UNIVERSITY
OF OULU

Generalizing Production Testing Operations for IoT Devices

University of Oulu
Information Processing Science
Master's Thesis
Kimmo Korhonen
2021

Abstract

A rapidly increasing number of new IoT products entering the market puts strain on the testing effort required to manufacture them. Every device needs to be tested at the manufacturing site before it can be shipped to the customer. This testing process during manufacturing is called Production Testing. If test automation systems running these tests are developed with a single system engineering approach, the number of test automation systems becomes unmaintainable. In addition to this, the development cost of such test automation system has to be covered by each product. Reusability of test automation and test assets is low when resources cannot be efficiently shared between products.

Existing solutions for generalizing testing effort from a single system approach to support multiple products were reviewed from the literature. Software Product Line Engineering is recognized as a possible solution, but its adoption requires organizational, economical, and technical changes. This thesis studied technical solutions for how test automation system could be developed to support the testing of multiple products.

Test automation system was designed based on existing literature, and two example products were used to mimic realistic IoT products. Work followed to define test requirements for two example products, implement tests, and execute them for the first example product. After tests passed for the first example product, they were executed for the second product. Test failures and evident problems were marked as variation points, and they were analysed. Test attributes that needed to be varied were recognized, and four different sources for that information were identified. Test logic was identified as one of the sources for attributes, and there was no need to variate it. Matching configuration was created for other sources: component, test, and hardware configuration. Tests were successfully executed for the second example product after introducing the variation via configuration files.

Prototype implementation succeeded in its goal to create production test automation system capable of testing two different example products using common test assets. Needed variation was introduced successfully through configuration files. This thesis shows that general test assets can be created for production testing, despite the fact that production testing is tightly coupled to the target hardware.

Future work continues by testing additional hardware platforms to reveal more variation points. This helps to develop production testing test automation to support a wider range of hardware platforms and components. Storing the hardware-specific configuration data to the device looks promising topic for further study.

Keywords
production system, production testing, test automation, embedded system testing

Supervisor
D.Sc. Professor Mika Mäntylä

Technical advisor
Sr. PE. Timo Räty

Foreword

I would like to thank my supervisor Professor Mika Mäntylä, for his guidance and valuable insights on presenting and improving my work. Bittium I would like to thank for the opportunity and support to turn one of the technical challenges into a topic for this thesis.

Special thanks I owe to Timo Räty, mentor and technical advisor of this work. Not just for his unparalleled expertise but also for pushing me to finish this work eventually. I highly appreciate your sense of humor and talent for delightful sarcasm. Eero Sankila, I would like to thank you for our collaboration during the technical phase of our work. Getting those example products up and running really helped my work.

I want to express my gratitude to my family, who supported and encouraged me to pursue my education.

Thank you, Piia and Hilta, for your patience and especially for the moments that bring me joy.

Marja-Leena, kiitos.

Contents

Terms

design tests	tests performed to determine the adequacy of the design of a particular type, style, or model of any unit of equipment, or its component parts, to meet its assigned ratings and to operate satisfactorily under normal service conditions or under any specified conditions. Such tests may also be used to demonstrate compliance with applicable standards of the industry. (IEEE Std C37.121-2012)
device under test (DUT)	device to be placed in a test *fixture* and tested. (IEEE Std 1450-1999)
end-user	the ultimate consumer of a finished product. (Merriam-Webster, 2020)
end-user software	software used in finished product when it ships out. Targeted for *end-users*.
fixture	an assembly that integrates a fixture frame, fixture enclosure and any other passive and active components or wiring necessary to interconnect the test system to the *unit-under-test* - UUT. (IEEE Std 1505-2010)
production test	a test conducted on every unit of equipment prior to shipment. (IEEE Std 1547.1-2005)
production tests	tests made for quality control by the manufacturer on every device or representative sample, or on required parts or materials to verify during production that the product meets the design specifications and applicable standards previously demonstrated during *design tests*. (IEEE Std 1505-2010)
production testing	refers to activities to carry out *production tests*.
production testing software (PTSW)	refers to the software which is used in *device under test* during production testing
system under test (SUT)	see *device under test*
test automation	The use of software to perform or support test activities, e.g., test management, test design, test execution, and results checking. (Chopra, 2018)
test environment	an environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test. (ISTQB Glossary, 2020)

1 Introduction

The rise of embedded systems in the world continues, as seen from the number of sold microcontroller units (MCU). Based on the annual McClean Report, there were 28.1 billion units sold in 2018, and the forecast for 2023 is 38.2 billion units. One factor for embedded systems growth is the Internet of Things (IoT) ("MCU Market History", 2019). At the same time, when vendors compete to push new IoT products to market, the pressure in production testing to support new hardware grows rapidly. Production testing refers to the process where every manufactured device is tested before shipment to the customer to verify that it meets its specification (IEEE Std 1505-2010). Production testing focuses on hardware, not on software.

Testing is a significant part of manufacturing costs. It increases the manufacturing time of each unit while the unit is being tested. Often there is also a need for specialized equipment and testing personnel. If production testing cannot be done cost-effectively, the cost of testing can exceed other manufacturing costs, which is hardly economically viable (Bailey, 2014). The production testing costs must be balanced with other manufacturing costs for the product to be economically viable. For this reason, there is continuous effort to lower the production testing costs while increasing the accuracy and performance of production tests. This can hardly be achieved by ad-hoc changes to the test environment, a more strategic approach is needed.

In traditional product development, the production testing cost is a fixed amount of total manufacturing cost. This means that for similar products A and B, the production testing cost are roughly the same. This is not an ideal model when vendors bring new IoT products to market at an increasing pace. The cost of production testing is eating a fixed share from the profits in every product. One obvious way to lower the production testing costs for each product is to share testing assets between the products. Software product line engineering (SPLE) was developed to improve and manage the asset sharing between products (Pohl, 2005).

SPLE literature mainly focuses on the development of the end-user product and software. Software needed during production testing greatly differs from the software targeted for end-users. The production testing software (PTSW) aims to control the device in a precise manner during testing and allow test automation to gather needed information from it to verify operational status. The intended user for PTSW is test automation, not human, and the difference of "user" reflects itself to the user interface and to software architecture overall.

The time that the device runs the PTSW in production is usually measured in seconds and minutes. Software is needed until all the production tests for a specific device are passed. After that, the PTSW is erased from the device, and the end-user software is flashed in. Flashing the end-user software usually prevents flashing the PTSW back on the device permanently via anti-rollback or by some other security mechanism. Even so that PTSW fulfills its intended purpose in minutes, it still has an important role for both manufacturers and for end-user. It protects both parties, the manufacturer, from shipping out faulty units and the end-user from receiving and using them.

Even though the importance of production testing software is recognized, its development seldomly is so structured when compared to other software projects. PTSW is targeted specifically for the hardware used in the product. This makes the implementation faster, but it also reduces the reusability and portability of the software. For this reason, each product usually develops its own PTSW and test automation system. This means that the amount of different test automation systems grows with each new product and the maintenance of these systems becomes more challenging over time.

This technical challenge has been recognized at Bittium, where new products enter production at a faster pace than before. Moving from the previously used single system approach towards software product line engineering is considered as a solution for the problem. However, the change affects multiple different areas and needs to be studied from multiple different perspectives: organizationally, economically, and technically. The purpose of this thesis is to study the technical feasibility by creating a proof of concept (PoC) of a test automation system that is capable of testing several different IoT products that are currently in development or in production. Organizational and economical changes related to adopting SPLE are not in the scope of this study.

The research problem and research question are stated for this thesis following way:

RESEARCH PROBLEM

> *Production test automation needs to be developed per product (fixed cost).*

RESEARCH QUESTION

> *How to create production test automation system capable of testing multiple products?*

This thesis aims to create a proof of concept as a prototype of the production test automation system, which is capable of testing multiple products using common test assets. Testing will be limited to features and peripherals found from all of the chosen products.

This thesis looks for existing knowledge from literature to find best practices moving from single system software towards software product line. Based on the existing knowledge, the new test automation system is designed. Design functions as theoretical background for the system and can be later revised. Lastly, the prototype of the designed test automation system is implemented and evaluated.

As a contribution, this thesis aims to provide practical information about the problems and their solutions, converting test automation targeted to a single system to support the testing of multiple products. Full adoption of SPLE is not feasible in the scope of this thesis, but the same concepts and techniques can be used, for example, to identify and address variation. The production testing context brings new insight into how single system solutions can be moved towards SPLE in the embedded systems domain, and this broadens the body of knowledge of adopting SPLE.

2 Literature

This section gives background information from literature related to this work. The literature presented in this section is aimed at common software development and testing purposes. The special nature of production testing is not discussed in this section. Instead it is presented separately at the Section 4 as there is not so much peer-reviewed literature available.

First quality factors of the product are reviewed through quality assurance and control. Then production testing and its high-level goals and objectives are reviewed based on literature. Next, test automation, the essential part of this work, is discussed by presenting its goals and objectives. Afterward, different testing types and different test automation approaches, including the concept of test automation framework, are presented. Lastly, the software product line engineering is discussed.

2.1 Quality Assurance and Control

Quality assurance focuses on the prevention of problems. It is part of the development process and it aims to improve and monitor the process (Tech, 2016). In order to improve the process, the problem areas must be identified, monitored, and detected. The practical activity of monitoring and detection is referred to as quality control. Quality control provides concrete data based on what the quality assurance can guide and improve the process overall. Quality assurance is described as a management process that defines what needs to be measured and the metrics used for monitoring. Quality control is a practical activity to find defects, faults, and failures. It provides the data and metrics for quality assurance. (Bourque & Fairley, 2014)

Chopra has summarised differences between quality assurance and quality control in his book Software Testing (2018) presented in Table 1.

Table 1. Differences between Quality Assurance and Quality Control summarised

Quality Assurance (QA)	**Quality Control (QC)**
It is process related.	It is product related.
It focuses on the process used to develop a product.	It focuses on testing of a product developed or a product under development.
It involves the quality of the processes.	It involves the quality of the products.
It is a preventive control.	It is a detective control.
Allegiance is to development.	Allegiance is not to development.

As we can see from Table 1 QA is focused on the processes required to develop the product while QC is focused on testing the product itself.

A guide to the project management body of knowledge (PMBOK®) states that quality assurance is about using project processes effectively. This involves following and meeting standards to assure stakeholders that the product will meet their needs, expectations, and requirements. PMBOK® also mentions that quality control is the process of monitoring and recording results of executing the detective activities in order to assess performance and ensure the project outputs are complete, correct, and meet customer expectations. (Project Management Institute, 2017)

Looking at quality assurance and control from a wider perspective they are often seen as part of a larger process, quality management. Quality management is collection of all processes that ensure that products, services, and implementations meet organizational quality objectives and achieve stakeholder satisfaction (Bourque & Fairley, 2014). The relationship of quality management, quality assurance and quality control is often presented as Euler diagram shown in Figure 1 (American Society for Quality website, 2020a).

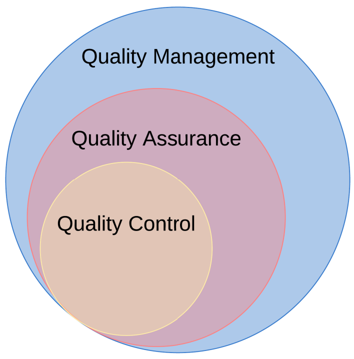


Figure 1. Relationship of QM, QA and QC.Adapted from *American Society for Quality* website.

As Figure 1 shows, quality assurance and control are part of the larger quality management process. Quality control is the oldest of quality processes, started around the 1920s following the rise of mass production. While manufacturing has become more complex over time, a more disciplined approach to ensure quality has been developed, namely quality assurance and quality management. (American Society for Quality website, 2020a)

Production testing is a quality control activity. It focuses on testing the product and is detective control by nature, see Table 1. The work on this thesis is more related to quality assurance, how to improve the process of production testing and allegiance in this work is to development.

2.2 Production Systems and Production Testing

To understand the context of production testing, we have to look at production systems. One of the most complete book of this topic is the Groover's *Automation, production systems, and computer-integrated manufacturing* (2015). In his book, Groover describes a production system as a collection of people, equipment, and procedures organized to perform the manufacturing operations. Production systems consist of two components, facilities, and manufacturing support systems. This can be seen from Figure 2.

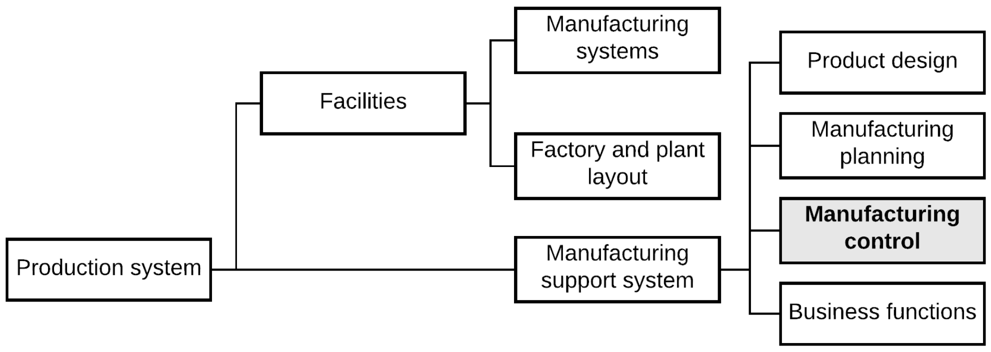


Figure 2. Production systems. Adapted from *Automation, production systems, and computer-integrated manufacturing*, by Groover (2015).

