevance or quality. Practically all fixes to the system requirements are made reactively, when a problem presents itself. This might be a strategy to save time because scanning through the system requirements is a time-consuming activity, but since unexpected events affect the up-to-dateness of system requirements, it is not a thoroughly calculated risk. Also, problems with system requirements may multiply when going down the hierarchy of requirements or going forward with the process of creating test procedures. The earlier these problems are fixed, the less time they have to create a snowballing effect on the workflow.

Some of the bigger problems revolve around dry runs. Their execution is a dilemma: on the one hand, they take up a lot of time with no guarantee for the desired effect, but on the other hand neglecting them is a risk that can lead to a defective test procedure being used in official verification. The effects of not dry running a test procedure were recently seen. Issues that would have taken minutes or hours to fix before official verification took days in the worst cases. A big factor in this is the bottlenecking of some roles, such as lead system designers. Every fix needs to be approved, and there is a limited amount of people who are authorized to do so. One might think that it would be as simple as removing these restrictions, but the compliance of documentation could be compromised. A potential danger with dry runs is a lack of motivation for precision, which can translate into a waste of time and resources. The quality of dry runs has been noticed to vary between individuals.

Generally speaking, some decisions are made too late. Occasionally, for example, the test method of a requirement or a group of requirements is changed after writing the test procedure has already been started or even finished. This ties in with the issue of poor information flow within the workplace. Regarding that, there seems to be three core problems: 1. the teams are autonomous units, so cooperation is not a built-in mechanism, 2.

guidance documents are not generated and distributed efficiently, and 3. some valuable information is known only to individuals. Cross-team communication needs to be artificially maintained, which creates extra work. Work instructions are high-level legal bounds that cannot be applied to the everyday activities of a verification engineer, so they need to be translated into unofficial guidance documents. While this activity is essential to the quality of verification activities, virtually no resources are allotted to it so it is hastily done without much coordination.

The issue of guidance documents and valuable information being stuck on individuals is part of a bigger problem which needs to be explored. Roughly speaking, the level of expertise is kept at the bare minimum while workflow is controlled using rigid processes and guidance documents, especially in the case of manual testers. While this is a strategically sound decision, it places a lot of pressure to the documents' quality and coverage. For example, testers are also partly responsible for writing test procedures and are reliant on the guidance documents to do it competently. However, writing test procedures is essentially a creative process that involves unpredictable variables so comprehensive guidance documentation is difficult to produce, especially when done with limited resources. These effects together create a problem: individuals can unknowingly write defective test procedures, which then places unreasonable pressures on corrective mechanisms such as dry runs and document reviews. Similarly, valuable information does not get distributed efficiently because problems that could be resolved with increasing collective expertise are instead handled by having the already-competent people perform the difficult tasks. This can be considered an example of the Pareto Principle in action, where 20 percent of workers produce 80 percent of the results [18]. As a naturally occurring statistical phenomenon, it is difficult to offset but some measures can be taken to mitigate its effects.

The system verification procedure suffers from the most issues. It is derived from system requirements, which are not as detailed as software requirements, so creating test cases can be more complex. Abstract requirements can also be tested in multiple ways, inherently causing inconsistency. It is the longest procedure, so managing it is more difficult. It has the most authors, further complicating the issue of consistency. It sometimes includes the same test cases as some software verification procedures, so choosing test coverage requires more work. It is also written with less specific procedural steps, so executing it can be more challenging. Many of the problems concerning the system verification procedure itself are inherent to it, but some of them can be attributed to general

bad practices. For example, inconsistent use of terms and differing test cases for the same requirements stem from bad communication between authors.