As shown in Figure 2 facilities refer to physical facilities to hold equipment needed for the actual manufacturing process to take place. The manufacturing support system includes all the procedures needed to carry out the manufacturing process. Particular interest in the scope of this thesis is manufacturing control which is concerned with managing and controlling the physical operations to implement manufacturing plans. Quality control is part of manufacturing control operations. Figure 3 shows the information flow in production system.

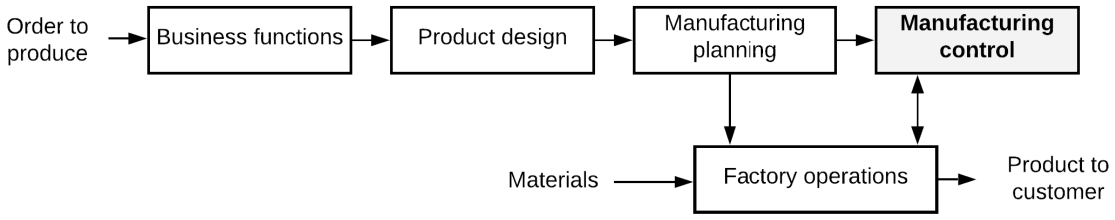


Figure 3. Information flow. Adapted from *Automation, production systems, and computer-integrated manufacturing*, by Groover (2015).

From Figure 3 it can be seen that information flows back and forth between factory operations and manufacturing control processes. Quality control is both monitoring the manufacturing process by testing the products and controlling the manufacturing process based on the results of the tests. (Groover, 2015)

Production testing refers to the activity of carrying out production tests. Pries & Quigley (2018) state that the purpose of production testing is to verify that the manufacturing lines are running correctly by assessing the correctness of the product.

Production tests and production testing are not widely recognized terms but they are used, for example in Institute of Electrical and Electronics Engineers (IEEE) standards. IEEE Std 1547.1-2005 defines a production test as a test that is conducted on every unit of equipment prior to shipment for the customer. The term was further defined in IEEE Std 1505-2010 as tests made for quality control by the manufacturer on every device to verify during production that the product meets its specification and standards. The full definition of the term according to IEEE standard presented in Section Terms: *production tests*.

As the standard states, production tests are quality control method, by which manufacturer verifies the quality of manufactured device before shipment to the customer. In summary, production testing is a quality control activity to ensure that product meets its specification. Quality control is part of manufacturing control operations which in turn is part of the manufacturing support system as seen in Figure 2.

Production testing should not be mixed to a style of testing often mentioned on software testing literature, especially when talking about web-based applications, *'testing in production'*. Testing in production refers to releasing new application version to a real environment for its intended use, referred to as production (Bose, 2020). The environment referred as production in the context of 'testing in production' usually means internet or local area network. In recent times some sources have been using the term 'production testing' as an alias to 'testing in production', which creates double meaning for the term. Previously cited web-page uses the term in sentence *'If possible, give users the option to participate in experimental production testing'* (Bose, 2020). Clearly, the production in this context refers to the internet rather than to the manufacturer's production facility. Production testing is discussed in more detail in Section 4.

2.3 Test Automation

Test automation is quite a common term that has broad definition and also different meaning for different interest groups. International Software Testing Qualifications Board (ISTQB) defines test automation as *'the use of software to perform or support test activities'* (ISTQB Glossary, 2020). So the definition of test automation depends on what are considered to be test activities. Anything that is required to be done in order to execute a specific test can be viewed as a test activity. Accompanied with the supporting activities, such as controlling and using additional instrumentation, the definition becomes quite broad, and it is hard to say what test automation is precisely.

Kinsbruner (2019) gives a more practical definition in his blog, saying that test automation is a method that leverages automation tools to control the execution of tests and to compare actual test results with predicted or expected results.

What is usually more important than the definition (of what test automation is) is the purpose of test automation. Goals and objectives are much more aligned with each other in literature. The goals of test automation extend the general goals for automation. Four commonly found goals for test automation are: (Fewster & Graham, 1999; Dustin, Rashka & Paul, 1999; Graham, Fewster & Copeland, 2012; Groover, 2015; Chopra, 2018; Encyclopedia Britannica, 2020)

- Speed up common or repetitive tasks to lift productivity
- Reduce mistakes or errors
- Improve safety for humans
- Free up humans so they can focus on higher-value work

All or several of the aforementioned goals are usually mentioned when test automation objectives are discussed in the literature. Other goals are also often mentioned, such as the ability to repeat test accurately and improve quality of the testing, but these can also be seen as a part of the aforementioned goals.

2.3.1 Testing Types

Test automation is tightly related to the type of testing it is harnessed into. Testing is commonly divided into functional and non-functional testing. Functional testing focuses on testing that software and/or hardware meets the predefined requirements. Non-functional testing focuses on testing aspects that are not functional by nature, such as security, reliability, and usability (Goericke, 2019). These can be viewed as product quality factors. Non-functional testing is geared towards customer's expectations for the product, while functional testing focuses more concretely on specifications and requirements (Chopra, 2018). Production testing, the main theme of this thesis, focuses on ensuring that manufactured units meet the requirements and applicable standards. As such activity production testing can be categorized to be functional testing, and for this reason, literature presented here focuses on functional testing.

Functional testing is commonly described to consist of the following levels of testing: unit testing, integration testing, system testing, and acceptance testing.

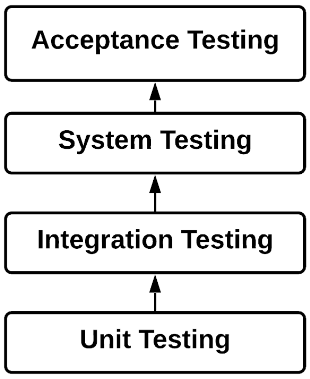


Figure 4. Different levels of testing. Adapted from *Software testing*, by Jorgensen (2013).

Above Figure 4 presents the different levels of testing and the direction from simpler tests at the bottom towards more complex ones at the top (Jorgensen, 2013).

Unit testing focuses on testing individual units or component of software, usually a single function or part of it. The literature recommends that every line of software should be covered by unit tests as they are the easiest tests to implement and execute (Martin, 2014). Automation on unit test level is very high, in practise, it can be seen as fully automated activity that is very well supported by tools and techniques (Meszaros, 2007).

The next level of functional testing is integration testing which focuses on testing logically connected software modules as a group. In other words the piece of code which are tested individually in unit testing are tested in interaction with each other to carry out more complex operations (Martin, 2014). At this level, automation is still very high while writing the tests is getting more complex compared to unit tests. Both unit and integration testing are usually done purely as software testing without the need of the target hardware. Tests can be executed in software development environment, which can drastically differ from the actual target hardware where the software will ultimately be running. (Broekman & Notenboom, 2008)

Embedded system literature differs from traditional software literature when it comes to integration testing. Embedded system literature often separates integration testing to software integration and hardware-software integration testing. Software integration tests are done without target hardware. In hardware-software integration tests, the software is executed in the target platform. Another similar term for software integration testing is software-in-loop (SIL). Hardware-software integration testing may be referred to as hardware-in-loop (HIL). (Broekman & Notenboom, 2008)

Following integration testing is the system testing which includes both the software and the hardware. In the early stages of product development, the actual hardware is not yet available, so it is often mimicked using development and macro boards. These boards are usually crafted to have components that will be used in the actual product, or if not yet available, components that are similar to those to be used (Broekman & Notenboom, 2008). The purpose of system testing is to execute software in the intended hardware and to monitor the functionality of both software and hardware. System testing focuses on end-to-end system operations, which means that functionality or service is tested from the beginning to the end to validate the functionality of tested solution (Rungta, 2021). Production testing in its essence, is form of system testing to be executed as part of the manufacturing process.

Following system testing comes acceptance testing. The purpose of acceptance testing is to demonstrate that all requirements are met, and product is suitable for its intended purpose. In other words product meets its functional purpose from the end-user perspective (Vance, 2013). However, acceptance tests are seldom only functional by nature. They usually include non-functional tests also, such as performance and usability tests. Acceptance testing is a larger testing concept that includes both functional and non-functional parts (Chopra, 2018).

2.3.2 Test Automation Pyramid

Test automation pyramid was first introduced by Cohn in his book *Succeeding with agile: software development using Scrum* (2013) to highlight the importance of test automation for agile development. Pyramid demonstrates different levels of automation, unit test, service test, and UI test level. The purpose of the pyramid was to give an idea of how an effective test strategy should be constructed.

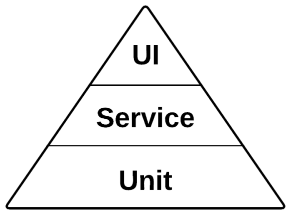


Figure 5. Test automation pyramid. Adapted from *Succeeding with agile: software development using Scrum*, by Cohn (2013).

The wider base of the test automation pyramid in Figure 5 represents the idea that the majority of the testing should be done at unit test level. This is due to the fact that both writing the tests and executing them at this level is easiest. User interface (UI) tests that are hardest to write and execute should be kept in a minimum while still covering requirements for testing. In the middle, at service level, tests are meant to test the interaction between software units. Interaction between software units provides service, which is the target of testing. Testing at UI level has several drawbacks: tests are brittle, the effort to write test cases is high, and they are time-consuming to execute. (Cohn, 2013)

Literature uses slightly different terms for testing levels based on the context. Testing can be viewed as entirely software-related matter, or it can include the hardware in addition to software which is the case in embedded software development.

The test automation pyramid was created to demonstrate the importance of testing in the Scrum development method, but the test automation concept has widened in other areas as well. Martin (2014) presented the pyramid in a way where the service level was further divided into component, integration, and system tests. Component and integration tests are usually where testing can still happen entirety in software form, but in system tests level, the actual target hardware, or simulation of it, will be used.

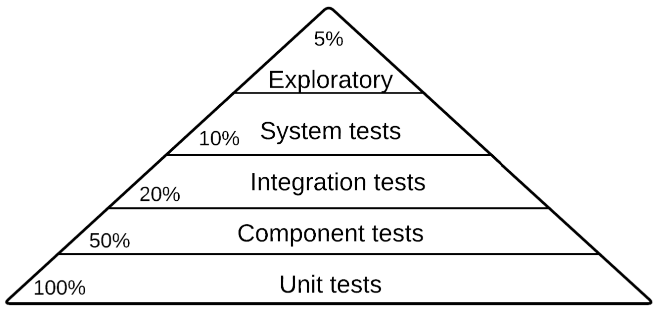


Figure 6. Test automation pyramid. Adapted from *The clean coder: a code of conduct for professional programmers*, by Martin (2014).

In Figure 6 it can be seen how Martin defines testing levels. He also uses term *exploratory tests* for the highest level in pyramid compared to *user interface testing* used by Cohn. On Martin's model, the exploratory tests involve human interaction and, as such, are not to be meant to be automated. Their purpose is to explore the system for unexpected behavior while confirming the expected behaviors. Also visible in Figure 6 are the percentages which demonstrate the target test coverage for each level. Unit tests should aim to cover 100% of the code, while only a small portion of the code should be tested at the exploratory testing level.

Production testing is placed at the system tests level in Figure 6. Also worth mentioning is that in the production testing context, the exploratory tests can be seen as a quality assurance activity rather than a quality control activity.

Myers, Sandler & Badgett (2012) described how the test automation pyramid levels refer to different development levels. Their work is presented in Figure 7 slightly modified, leaving out the software version information and adding the unit tests for code level.

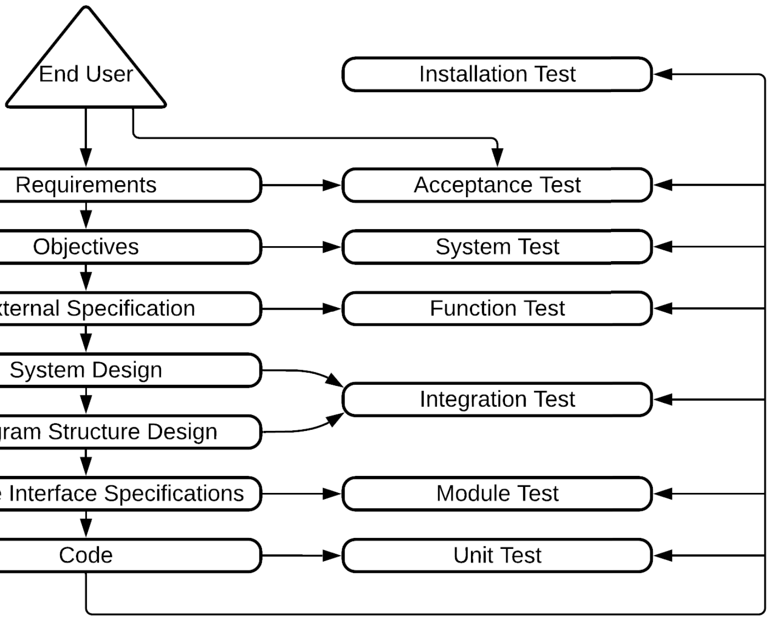


Figure 7. Correspondence between development and testing process. Adapted from *The art of software testing*, by Myers et al. (2012).

As can be seen from Figure 7 each level of testing and test automation tries to verify some aspect of the development process. Related to above Figure 7, accordance to technical requirements are verified in system tests level while accordance to user requirements are verified in acceptance tests level. We can look at Figure 7 in the context of production testing. The purpose of production testing is to ensure that manufactured devices fulfill the technical requirements, called objectives in Figure 7, so production testing can be viewed as system tests activity. The difference between system tests during development and production is the fact that at production phase the expected outcome of tests is known in detail. In the development phase, system tests are also used to gather information about the system in addition to actual testing purposes. The normal behavior of the system cannot be easily determined without actually monitoring the system. Based on the observed behavior during development, the limits for production testing can be defined. Myers et al. state in *The art of software testing* (2012) these two implications for system tests:

1. System testing is not limited to systems. If the product is a program, system testing is the process of attempting to demonstrate how the program, as a whole, fails to meet its objectives.
2. System testing, by definition, is impossible if there is no set of written, measurable objectives for the product.

The second implication is very important in the system testing context, which production testing is part of. If there were no objectives for test results, it would not be possible to conduct system testing in a meaningful way. In order to carry out system testing effectively, the information of what to measure, how to measure, and what the limits are has to be defined in detail beforehand. (Myers et al., 2012)

2.3.3 Test Automation Frameworks

Test automation framework is not by itself a component of the testing process. It is a combination of concepts, guidelines, and tools to bind software libraries, modules, and other tools to provide the core functionality of the framework (Techopedia, 2021). To understand the purpose of the test automation framework, it is helpful to review it as an extension of the generic software testing framework. Alsmadi (2012) has published model of generic software testing framework, shown in Figure 8.

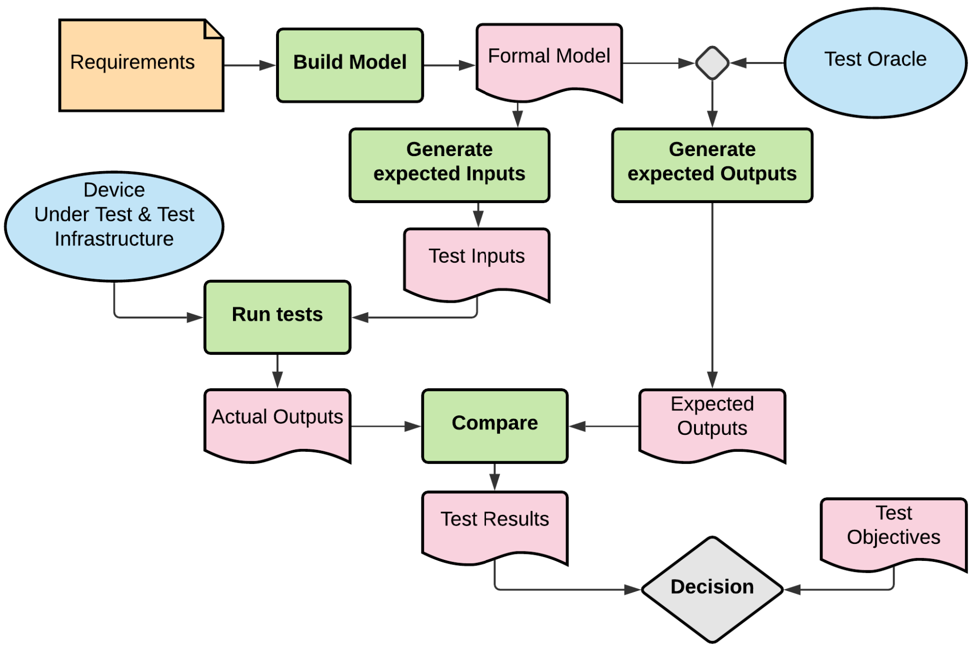


Figure 8. Generic software testing framework. Adapted from *Advanced automated software testing*, by Alsmadi (2012).

In Figure 8 active processes are presented as squares, and document symbols present artifacts which are created by the process. Testing starts with the creation of the test model. The model can be constructed formally from the requirements, or it can be created by gathering information from the actual application to be tested. Test input is generated from the model to be passed to the test case (Alsmadi, 2012). The test oracle in Figure 8 refers to a mechanism of determining what the expected output from the test is. Test oracle usually uses derived information from existing artifacts, such as requirements, documentation, earlier results, and log files. Test oracle can also use implied information and assumptions to complete the derived information (ISTQB Glossary, 2020).

When inputs and expected outputs are known, the tests are executed using the existing test infrastructure. Output from the test execution is gathered, and it is compared to the expected output to conclude the test result. Test results are then compared to test objectives to decide if objectives are fulfilled. At the decision phase, there can be, for example decision to modify test model, generate more tests, estimate reliability, or to stop testing. (Alsmadi, 2012)

Test automation framework is based on the principles of the generic software testing framework. Each phase of testing usually uses software tools to ease the workload of humans, but the aim is not in fully automating the process from start to finish. Instead, automation focuses on areas where it is more time and cost-effective. Time and effort to build test model and to design test algorithms by humans is usually markable lower than trying to automate those phases fully. On the other hand, achieving the same efficiency, accuracy, and reliability to execute test cases, collect results, and report findings is virtually impossible for humans. For this reason, test automation framework tries to reduce human involvement to a practical minimum rather than replacing humans altogether. (Alsmadi, 2012)

As noted before framework also consist of concepts and guidelines. These aim to provide theoretical and practical knowledge primarily for the test automation design and implementation phase. Concepts and guidelines help to base design decisions on domain knowledge and to use known workable solutions. Basing decisions on domain knowledge helps to create a more structured and systematic way to design test cases, decide what tools to use and how to carry out different processes. (Ramos, 2021)

2.4 Embedded System

Embedded system is a very broad term that can be used to describe systems from wireless headphones to satellites. The common feature for all embedded systems is that they interact with the real physical world by controlling specific hardware. (Broekman & Notenboom, 2008)

An Embedded system is typically a small-scale computer system that is part of a machine or larger system. They are typically real-time systems dedicated to perform specific tasks. Embedded systems are getting increasingly used in everyday life, and they can be found in digital watches, refrigerator, smart televisions, cars, and e-bikes. As they are usually part of a larger machine or system, they commonly need to be small in size, low in cost, and have relatively low power consumption. (Xiao, 2018)

Xiao (2018) illustrates typical embedded system in a diagram shown in Figure 9.

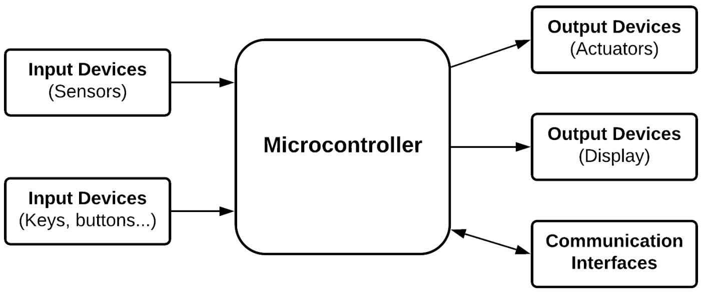


Figure 9. Schematic diagram of a typical embedded system. Adapted from *Designing embedded systems and the internet of things (IoT) with the ARM® Mbed™*, by Xiao (2018).

Figure 9 gives good high-level abstraction of a typical embedded system. The device gathers input from various sources, process them, and communicates it through output devices and communication interface. Broekman & Notenboom (2008) present similar generic scheme for embedded system in more detail. This is shown in Figure 10.

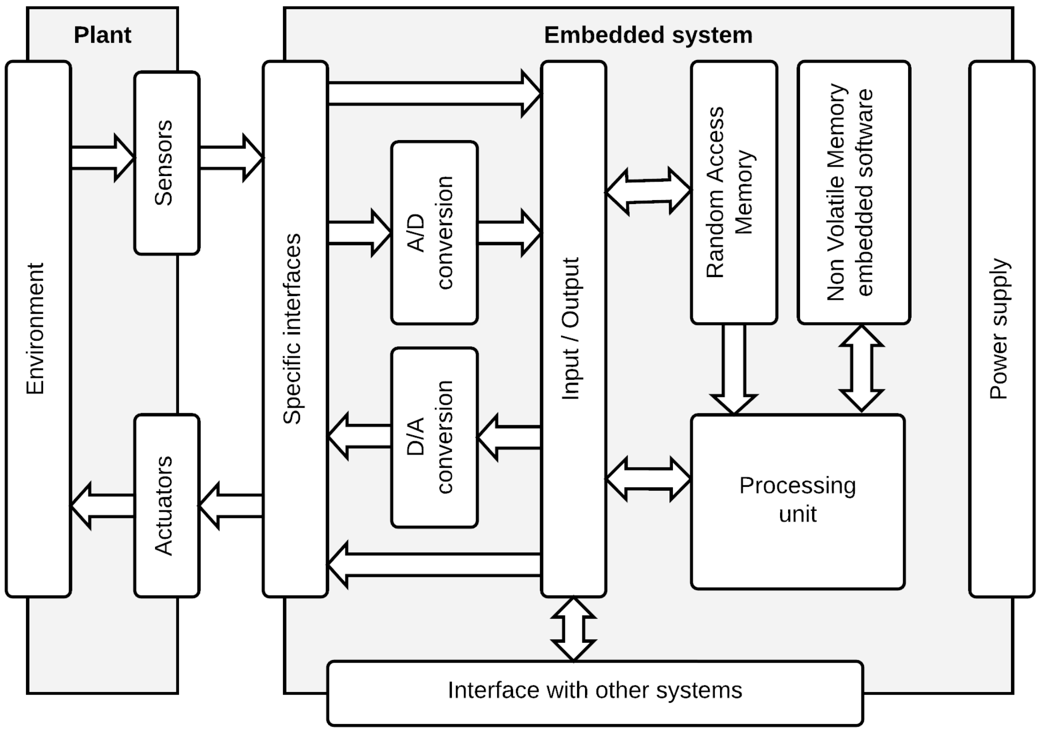


Figure 10. Generic scheme of an embedded system. Adapted from *Testing embedded software*, Broekman & Notenboom (2008).

Embedded system interacts with the real world through sensors (input) and actuators (output). The real world in Figure 10 is referred to as environment. The combination of environment, sensors, and actuators is commonly referred in embedded system literature as plant. The Embedded system has specific interfaces to handle input from sensors. Sensor values need to be converted from analog format, for example, from millivolts, to digital format. The conversion is done in analog to digital converter (A/D). Similarly, the actuators are controlled in an analog format, so the digital values sent by the processing unit are passed through digital to analog converter (D/A). The processing unit can operate with inputs and outputs directly or via random access memory (RAM). RAM memory is volatile, so its content is lost when the power to the component is switched off. For this reason, embedded systems also have non-volatile memory (NVM) where the actual embedded software is stored. Embedded system communicates with other systems through input/output (IO) interface. (Broekman & Notenboom, 2008)

2.5 Embedded System Testing

Embedded system testing differs from software testing by the fact that hardware is part of the testing process.

In software testing, system testing refers to testing where the whole software product is tested. It was briefly discussed in Section 2.3.1. In embedded system testing, the actual product, including both the software and hardware, is tested. At this level of testing, the software functions are executed within the actual target hardware without the use of mock interfaces used in unit test and integration testing phases (Martin, 2014; Pries & Quigley, 2018).

2.6 Software Product Line Engineering - SPLE

Software product line engineering emerged at the beginning of the millennium as a paradigm to increase the productivity of IT-related industries, enabling them to handle the diversity and to reduce the time to market. The key idea is the realization that most software systems are not new in their application domain, they share more commonalities than uniqueness. (Sugumaran, Park & Kang, 2006)

Many companies have adopted SPLE in pursuit of its benefits. While configurability and reduced time to market are important for businesses, the economical benefits are the main driving force. SPLE supports large-scale reuse during development which can be as much as 90% of the overall software. Even if reuse would be around 50% it would be very desirable as reuse is more cost-effective by order of magnitude compared to traditional software development. (Linden, Schmid & Rommes, 2010)

One of the biggest challenges in SPLE adoption is the up-front investment to create the software product line. Adoption requires that reusable software assets are created throughout the software development process. Such undertaking usually requires organizational changes. This increases the initial effort of SPLE. However, there are several studies that show that the initial investment hits the break-even point with traditional single system development around third software system, after which point each system costs less to develop (Pohl, 2005). Figure 11 shows the accumulated costs needed to develop *n* systems.