5.3 Improvement recommendations

Some problems seem to have no practical solutions to them. Firstly, the system verification procedure is written as a shared effort, piece by piece, by all of the teams. To ensure a consistent style of writing for the whole procedure, either all of the authors should be in absolute agreement on all aspects of procedure writing details or a dedicated taskforce of few enough people should be employed to write the whole system verification procedure. Both options are unrealistic, so the problem cannot be completely eliminated, only mitigated. Secondly, most verification procedures are recycled from an old procedure. If the old procedure's quality is bad, the new procedure will also be of poor quality. This problem cannot be solved in one fell swoop, rather the quality of the procedure will increase between each iteration as it is recycled repeatedly. Finally, to properly estimate the effects of the recently increased test coverage and amount of procedure steps on the quality of the verification procedures, an unreasonable amount of work would have to be done. A quality evaluation method would have to be developed and a statistically significant sample of procedure steps from before and after the changes would have to be inspected. These three issues will not be examined further in detail.

The first task for improving the quality of manual verification procedures is to determine the earliest feasible point of entry to adjust the procedure creation process and make one or more systematic corrections. These corrections will produce results by themselves in addition to making changes further into the process easier to implement. The recommendations are numbered for further use.

1. This particular improvement recommendation can be applied to both system and software requirements. When creating test cases for requirements, those requirements should be inspected for quality and relevance. The advantages of this practice would be the 100 percent inspection coverage of tested requirements, the early detection of errors and time-efficiency of requirements maintenance. The disadvantage would be that the absolute time used to create test cases would show an increase. However, this time would be well-spent since fixing errors in later phases would be significantly slower and

the routine maintenance of requirements would increase the expertise of individuals performing it.

2. The issue of dry runs must be resolved by accepting the dilemma explained earlier and choosing the better of the two bad options. Coupled with effective requirements maintenance, comprehensive dry run execution is a good way of minimizing the risk of defective procedures being used in official verification. Dry running and fixing a procedure is also an expertise-reinforcing activity that can be performed by less-experienced members of the verification team owing to the lower risk. In addition to fixing mistakes in the sequence of procedural steps, it should involve the assessment of test cases against requirements, i.e. are the test cases properly testing the requirements linked to them. At this point, the requirements themselves would have been assessed, so checking the test cases is important because some of the requirements might have been modified. The assessment of test cases would then have a three-fold effect: assuring the quality of the test cases, controlling the side-effects of the requirements maintenance and educating the tester performing test case assessment.

3. Improving information flow would solve many problems. Presently, members of each role (test leads, system designers etc.) hold regular meetings to discuss and make decisions on issues relevant to that role and are then trusted with forwarding that information, but there is no mechanism to ensure this. These meetings should be more methodical: topics should be better known beforehand so that attendees can prepare and making informed decisions should be valued over making fast decisions. The deployment of these decisions should be monitored, which can be done by conducting a follow-up discussion in the next meeting about items discussed in the preceding meeting. Especially for test lead meetings, a recurring topic should be the unification of practices. For example, test case creation should be based on an agreed format.

4. All useful information should be centralized in a single, accessible hub. Presently, guidance documents are difficult to find without explicit knowledge of their location, of which there are many, and they are not comprehensive. The self-education of workers through collective documentation should be made as easy as possible, because that is the way information is most efficiently distributed within the workplace. This practice would also limit the effects of information loss deriving from changes in the employee roster.

5. A common terminology across all teams would help reduce the inconsistency of verification procedures. For example, the term parameter is used to mean either a singular measurement (ECG I) or a measurement type (multiple ECG channels are still considered one parameter). Establishing terminology should be a collective effort because it would motivate all parties to conform, in addition to the more obvious benefit of higher levels of knowledge being available in the process. For the same reasons, the generation of guidance documents should include members from each team. Enough time should be explicitly allotted to these activities.

With these improvements in place, procedure creation would be a more robust process. System requirements would be maintained at an adequate frequency, improving the quality of all requirements and procedures dependent on them. Verification procedures would be written in a more uniform way, important information being more readily available to all participants. Test cases would be created more methodically, with a corrective mechanism in place to catch errors. Testers would be less isolated from the process and able to accumulate expertise more freely. Dry runs would be a more comprehensive method in catching errors in test procedures. These changes have been designed with a long-lasting impact in mind, so their deployment can initially be somewhat intrusive for the process.

Of course, some improvements can be prioritized over others. This can be done by placing them in an impact analysis matrix (table 2). Note: required effort and impact are ranked from 1 to 5. For required effort, a high number signifies a high effort but for impact and cost-effectiveness, a small number is desirable.