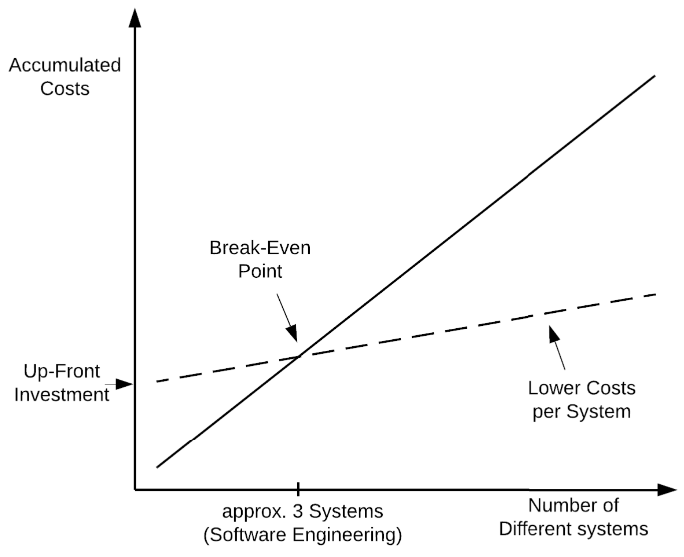


Figure 11. Cost of developing n kinds of systems as single systems compared to product line engineering. Adapted from *Software product line engineering: foundations, principles, and techniques*, by Pohl (2005).

Solid line in Figure 11 shows development cost of single-system approach. Accumulated development costs increase steadily with each new system as their development costs are roughly the same. The dashed line shows the development cost of the product line engineering approach. It has a significantly higher initial cost compared to the single-system approach, but the break-even point is usually reached at a third developed product based on Pohl (2005) and Linden et al. (2010). After this, each new system developed with software product line engineering has a lower cost compared to single system development.

Pohl (2005) mentions that SPLE is not very well suited for novel systems. The initial cost of SPLE development is much higher compared to single system development. If SPLE is only used to develop one product, the costs can be multiple times higher than if product had been developed as a single system. The benefits of SPLE are only achieved when multiple products from the same platform are developed. This is the reason why SPLE is often adopted when there is already at least one product developed, and the product is known to be economically viable. (Pohl, 2005)

3 Research Problem and Methodology

The thesis aims to create a prototype of a test automation system capable of testing multiple different products on different hardware platforms. As the aim of the thesis is the creation of an IT artifact in the form of prototype, the design science research method (DSR) was selected as a research method. Hevner, March, Park & Ram describe in their article *Design Science In Information Systems Research* (2004) that knowledge and understanding of a problem domain are achieved in building and application of the designed artifact. Artifact in this study will be the prototype system, and its design is the main focus in this study. The creation of to prototype system aims to answer the research question:

> *How to create production test automation system capable of testing multiple products?*

Prior literature gives guidelines for common steps needed to build test automation systems, but it still leaves few questions open. How to adapt the test automation system for production testing purposes and how to ensure that the system is capable of testing multiple products? These open questions are the focal point of the design, and based on earlier experiences, they are not trivial to solve. The research problem was raised by Bittium, and it was discussed that solution in the form of a working prototype system would be preferred. For this reason, the test automation system needs to be designed and operationalized as a prototype.

Hevner & Chatterjee (2012) state that design science research (DSR) is a research paradigm in which a designer answers questions relevant to human problems via creation of innovative artifacts. The creation of artifacts and the artifacts themselves contribute to the body of knowledge. They also state that artifacts are both useful and fundamental in understanding the problem (Hevner & Chatterjee, 2012). Given the objective of this thesis, the DSR is well suited as the research method.

3.1 Research Method

Hevner et al. (2004) present in their article the complementary research cycles between design science and behavioral science to address fundamental problems in the application of information technology. The behavioral science paradigm seeks to develop and justify theories that explain organizational and human phenomena surrounding the analysis, design, implementation, and use of information systems. The behavioral science paradigm has much in common with natural science research methods. Theories developed inform researchers and practitioners of the interactions among people, technology, and organizations that must be managed if an information system is to improve the effectiveness and efficiency of an organization. (Hevner & Chatterjee, 2012)

Contrary to behavioral science, the design science paradigm has its roots in engineering and the sciences of artificial (Simon, 2019). It is often referred to as a problem-solving paradigm that seeks to create innovations that define ideas, practices, technical capabilities, and products. Through these innovations, analysis, design, implementation, and use of information systems can be effectively and efficiently accomplished. (Hevner & Chatterjee, 2012) Complementary nature of design science and behavioral science research can be seen in Figure 12.

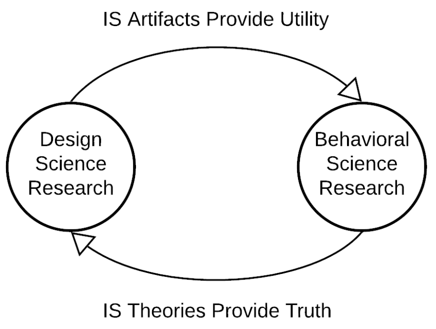

IS Artifacts Provide Utility
Design Science Research
Behavioral Science Research
IS Theories Provide Truth

Figure 12. Complementary nature of design science and behavioral science research. Adapted from *Design research in information systems: theory and practice*, by Hevner & Chatterjee (2012).

Figure 12 demonstrates that design science research mainly provides a utility that can be used to develop theories and laws in behavioral science research. On the other hand, design science research uses IS theories to establish grounding for the research.

Hevner & Chatterjee (2012) highlight the idea that technology and behavior are inseparable both in information systems and in IS research. Truth in the form of justified theory and utility in the form of effective artifacts are two sides of the same coin. Similarly, the practical relevance of the research result and rigor of the research performed should be equally valued (Hevner & Chatterjee, 2012).

The purpose of DSR artifacts is to provide a utility that can be used as a tool to develop a theory. Information systems research framework presented by Hevner et al. (2004) shows how the DSR is positioned in relation to the problem domain (environment) and to the body of knowledge.

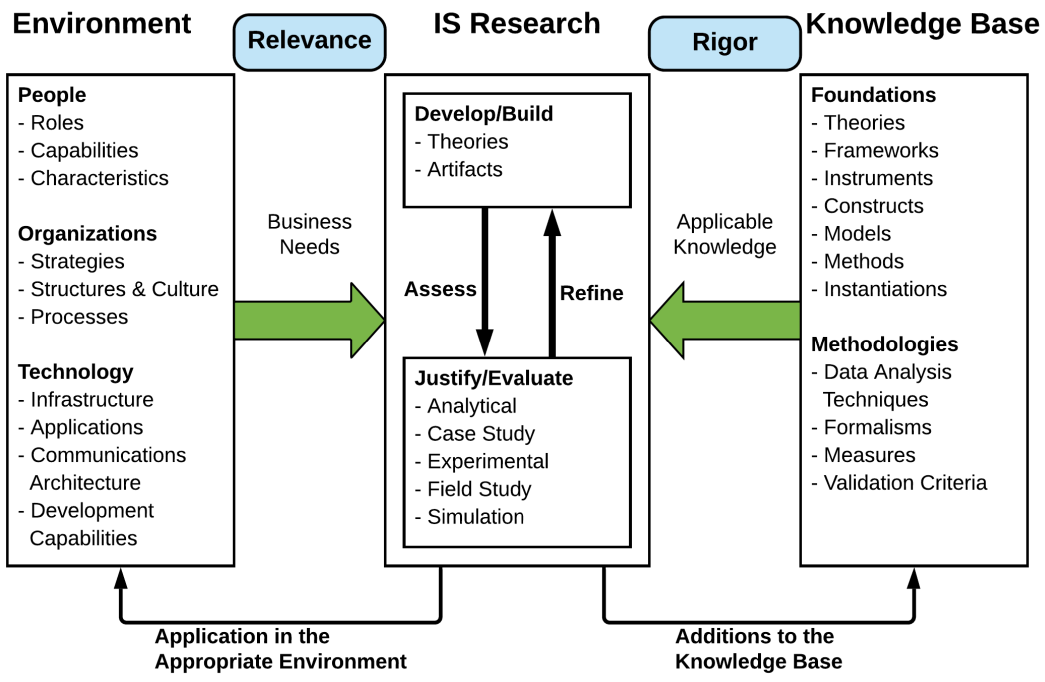


Figure 13. Information Systems Research Framework. Adapted from *Design Science In Information Systems Research*, by Hevner et al. (2004).

Information systems research framework in Figure 13 shows how the research draws knowledge from the knowledge base (behavioral science). The problem domain for research is coming from the environment. The research contributes to the environment in the form of artifacts to solve practical problems and to the knowledge base as additions to existing knowledge.

Purao (2002) introduced different maturity levels for DSR artifacts which Gregor & Hevner (2013) has further refined. Gregor & Hevner described maturity levels in correlation to research artifacts which can be seen from Table 2.

Table 2. Design Science Research Contribution Types

	Contribution Types	**Example Artifacts**
More abstract, complete, and mature knowledge	Level 3. Well-developed design theory about embedded phenomena	Design theories (mid-range and grand theories)
↕ ↕ ↕ ↕	Level 2. Nascent design theory-knowledge as operational principles/architecture	Constructs, methods, models, design principles, technological rules.
More specific, limited, and less mature knowledge	Level 1. Situated implementation of artifact	Instantiations (software products or implemented processes)

From the model developed by Purao and refined by Gregor & Hevner it can be seen that instantiations are placed at the lowest maturity level of artifacts. Maturity level 1 artifact contribute mainly to the environment by providing solutions to practical problems, and they have somewhat limited contributions to the knowledge base. Maturity level 2 artifacts provide more generic solutions to the problem, and their contribution to the knowledge base is higher as the artifact usually is applicable to new problems or problem domains. Maturity level 3 artifacts are well-developed design theories of the phenomenon under study (Gregor & Hevner, 2013).

Even though instantiations are at the lowest maturity level of artifacts, they are fundamental in the IT field and deemed profound and necessary. Kuechler & Vaishnavi (2015) give example from aeronautical engineering. Prototypes preceded the full understanding of physics involved by decades. In fact, the aeronautical engineering knowledge base was built almost exclusively through prototypes that provided the required utility to develop a complete theory and understanding of the phenomenon (Kuechler & Vaishnavi, 2015).

March & Smith (1995) point out that instantiation sometimes precedes a complete articulation of vocabulary, models and theories that it embodies. They see instantiations as an operationalized construct, models, and methods (March & Smith, 1995). The utility which instantiations provide is necessary for DSR. They enable research to take place in problem domains that are not yet well understood.

Vaishnavi, Kuechler & Petter (2019) presented adaptation of DSR knowledge contribution framework initially published by Gregor & Hevner (2013). Framework helps to place the contribution of the research in correlation to both solution and problem domain maturity. Framework is presented in Figure 14.

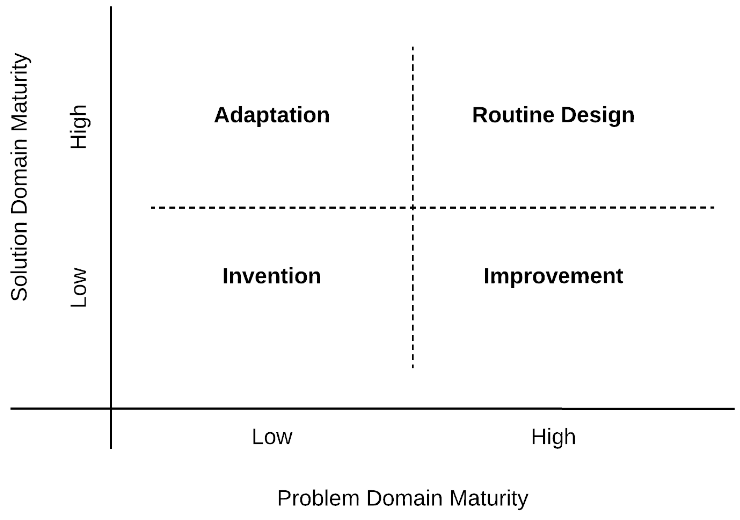


Figure 14. DSR Knowledge Contribution Framework. Adapted from *Design Science Research in Information Systems*, by Gregor & Hevner (2013).

As can be seen from Figure 14, Invention is placed in the quadrant in which the solution domain, as well as the problem domain maturity, is low. In other words, inventions provide new solutions for new problems. Adaptation is placed in the quadrant where the solution domain maturity is high, but the problem domain maturity is low. Adaptations are a non-trivial adaptation of known solutions for new problems. Improvements are placed in the quadrant where solution domain maturity is low, but the problem domain maturity is high. Improvements provide new solutions to known problems. The final quadrant where both solution domain, as well as the problem domain maturity, is high is regarded as Routine Design which is rarely considered as research contribution. (Vaishnavi et al., 2019)

Using the DSR knowledge contribution framework, the contribution of this thesis will be adaptation. The purpose is to use known solution, in the form of test automation, to solve new problem of executing production testing for multiple different embedded products. This work is considered adaptation rather than routine design because based on earlier implementations of test automation, the support for multiple products has not been achieved, which indicates that the task is non-trivial.

3.2 Design Science Research Guidelines

Hevner et al. (2004) present seven guidelines for conducting DSR. Following is the list of the guidelines, a summary of the author's explanation, and how the guideline is taken into account in this thesis. Guidelines are repeated in the evaluation section, where guidelines are reviewed in the context of results provided by design and evaluation.

Guideline 1. Design as an Artifact.

Design science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation. (Hevner et al., 2004)

The goal of this thesis is to design and implement a prototype of production test automation system capable of testing multiple products using the same test assets. The prototype system is instantiation.

Guideline 2. Problem relevance

The objective of design science research is to develop technology-based solutions to important and relevant business problems. (Hevner et al., 2004)

Amount of embedded systems is increasing in the form of new IoT products very rapidly. Developing and maintaining production testing support for new products is already recognized as a growing technical challenge at Bittium. It is assumed that other product manufacturers are experiencing similar difficulties.

Guideline 3. Design evaluation

The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. (Hevner et al., 2004)

Evaluation of prototype is done by using it in its intended purpose. Two example products will be created using different microcontrollers that are used in actual products. Sensors and other peripherals are added to example products to mimic actual products. A realistic production test suite covering required parts of the hardware will be executed for both products. Results for both products are compared against the test requirements.

Guideline 4. Research contributions

Effective design science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies. (Hevner et al., 2004)

Design principles and rationale behind design solutions are documented for the prototype. These are most likely valuable for others as example solutions to similar problems. The prototype system itself provides utility back to the practice.

Guideline 5. Research rigor

Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact. (Hevner et al., 2004)

The research will be conducted using design science methodology and following its guidelines.

Guideline 6. Design as a search process

The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment. (Hevner et al., 2004)

Design solutions were searched from existing literature, and their applicability will be tested in practice. Problems are documented, and alternative solutions presented.

Guideline 7. Communication of research

Design science research must be presented effectively to both technology-oriented and management-oriented audiences. (Hevner et al., 2004)

This thesis presents the creation of the prototype system for production testing purposes. Design and evaluation are present in this thesis. The design of the prototype system and the prototype itself are aimed at the technology-oriented audience. Description of the problem domain and its economical impacts are aimed at a management-oriented audience.

4 Overview To Production Testing Domain

This section gives a short domain overview of production testing, which is the problem domain in the context of this thesis. First is described the purpose of production testing and some constraints that apply to it. Different parts of the production testing environment are introduced, covering both hardware and software components. Lastly, the cost of production testing is viewed covering both the development and the operational phases of production testing. Domain overview gives background information about the topic and context to the research problem.

The overview presented in this section relies partly on lessons learned from earlier experiences. Information related to manufacturing and production testing is often considered business-sensitive information, which makes describing it somewhat challenging even in a general sense. The topic needs to be discussed in a general sense to avoid falling under a non-disclosure agreement. If references from literature cannot be found for the discussed topic, information is presented based on the author's own experience or common domain knowledge. This will be highlighted in the text, but the reader should be critical of such information.

4.1 Purpose of production testing

The purpose of production testing is to ensure that manufactured units meet the specification and applicable standards before it is shipped to the customer (IEEE Std 1505-2010, 2010). Production testing can be viewed as a quality control activity carried out during the manufacturing process. The aim is to catch defective units before they are sent to the customer.

There are many reasons why shipping out defective units is harmful. The most important ones are that it is expensive to fix units after the shipment, and also it has a negative impact on the brand image, which will cause loss of revenue. (American Society for Quality website, 2020b)

Defect prevention and inspection have costs of their own so economically good balance has to be found between testing and the risk of producing defective units. Gupta & Starr (2014) have summarised the cost of quality based on four different factors:

- cost of inspection
- cost of prevention
- cost of failure
- total cost

Inspection refers to examining if the product conforms to agreed standards. Inspection is commonly done by sampling instead of inspection of all units.

Prevention involves using conscious strategies to reduce the production of defective products applied to all manufactured units.

Cost of failure includes the costs of fixing or replacing the defective unit and possible recall of defective units.

The total cost presents the sum of the three aforementioned factors. Figure 15 presents the previous costs and how they are related to the percentage of defective units.

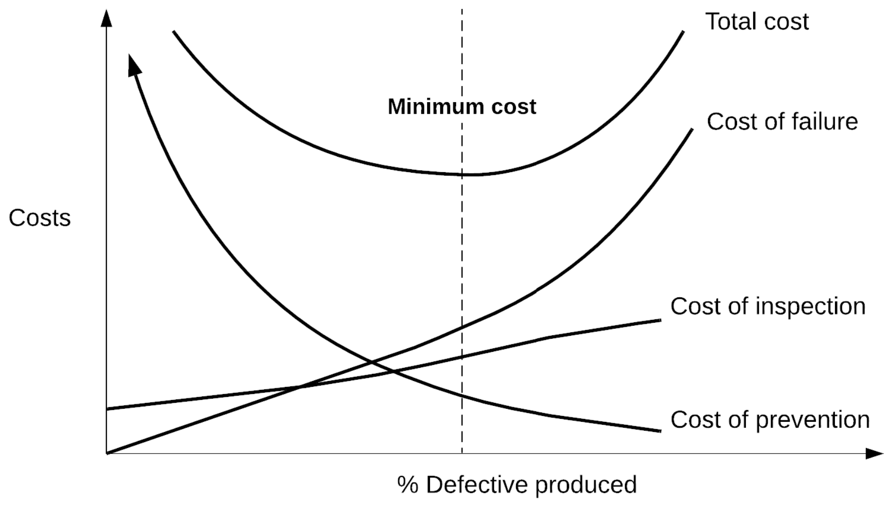


Figure 15. Percentage defectives produced versus costs. Adapted from *Production and operations management systems*, by Gupta & Starr (2014).

Figure 15 shows that prevention cost and cost of failure are the two most important factors of the total cost structure. The cost of inspection grows almost linearly together with the percentage of defective units. The cost increases because a larger amount of units needs to be sampled when the number of defective units increases. Cost of failure tends to grow linearly at first but accelerates rapidly when the amount of defectives increases. One important factor in this is the reputation of the product among customers, which usually escalates the loss of business. On the other hand, the cost of prevention also grows rapidly when the amount of defectives is decreased. It is possible that prevention costs become larger than the cost of failure would be otherwise. (Gupta & Starr, 2014)

Based on the aforementioned cost of quality structure, the cost of production testing, part of prevention effort, has a significant economical impact. To examine what is included in the cost of prevention (Figure 15), we observe this through experiences gathered at Bittium. Testing takes time which increases the production time of a single unit, which decreases the number of units produced daily. Testing also requires equipment and instruments in addition to manufacturing equipment. Equipment has cost of their own, and they also require physical space on the factory floor, which adds to the expenses due to space requirements. Production testing is seldom fully automated, for example, the device to be tested needs to be manually placed to test fixture and removed from it, so personnel is needed to carry out the testing.

The cost of production testing at Bittium can be roughly estimated as the sum of time, equipment and instruments, space, and personnel. However, this only covers operational costs during the production. There is also the development cost for the production testing software, production tests, and test automation.

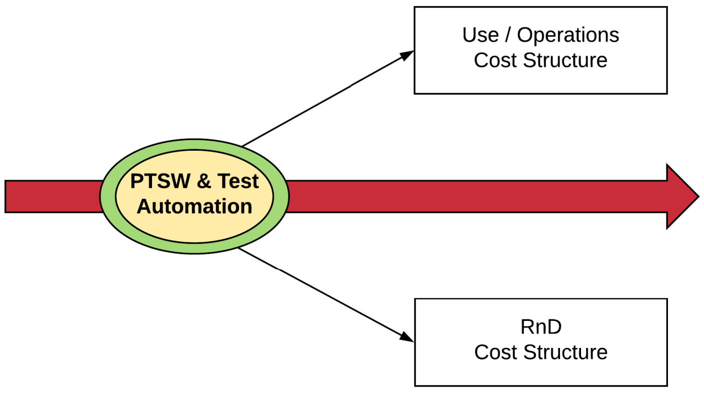


Figure 16. Cost structure for PTSW and test automation

Figure 16 illustrates that the development of PTSW and test automation has two separate cost structures at Bittium. Research and development costs before the product enter mass production and operational costs during it. In practice, development and operational costs overlap each other as the operational production testing starts already during prototype production. Development of the PTSW and test automation are still in progress during early prototype batches. Figure 17 shows how development, operational usage, and maintenance are commonly linked in companies.

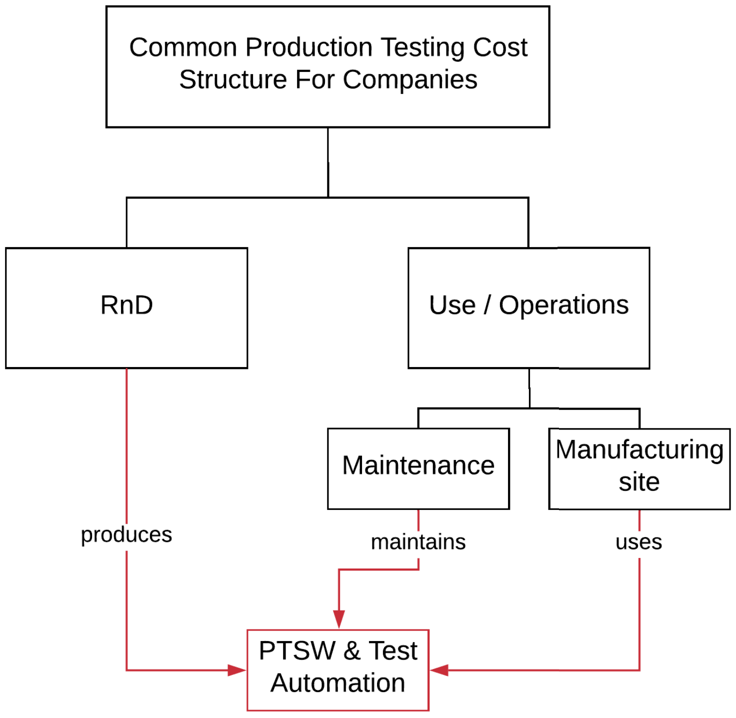


Figure 17. PTSW and production test automation linked to organizational units.

Figure 17 is abstraction of the basic costs of production test system: development, usage and maintenance. How the costs are divided between organizational units varies between companies. The purpose is to highlight that the total cost of production testing also includes other than operational costs.

4.2 Differences to normal software development

There is no common standard for production test software, so describing it in a general sense is not possible. In order to highlight some of the key differences between end-user software and production test software, this chapter uses production test software developed at Bittium as an example.

While all the parts of the production testing domain can be viewed as normal software development, there are some notable changes. One of the key differences is the concept of the end-user. Commonly software is intended to be used by humans, and the user interface of how humans can interact with the software and underlying hardware is a fundamental part of the software development. In production testing, the intended end-user is test automation. There is a deliberate intention to keep the human interaction at a minimum, and it has a profound impact, for example, on how the interfaces are designed. In the production testing domain, the intended user is often highlighted by the use of terms 'end-user software' and 'production test software'. End-user software refers to the software which is installed on the finished product, and production test software (PTSW) refers to software that is used during the production process.

The software quality criteria described in ISO 25010:2011 and its predecessor ISO 9126 can also be used to evaluate the quality of the production testing software. ISO 25010:2011 defines six main quality characteristics.

- functionality
- reliability
- usability
- efficiency
- maintainability
- portability

Priority of quality factors can differ greatly between end-user and production test software. For example, usability usually has high priority in the end-user software, but in production test software developed at Bittium, it often has the lowest priority. PTSW is targeted for automation, not humans. Production testing commonly prioritizes reliability, functionality, and efficiency as its main quality factors.

Another distinct difference between production testing and traditional software testing is the amount of hardware required to test the manufactured device in different stages of production. Based on experiences at Bittium, testing starts early in production before the final assembly of mechanics is started, so there is often a need for specialized fixtures or adapters so that the test environment carrying out the test can be connected to the device under test (DUT). DUT is often just a circuit board in the early stage of production testing before any additional hardware and mechanics are assembled. In addition, there is often a need for additional instruments so that needed parameters can be measured from the device during testing. These can be a controllable power source, radio frequency (RF) signal generator, and various types of sensors.

To carry out production tests in practice requires software both to control the test environment and the DUT. The software required to control the test environment and instruments is part of test automation and is described in more detail in later sections.

4.3 Production testing hardware and software

This section gives an overall view of the hardware and software that are commonly needed to carry out production testing at Bittium.

4.3.1 Hardware

To communicate with the device under test (DUT) fixture (or ”adapter”) is needed to connect the DUT to the test environment. Power supply and other instruments are commonly attached to the fixture. The fixture is connected to the test environment that executes the production tests. Production tests can utilize the instruments that are connected to the fixture as well as the DUT itself. Figure 18 shows common hardware configuration to carry out tests at Bittium.