Improvement No.	Required effort	Impact	Cost-effectiveness
1.*	4	1 (2)	4 (8)
2.*	5	1 (2)	5 (10)
3.	3	3	9
4.	3	2	6
5.	2	2	4

* If these improvements are implemented together, look at the number without brackets. Separately implemented, the impact and cost-effectiveness are shown in brackets.

Table 2. An impact analysis matrix of the improvement recommendations.

Looking at the table, the least complicated decision is regarding improvement number 5, common terminology across teams. The cost-effectiveness is the best, tied with number 1 but without any strings attached. The difference in impact for numbers 1 and 2 comes from two effects: if only number 2 is implemented, the weight of poorly maintained system requirements is a factor, and if only number 1 is implemented, the effects of requirements maintenance is not controlled and can introduce errors into the procedure. Note: these figures are estimations, real effects can differ.

The effort required for number 3 is too high for the relatively low estimated impact. This improvement recommendation should be considered the lowest priority. The impact from number 4 is quite high, owing to the wide positive effect of improving information sharing. However, the effort required for gathering large amounts of information into a single hub is substantial. This improvement should be considered the fourth priority.

Improvement recommendation numbers 1, system requirements maintenance, and 2, dry running and test case maintenance, have a large estimated positive effect for the quality of verification procedures both for system requirements and software requirements. Human resources for their execution are also easy to allocate: requirements management (number 1) for product owners and system designers, test case management (number 2) for testers. Test leads can perform either task, serving as a fill-in for the task

that turns out to demand more resources. For these reasons, they should be considered first priority, but only if implemented in tandem.

Improvement number 4 would be a collective effort. For example, when using a tool, the tester could check to see if the resource hub (a website such as Confluence) has a guide for using that particular tool. The guide can then be assessed and corrected as needed or written if it is missing altogether. This work should be treated like any other work within a team and needs to be included in sprints. Similarly, improvement number 5 would be treated as a shared task. Both tasks can be done by creating a document for each and sharing the documents for each team.

All improvement recommendations should be put into effect as soon as they can be implemented through official channels, such as the backlog. From the perspective of the client of this thesis, this would mean the next project.

6 Conclusion

A patient monitoring device is a complex system whose quality assurance requires attention throughout the lifecycle of the product. At the very least, the product should comply with its requirements at launch, but the requirements and verification procedures also need to be formulated and maintained in a way that enables the implementation of a high-quality product. The process of formulating lower-level requirements and verifying their implementation is the area that was examined for this thesis. The verification of software is done either by freely testing it and creating documentation accordingly, or by testing the software following instructions in a verification procedure and providing inputs required in the procedure. The latter method is used more voluminously and was found to be affected by quality issues.

While the verification procedures are where the issues manifest themselves, root causes for them still had to be identified. Exploring the current process of creating verification procedures and the theoretical background for related activities proved to be the correct basis for any further work towards improvement recommendations. The improvement recommendations were written after every conceivable activity related to verification procedure writing was examined and information regarding each activity was gathered.

Many of the problems in verification procedure quality are linked to risk and time management. While it would be technically preferable to polish the procedures to perfection before executing them, it is nearly impossible to determine when the perfect condition of the procedure can be declared with confidence. Another example of the Pareto Principle, the amount of effort required to be put into an activity raises sharply after 80 percent of the results have been achieved, which typically happens at 20 percent of effort [18]. This can be seen when fixing a procedure: as the easiest-to-spot mistakes in the procedure are corrected first, it becomes increasingly difficult to find problems that need fixing. From this perspective, dry-running all procedures to some degree is preferable since a lot of mistakes can be found and fixed with little effort, while highly concentrated efforts dedicated to one singular procedure are inadvisable.

An inherent dilemma with improving any complex process is the issue of specialized improvements versus systematic improvements. The immediate effects of precise improvements are often desirable, but one runs the risk of turning the process into a patchwork of unforgiving procedural steps while people in the workforce have growing difficulties understanding the big picture. By contrast, broad modifications to the process and transferring responsibility to individuals will cause an initial dip in productiveness as changes are implemented and people go through a learning phase, afterwards settling it into a much more efficient process. The improvement recommendations given in this thesis slightly lean towards the latter scenario.