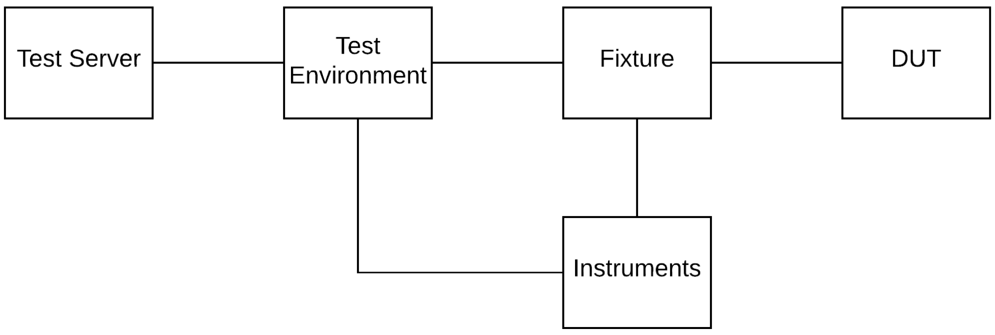


Figure 18. Production testing hardware overview

In the Figure 18 DUT is the unit to be tested. It runs the production testing software (PTSW) that allows the test environment to control and monitor its behavior. PTSW enables the test environment to access hardware components directly in order to control or to access data from them. These features are hardly ever present in end-user software because they pose a high security risk. In order for the test environment to communicate with the DUT fixture is needed to provide required connections and wiring. Additional instrumentation can be attached to the fixture, power supply, radio frequency (RF) generator, and different types of sensors. The test environment executes the test suite that contains individual test cases. Each test case can control and gather data from the DUT by sending commands to it via the communication channel provided by the fixture. The test environment analyses the result of the test and gives a verdict for each test case executed. Test execution is logged and reported during execution. Test results are sent to test server.

4.3.2 Software

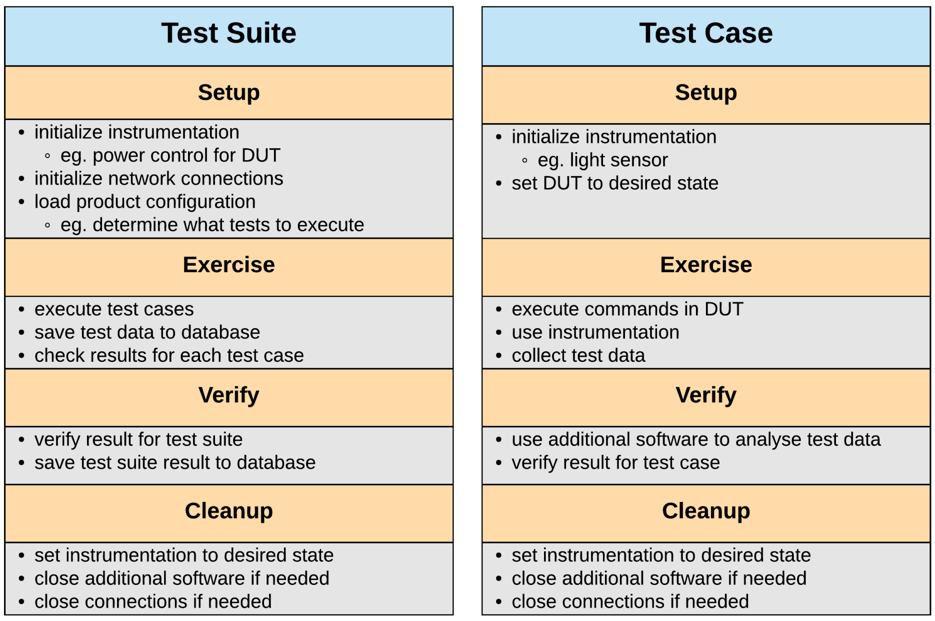
All the hardware components shown in Figure 18 are also running some software, with the possible exception of the fixture. The fixture usually has simple commands, for example, to open and close mechanical jaws. These commands can be sent via a software interface, but they can also be controlled electronically without the need for additional software.

Instrumentation is usually running vendor’s proprietary software, but there is normally some standard interface that allows controlling the instrumentation programmatically.

DUT is running the PTSW, which allows the test environment to access many internal features that are not normally exposed in the end-user software. For example, to read restricted memory addresses or to change register values. PTSW does not normally do anything unless it is specifically instructed. PTSW provides access to the hardware through a command interface. The purpose of the PTSW software is to allow the test environment to control the DUT and to collect data from it.

In the production test automation context, the test environment usually is the most complex software-wise based on the author's own experience. It is running an operating system for the common tasks, automation software that takes care of execution of the test suite, software that controls the instrumentation attached to test fixture, and possibly software that analyses the gathered data to produce pass-fail verdict for each test case. It also has networking capabilities to store test reports in the test server. The purpose of the test environment is to execute a set of production tests, the test suite, and to report the test results.

Test automation software is needed to execute the test suite. The test suite is a collection of individual test cases. Each test case can control the DUT and instruments that are connected to the test fixture. Test cases can also control additional instrumentation and gather measurement data from them. After the needed data from the test case is collected, it can be passed to an analysis program that produces a pass-fail verdict for the test case. Figure 19 shows how the work between the test suite and the test case is normally divided in production testing domain.



Test Suite	Test Case
Setup	**Setup**
• initialize instrumentation ◦ eg. power control for DUT • initialize network connections • load product configuration ◦ eg. determine what tests to execute	• initialize instrumentation ◦ eg. light sensor • set DUT to desired state
Exercise	**Exercise**
• execute test cases • save test data to database • check results for each test case	• execute commands in DUT • use instrumentation • collect test data
Verify	**Verify**
• verify result for test suite • save test suite result to database	• use additional software to analyse test data • verify result for test case
Cleanup	**Cleanup**
• set instrumentation to desired state • close additional software if needed • close connections if needed	• set instrumentation to desired state • close additional software if needed • close connections if needed

Figure 19. Role of test suite and test case

From Figure 19 we can see that the test suite is just a collection of individual test cases. The test suite is created normally based on some logic behavior, feature, or possibly production phase where closely related tests are collected to larger sets. The test environment can execute any number of test suites depending on how they were organized. Each test suite consists of one or more test cases. Test case focuses on carrying out one well-defined test procedure and verifying its result. The production test automation system designed in this thesis will use both test suites and test cases to carry out testing in practice.

4.4 Automation level

When discussing functional or dynamic testing, which is a large part of production testing, literature often cites that a high level of automation can be achieved. Nevertheless, literature also mentions that full automation is not often feasible or it is not worth pursuing in order to spare the automation team from additional work (Alsmadi, 2012). There is a profound difference in developing test automation for software testing or versus production testing purposes.

For production testing purposes high automation level must be achieved to carry out functional tests in a mass production environment. The difference between a moderate and high level of automation is not insignificant. To clarify the difference between different automation levels following example is given based on common requirements for such test case in the production testing domain.

Example - Take picture with mobile device

- verify that flash light is detected
- verify that image file is available

The example given above is pretty easy to automate to a moderate level of automation. Test automation runs the test and then asks a yes-no style question from the user ”was the flashlight detected”. Everything else except the answers to the questions can be automated, thus achieving a moderate level of automation.

For production testing purposes, the user input is not possible. It would in effect, defeat the purpose of automating the aforementioned test in a mass production environment. Instead, the detection of flashlight could be implemented by adding a light sensor to test environment. Selected additional instruments must be controlled during testing, so timing and synchronization between instruments and test case must be configured, and the data from instruments must be read and analysed. Also, limits for accepted light color, brightness, and duration need to be configured. The effort going from a moderate automation level to a high should not be underestimated. It is often the most time-consuming part of functional test and test automation development (Fewster & Graham, 1999).

Based on the author’s experiences in production, tests that cannot be executed without human interaction are mostly regarded as unusable. Running the aforementioned test without human involvement is possible, but the full automation level is not yet achieved. The device must be placed into the test adapter and removed after testing. For consumer products, the physical manipulation of the device to test adapter is rarely automated. Instead, it is done by test personnel (operators). So even if the testing process itself does not require human involvement, the ”pre” and ”post” phase probably will. For some products (especially those which are manufactured in very high quantity), the high level of automation is not enough. This means that also the physical manipulation of the device under test must be automated in order to achieve full automation level.

4.5 Cost of production testing

This chapter describes how the cost of production testing is formed at Bittium. The research problem is related to this context.

Production testing costs include hardware and software costs. Hardware costs are somewhat fixed: acquisition and maintenance costs. Hardware costs are not in the focus in this thesis. The main software costs come from developing the PTSW software, test cases to be executed during production, and the test automation software that controls the test environment to perform the test cases. This thesis focuses on the development of test cases and test automation framework but not on the development of PTSW.

Production testing software (PTSW) is the software that is running inside the device that is tested. The purpose is to allow the test environment to control the device via external commands and monitor and collect data from the device during testing. PTSW is tightly bound to the product hardware, and for this reason, it is not directly portable for the new hardware platform. Porting PTSW for new hardware usually requires that the hardware abstraction layer (HAL) be rewritten to facilitate new hardware. If the command interface is directly coupled to hardware without a proper abstraction layer between command interface and hardware, it might require that the whole PTSW needs to be rewritten. This is especially true if there are no common guidelines for the command interface.

4.6 Software development costs

This chapter describes how the software development costs of production are formed at Bittium. The research problem is related to this context.

If there is no common interface for the project&hardware, then each project usually creates its own interface that is specifically tailored for the hardware they use. This makes porting the PTSW to new hardware extremely difficult and leads to a situation where PTSW is developed 'per' product. This single system development approach leads to the situation that is described in the SPLE literature section (Figure 11) where accumulated development cost rises by a constant amount with each new product. If PTSW code cannot be reused in a meaningful way for new product, the development will cost roughly the same regardless of whether previous products have already developed from the same hardware platform or not.

Suppose the command interface defined in PTSW changes from product to product, so must the test automation implementation change to support the command interface. Test cases must be updated to match the interface defined in PTSW. Usually, updating just the commands used to control DUT is not enough because the functionality of the commands is also different, so the test logic as a whole needs to be reconstructed. This means that both test automation and test asset are tied to certain PTSW interface, and it is not portable from product to product.

Lack of a common interface between PTSW and test automation leads to a situation where the development of PTSW and test automation needs to be done per product. Each new product has to cover the costs of development as there is very little that can be ported between different products. This is the problem of single system development which Pohl describes in his book: each new product has to cover the fixed cost of PTSW and test automation development (2005). Figure 20 demonstrates the situation when PTSW and test automation are developed per product using a single-system approach.

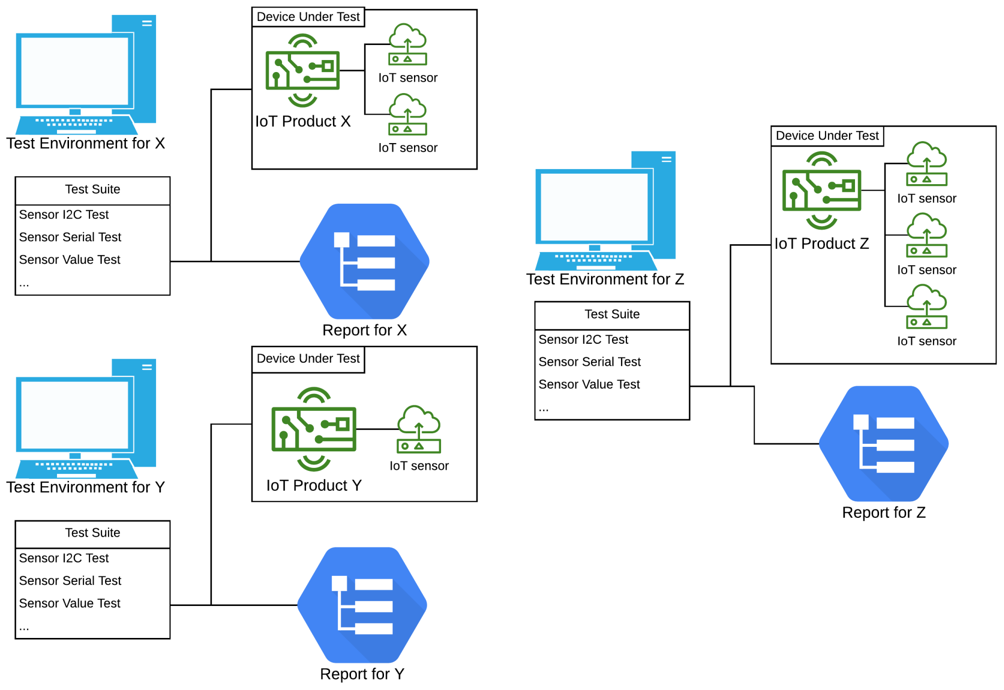


Figure 20. Test automation developed per product

As can be seen from the Figure 20 test automation needs to be duplicated for each product as each of the products uses its own PTSW and interface. The production testing generalization effort at Bittium tries to find a solution for this trend. From the product perspective, Figure 20 might seem a natural way of working; each product creates and uses its own test automation system. However, the reality of product development is that when the product enters mass production, the development team, including the test automation team, is transferred to new tasks, and the test automation is moved to the maintenance team. The maintenance team rarely works only on one product at a time. They may have several different products simultaneously in production. Looking figure 20 from the maintenance perspective seems terrifying as every new product will increase the work of the maintenance team. In the worst case, the different test automation systems have nothing in common, which means that each test automation system requires unique product-specific knowledge. The ever-growing amount of test automation systems will eventually become unmaintainable.

4.7 Effort to increase software reusability

There are many methods in literature to increase the reusability of the software. This thesis focuses on SPLE that is initially recognized as a possible solution to increase the reusability of software resources. One of the identified problems from the initial analysis was the lack of a common interface between test automation and PTSW. A common interface, from now on called PTSW API, allows to communicate with different products using the same command structure. This makes it possible to create a system where a single test environment is capable of testing multiple different products, as demonstrated in Figure 21.

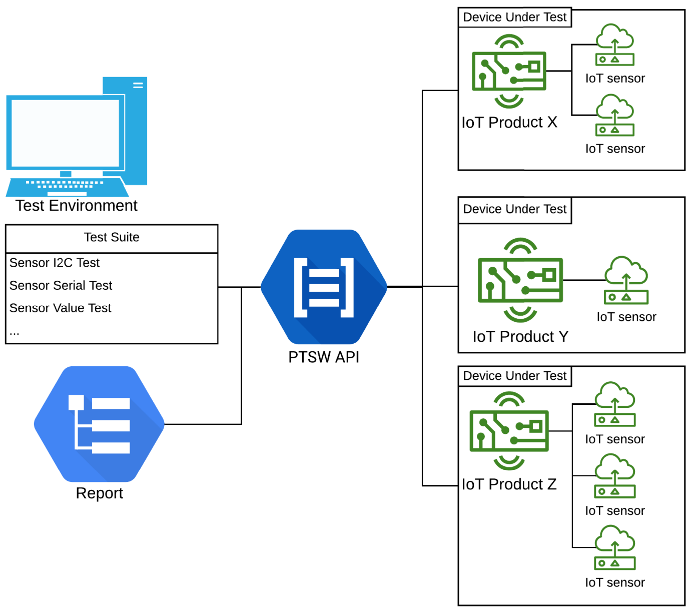


Figure 21. Test automation using common PTSW API

Figure 21 highlights the importance of common language (the API) between test automation and device under test (DUT). PTSW API does not automatically guarantee high reusability between DUT and test automation but makes it achievable. Development of PTSW API was already started before the work in this thesis. Development of PTSW API is not in the scope of this thesis. However, it is required that the proof of concept (PoC) developed in this thesis must use the current version of the PTSW API.

PTSW API in Figure 21 is also a simplification of the practical reality as the hardware-specific configuration, such as memory addresses and I^{2}C addresses, need to come from somewhere. This means that in order to execute the same test for multiple different hardware platforms, the test must be adapted dynamically for the used hardware. This can be achieved by separating hardware specifics from the test logic. Adapting the test case dynamically for specific hardware is in the focus of the creation of PoC in this thesis. The goal of PoC is to execute the same test case for multiple different hardware platforms by adapting the test case dynamically, using hardware configuration as parameter for the hardware being tested. In other words, the PoC created as a part of this thesis is an attempt to implement the system shown in the Figure 21 in practice.

5 Design

This section presents the design for the prototype test automation system developed in this thesis. First, there is an overall description of the problem to which the solution is searched for. Possible economical effects are also mentioned even though they are not in the scope of this thesis, but they provide insight into problem relevance. After the problem description, the design is presented in a top-down manner. Starting from the test automation system architecture, then presenting methods on how the choose what to test and how to test it. Design continues to construct test suites, and lastly, the design of individual test cases is presented. Implementation is described when there is a reason for it. For example, if unexpected difficulties were encountered, new or interesting solutions were discovered, or implementation highlights where the design succeeded well or poorly.

5.1 The Problem

The problem described in this section is based on the author's and colleagues experiences in the production testing domain in recent years. A rapidly increasing number of new products entering production creates technical challenges for production testing to keep up. This section tries to clarify the background for this trend.

The development of a new product starts with designing the hardware. When schematics for the new hardware are ready, the prototype version is usually built using development kits and other peripherals. The prototype is built to verify the hardware design and to function as an early version of the target hardware for software development. Low-level software features are implemented first. Parts that directly interact with the hardware, bootloader, drivers, kernel (if used), and so on. After essential parts of low-level software features are implemented, the actual product software development begins. Product software is synonymous with end-user software. PTSW and PT automation development usually start relatively late compared to end-user software, and commonly PTSW is based on the end-user software. End-user software is forked to create the PTSW, to expose access directly to hardware which is prevented in the end-user SW. Tests are designed and implemented when new features are added to PTSW, little by little. When there is functioning end-user software, PTSW, and test automation, production can begin. Following Figure 22 show the development phases needed before entering production.

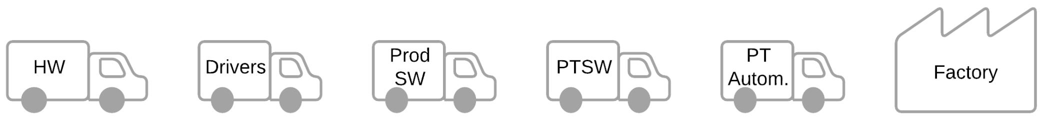


Figure 22. Development phases before production. PTSW based on end-user SW.

In practice, the development phases overlap with each other, but there is apparent linearity similar to what is presented in Figure 22. PTSW development can start when there is enough functionality in the end-user software, and automation development can start when there is an initial version of PTSW, which allows communication with the device. Figure 22 highlights that before the product can enter production, all of the development phases need to be completed. Development phases cannot start simultaneously because each requires some output from the previous phase. It is hard to shorten the development time needed before production because of this.

The logical first step is to separate the production testing from the end-user software as they have very little in common. After the hardware is designed, the product team can start implementing drivers and other low-level software features to allow the end-user software development to begin. Simultaneously the implementation of hardware abstraction layer for production test software (PT-HAL) can begin.

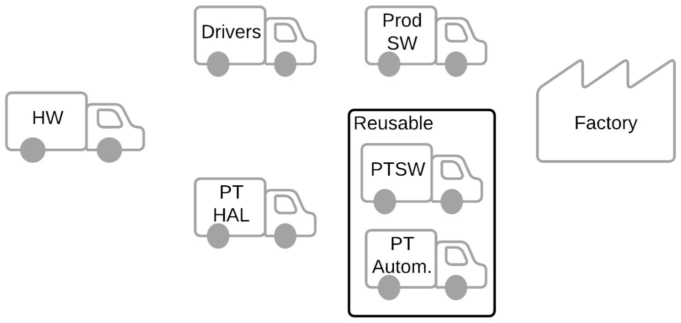


Figure 23. Development phases before production. PTSW separate from end-user SW.

Figure 23 shows that after hardware design is ready, the production testing and product development can begin simultaneously. This is possible if they are separated from each other. In addition, there is no longer a need to follow the API's used for end-user software, which allows using API designed specifically for production testing purposes. Such an API is demonstrated in figure 21 on page 36.

Based on the common API the PTSW and automation can be reused from previous products given that the required PT-HAL layer is implemented. This would mean that the required production testing software and automation are ready for production right after the PT-HAL is implemented. It is understood that in practice, there would be a need to adapt PTSW and/or automation to support product-specific features. The adaptation effort compared to developing both from scratch would be significantly lower. The needed adaptation would likely happen well before end-user software is ready for production.

If focusing strictly on production testing efforts during the manufacturing, it can be argued what would be the benefit to have PTSW and test automation ready before the actual end-user software. There are several reasons, and they are not apparent at first.

Often the end-user software is targeted to be ready when the product enters the mass production. However, mass production is rarely the first time the product is manufactured. There are usually one to three RnD and development builds (manufacturing batches) before actual mass production is entered. The purpose of these early builds is to produce a viable amount of prototype devices for the product verification process, development, and verification purposes. Production testing of these prototype devices could be considered vital, as the product verification process depends on them. If faulty units are passed to product verification and development purposes, they can skew the development and verification process significantly. It benefits products development if PTSW and test automation are already used in these development builds.

In addition to testing the prototype devices in development builds, it also enables to gather real test and measurement data from the device during manufacturing. This data forms the base on which the test limits and requirements for mass production are based on. If test and measurement data cannot be gathered during development builds, then it must be gathered during PTSW and test automation development. The development test environment usually differs from the one used in manufacturing, and also the number of devices available for testing is lower. It is common that if the development build produces 100 devices, the production testing development team will get realistically 1-10 devices of that batch for development purposes. This means that test and measurement data on which the mass production requirements are based on would be gathered using less than 10% of devices manufactured during the development phase.

In practise, it is also discovered that PTSW and test automation can be valuable in hardware development and other types of testing which the product has to go through, for example to get the European Conformity (CE) certification. Examples of other tests that the product usually needs to go through are environmental tests (heat, cold, chemicals), physical tests (drop tests, mechanical forces) and EMC testing (immunity to RF emission and emission levels) (European Comission website, 2021). Both the hardware development and the other type of product testing mentioned above need a way to confirm that the device is still operational. This can be achieved using end-user software but based on experience, that is much harder. PTSW is suited for this purpose, and running the production tests is easy because automation is made for it.

The reasons mentioned above suggest that finishing PTSW and test automation development as early as possible would benefit product development. Using the PTSW and test automation, the products verification process is easier to carry out, devices used in end-user software development would be more thoroughly tested, and more of test and measurement data could be gathered for mass production beforehand.

Another reason for using generic PTSW and test automation development instead of product specific approach is the economical factor. When PTSW and test automation are separated from the end-user software development, common PTSW API can be used between device and test automation communication. This enables reusability of test assets from product to product, given that the required PT-HAL is implemented for each new hardware platform. Figure 24 is simplification of the core concept.

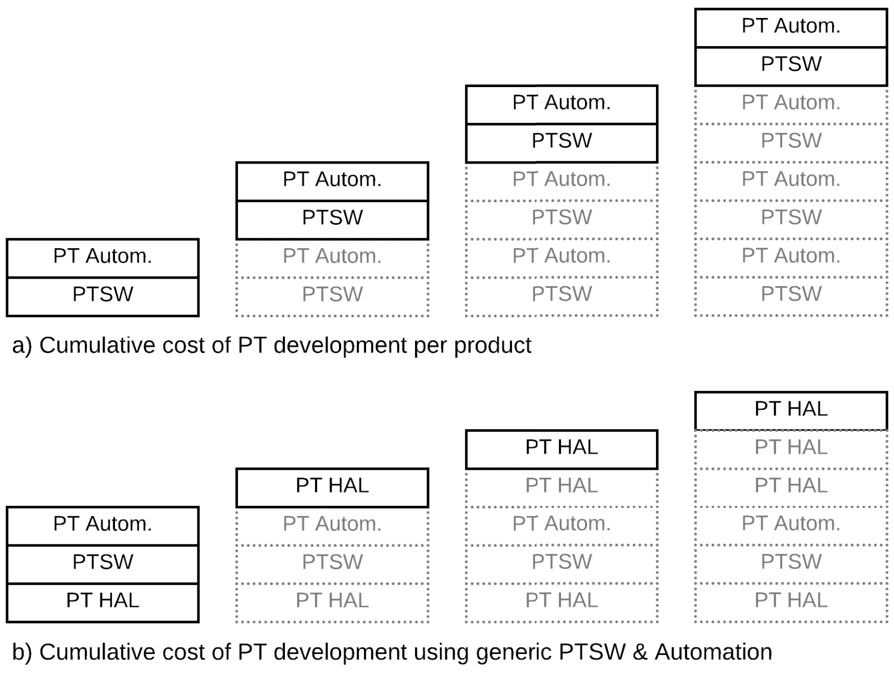

PT Autom.
PTSW
PT HAL
a) Cumulative cost of PT development per product
b) Cumulative cost of PT development using generic PTSW & Automation

Figure 24. Cumulative cost of PT development using different development strategies.

The effort of each step is presented in a similar size. It is safe to say that effort for PT-

HAL implementation is less than implementing full PTSW, but it is hard to estimate by how much. Also, initial effort of generic PTSW and automation is much higher compared to product-specific. Regardless, the point of the cumulative figure is to show growth over time. Option a) in Figure 24 grows faster than option b). Figure 24 can be compared to Figure 11 shown in SPLE Literature section (p.20). Both figures show that the generic approach will have a higher initial cost, but the cost will be lower over time compared to the product-specific approach.

5.2 Test Automation Architecture

Test automation architecture is based on a generic software testing framework presented by Alsmadi (2012). As was mentioned in the literature section, the purpose of automation is not to fully replace humans from the process but to decrease human involvement to a practical minimal. In the scope of the production test automation framework, we can focus automation effort on processes where it makes the most sense. For example, the process of building formal model can be done manually. Figure 25 shows the areas where the automation efforts in this thesis are focused on.