Realistic and concrete improvement recommendations with accompanying analysis and implementation guidelines were given in chapter 5.3, as was agreed. It is worth noting that these recommendations are based on estimations that were devised by qualitative means, rather than calculations. However, they are likely to contain valuable information for further analysis or development done by experts.

References

1 Engel, Avner. 2010. Verification, Validation and Testing of Engineered Systems. Wiley.

2 Fernandes, J.M. and Machado, R.J. 2016. Requirements in Engineering Projects. Springer International Publishing Switzerland.

3 Vogel, D. 2010. Medical Device Software Verification, Validation and Compliance. Artech House.

4 Koelsch, George. 2016. Requirements Writing for System Engineering. Apress, Berkeley, CA.

5 Kung-Kiu, Lau. 2004. Component-Based Software Development: Case Studies. World Scientific Publishing, Singapore.

6 Schmidt, Cristoph. 2016. Agile Software Development Teams. Springer International Publishing Switzerland.

7 McKenna, Dave. 2016. The Art of Scrum. Apress, USA.

8 Beck et al. 2001. Manifesto for Agile Software Development. Internet material. <agilemanifesto.org>. Read on 29.1.2018.

9 Measey, Peter. 2015. Agile Foundations: Principles, practices and frameworks. BCS Learning & Development limited.

10 General Principles of Software Validation; Final Guidance for Industry and FDA Staff. 2002. <https://www.fda.gov/MedicalDevices/ucm085281.htm#_Toc517237938>. Read on 11.1.2018.

11 Dion, Denise. 2011. Understanding Objective Evidence: (What It Is and What It Definitely Is Not). <http://www.eduquest.net/Advisories/EduQuest%20Advisory_ObjectiveEvidence.pdf>. November 2011. Read on 16.1.2018.

12 Software Testing Techniques/Methods. 2012. Internet material. <https://tfortesting.wordpress.com/2012/09/04/software-testing-techniquesmethods/>. Read on 9.1.2018.

13 Cheng, Michael. 2003. Medical Device Regulations: Global Overview and Guiding Principles. World Health Organization.

14 ISO 13485:2016. Internet material. <https://www.iso.org/standard/59752.html>. Read on 8.2.2018.

15 Gough, Janet & Nettleton, David. 2010. Managing the Documentation Maze: Answers to Questions You Didn't Even Know to Ask. Hoboken John Wiley & Sons.

16 Boyce, Carolyn & Neale, Palena. 2006. Conducting in-depth interviews: A Guide for Designing and Conducting In-Depth Interviews for Evaluation Input. Pathfinder International, USA.

17 Seale et al. 2013. Qualitative Research Practice. SAGE Publications.

18 Understanding the Pareto Principle (The 80/20 Rule). Internet material. <https://betterexplained.com/articles/understanding-the-pareto-principle-the-8020-rule/>. Read on 8.3.2018.

Interview frame

How long have you been a <interviewee's position>? How experienced are you in handling verification procedures or requirements?

Theme 1: The test procedure creation process

Can you describe the procedure creation process in detail? Is it much different for SyVP vs. SwVP?

Is the process always the same or are there some dependencies? Is there a phase where there are more dependencies than elsewhere?

Is there any confusion about the test procedure creation process?

Is the process comprehensively documented or is it known only to individuals? Who are involved in it?

Theme 2: Issues with the process

Would you say the process itself is poorly designed or are the issues minor/local?

Do some tools have a significant negative impact on the process?

Are there some unnecessarily time-consuming activities?

Are tasks delegated to the right people? Work instructions creation, for example.

Theme 3: Probing for improvement ideas

What do you think are most crucial areas that need fixes?

Do some standards or requirements feel outdated?

The procedure fixing process is quite heavy in official verification. Do you see any way it could be made lighter or is the damage irreversible at that point?

Are agile practices implemented in a satisfactory manner? Are there some problems with the development method itself?