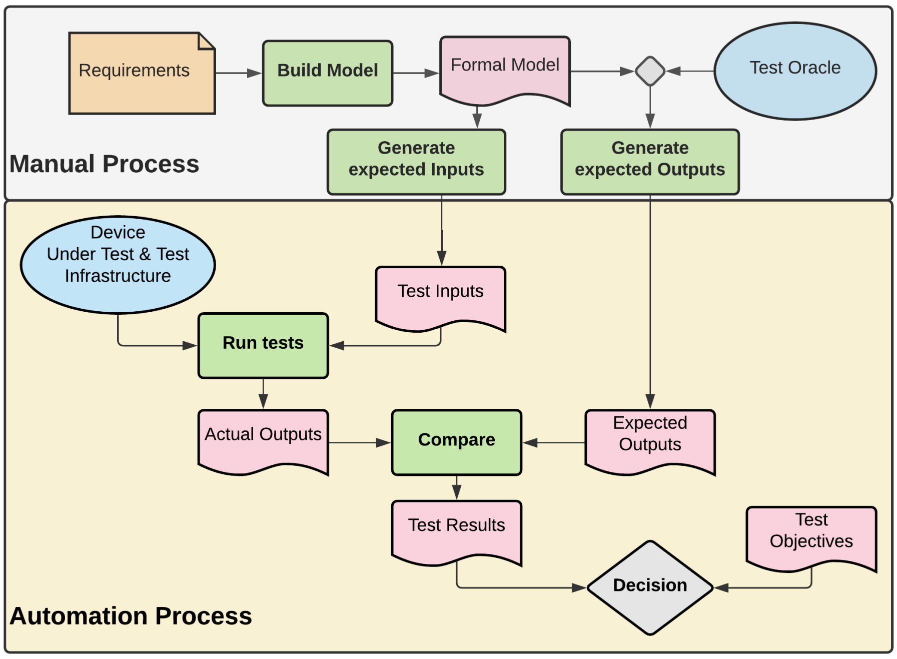


Figure 25. Generic software testing framework by Alsmadi (2012), divided for manual process and automation process

As Figure 25 shows, creation of the formal model, generation of test inputs and expected output is not part of automation effort, yet they are still included as part of a design for test automation framework. Model, expected inputs, and outputs are needed for automation. Test commands require input, test outputs are compared to expected outputs, or test limits that are defined based on the model. It can be summarised that the formal model in Figure 25 forms the test requirements. Test oracle, in this case would be a person or persons defining the limits for measurements, deciding the scope of testing, and in some cases verifying the test result if it cannot be automated. Design guidelines and the process of defining test requirements are described in the next chapter.

The target for actual prototype implementation covers the system capable of running the production tests, including connection to DUT and to test instruments, and comparing the results against predefined outputs or limits to generate test result report. The system must be able to determine if results did meet the test objectives in another word decide if the tests passed or failed.

Based on the description above the required functionality can be summarised in more detail:

- Test runner. Ability to define test suites and to execute individual test cases
- Allows creation of sub processes to handle communication to DUT and instruments
- Allows use of external software to analyse test results
- Ability to define pass/fail criteria for test cases and test suites
- Test result generator for executed test run

Requirements for test automation framework in this thesis were not very restrictive, required functionality can be found in a plethora of existing software testing frameworks. These frameworks usually offer much more than what is needed for a prototype system, such as management and development tools of test cases and test suites. They offer pre-configured communication protocols such as Visual Instrument Software Architecture (VISA), General Purpose Interface Bus (GPIB), Universal Serial Bus (USB) to be used to control instruments and test report output formatter to mention few.

Based on the requirements set for the test automation framework, the open source Robot framework was selected to provide the core functionality. Robot framework is Python based, it is available from Python Package Index repository (PyPI), and installation takes only a few minutes. Robot framework is very extensible, and it also allows to modify the existing core functionality (Robot Framework, 2021). For these reasons, Robot framework has gotten a lot of traction in recent years, and it has grown to be one of the major software testing frameworks used by industry.

The light-weight nature of the framework and the fact that everything is runnable from the command-line without a graphical interface make it well suited for automation development. Test server can be installed for example, on Raspberry Pi and operated via Secure Shell (SSH). These kinds of headless servers (computer without monitor, keyboard, mouse, or other peripherals) are an emerging trend in production testing.

5.3 Process of defining test requirements for production testing

Requirements form an essential part of production testing. As was previously mentioned in the Literature section by Myers et al. system testing is impossible without a set of written and measurable objectives for the product (Myers et al., 2012). For production testing (a subcategory of system testing) measurable objectives dictate the scope of testing and criteria for the test verification. Standards form additional test requirements for product. Standards commonly offer measurable criteria for different levels of standard or acceptance criteria in overall. Production tests must cover all of the test requirements, including those defined by standards, one way or another.

While the definition of test requirements is not part of automation in this thesis, it is still a fundamental part of test automation framework that cannot be ignored from the design.

For production testing, the main source for requirements is the hardware schematics and specifications. Production testing is limited to verifying the manufacturing process of the product (QC), not products behavior in its intended environment (QA). Furthermore, requirements must be measurable or otherwise unequivocally verifiable. The only reliable source for this information comes from hardware specification and schematics.

Before proceeding further, the hardware used in this thesis is described. Hardware specifications for selected components and the schematics of how they are connected are used later to define test requirements.

5.3.1 Design of example products

This section shortly describes the hardware that was used as a target for production testing. Target hardware itself or its design is not in the scope of this thesis. Description of the hardware is presented to help the reader to understand the context in which the production tests are performed.

Example products were created to mimic actual products without the burden of non-disclosure agreements. Microcontrollers and peripherals were selected based on actual products designed on Bittium, but there is no actual behavior planned for the product in addition to demonstrating basic functionality of the hardware.

Design for products were based on the general description of the embedded system given by Xiao in the Section 2.4 at the page 18. The example products have peripherals that provide input for the microcontroller and allow the microcontroller to send output.

For input devices, a press button, a real-time clock (RTC), and air quality sensor were selected. Button is connected via General-purpose input-output (GPIO), air quality (temperature, humidity, pressure, and gas) sensor with serial peripheral interface (SPI) and RTC clock using I^{2}C interface. Output devices have LED connected via GPIO, 7-segment display connected using I^{2}C interface, and organic LED (OLED) display connected to SPI interface. For the communication interface the Universal Asynchronous Receiver-Transmitter (UART) was chosen. Components are summarised in Table 3.

Table 3. Main components summarised by their interface and purpose

Interface	Component	Purpose
UART	Serial communication	Communication
GPIO	Button	Input
I^{2}C	RTC clock	Input
SPI	Air quality sensor	Input
GPIO	LED	Output
I^{2}C	Seven segment display	Output
SPI	OLED display	Output

Parallel to this study, another thesis was conducted focusing on the software used in the device during production testing. In his thesis *Reusable firmware for IoT device production testing* Sankila studies the implementation of the PTSW for new hardware platforms (Sankila, 2021). As our studies were aligned, some collaboration was done to advance our work further. This thesis uses the same example products that were constructed during Sankila's own study. Required adaptations to the production test software were also made by him. For example, one of these adaptations was the implementation of the SPI interface for the OLED display and air quality sensor.

First of the example products represents low-end microcontrollers with a limited amount of processing power, memory, and connections. First example product is based on the NXP^R LPCXpresso812-MAX Board. Second example product represents high-end microcontrollers with higher processing power, memory, and connections. Second example product is based on NXP^R MIMXRT1010-EVK board. The main features of the used example products are summarised in Table 4.

Table 4. Main features summarised for example products.

LPCXpresso812-MAX	MIMXRT1010-EVK
Arm Cortex-M0+ Core	Arm Cortex-M7 Core
30 MHz clock speed	500MHz clock speed
4 KB RAM	128 KB RAM
-	64 KB ROM
16 KB flash memory	128 Mb QSPI flash memory
I^{2}C, SPI, GPIO, UART	I^{2}C, SPI, GPIO, UART

Selected components can be wired to both example products, even though low-end platform imposes some restrictions for OLED display testing. After the selection process was done, components were wired to evaluation kits to function as example product.

5.3.2 Defining test requirements

When the product hardware schematics and component specifications are available, the test requirements can be defined. First, the functionality to be tested should be defined. Functionality to be tested should reflect the intended usage of the product. Most critical parts of the design should be recognized as this helps to prioritize production testing. After functionality to be tested is defined, schematics are studied to find which components are needed to provide the functionality and how these components can be tested.

For example products the intended usage was not defined. Even though the end-user software is never meant to be implemented for these products, the intended usage is important for defining test requirements. In order to specify test requirements, later on, the intended functionality listed in Table 5 was defined. The intended usage of example products is a simplified version of what the usage of actual products would be, but it covers much of the same functionality of the components that are used in real products.

Table 5. Functionality of the example product defined.

no	Functionality
1.	LED should turn ON when Button is pressed
2.	LED turns OFF when Button is not pressed
3.	Seven segment display shows the time retrieved from RTC clock
4.	OLED display shows the temperature, humidity, air pressure and carbon monoxide values

Table 5 summarises the intended functionality of the example product. From this functionality, we can define test requirements. To cover the intended behavior all the components of the product must be tested. This is usually the case as the hardware seldom contains unused components. In the case of example products, each component can be directly accessed from the MCU, which simplifies testing. This is not always the case as some components are rarely connected to the MCU directly. Instead they are connected to another component. For example, RF antenna is rarely connected to MCU, instead, it is connected to the modem. In this case, the antenna must be tested indirectly, for example, measuring the transmission (TX) signal strength from the modem. Measurement, in this case, would be done using external instrumentation, as the signal strength cannot be measured from the device itself. Before production, the limits for signal strength are set based on the earlier measurement results both with and without the antenna. In the case of example products, the indirect tests are not needed as all the peripherals are connected directly to MCU.

To cover the features of intended usage the production tests must verify the following hardware functionality.

Table 6. Functionality to be tested from example product.

no	Functionality to be tested
1.	Button On/Off state
2.	LED On/Off state
3.	RTC clock can be started
4.	Value of the RTC clock can be retrieved
5.	Temperature, humidity, pressure and carbon monoxide value can be retrieved from the air quality sensor
6.	Each segment of seven segment display can be turned on and off
7.	Each pixel of OLED display can be turned on and off

Table 6 summarises the functionality to be covered with production tests, and it forms a fundamental part of test requirements. Production tests aim to cover 100% of the hardware schematics, but that cannot always be achieved. For example, device might not have required test points on its circuit board to connect instruments for measurement, and signal cannot be measured inside to component to be tested. In these cases, indirect testing is used. Indirect testing simply means: if component A is working and it is dependant on component B, therefore also component B must be working. There are limitations on how long testing can take, what testing equipment is available, and so on. Production tests should be prioritized based on the most critical behavior of the product, and this information should also be included in test requirements. Overall test requirements try to answer the question of what should be tested.

5.3.3 Designing test cases based on hardware schematics

When test requirements are known, the next thing is to define how the testing should be conducted. The answer to this question should be based on the hardware schematics: what is the simplest way to verify that component is operational?

Testing almost always starts with verifying that the device can be turned on and there are no short circuits. The ”power up” test is pretty self-explanatory. The power supply unit is used to feed desired voltage, and current is monitored to detect short circuits. If power consumption stays within predefined limits, the testing continues to test the MCU. Short circuit and the power-up tests are not related to production testing effort in the scope of this thesis, so they are not presented in the test requirements.

MCU is the logical first component to test as it is the component needed to communicate with the device. Internal functionality of the MCU is already tested by the MCU manufacturer, so the production test usually just tests if communication can be established with the device or not. If the communication with the device succeeds, it is assumed that the MCU is operational. Only after the communication with the device is established can the testing proceed to other components.

The method for how the test cases can be mapped from the schematics can be demonstrated by looking, for example how the I^{2}C sensor could be tested. This method is commonly used to design production tests, albeit the author of this method is not known. This method can be considered as common knowledge in the production testing domain. As already stated, the schematics guide the test design, so we need to look at how the sensor is connected to the MCU. Example schematics is presented in Figure 26.

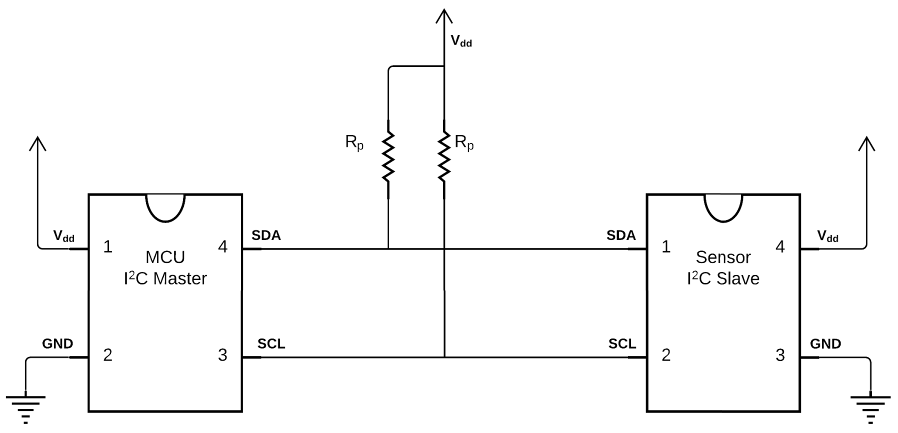


Figure 26. Example schematics for I^{2}C sensor.

Figure 26 is simplified schematics of MCU, pull-up resistors connected to the bus, and I^{2}C sensor module to make it easier to demonstrate the process how testing of the sensor should proceed. In order to communicate with the sensor, connection to the MCU has to be established first. MCU is the only way we can access the sensor itself. We have to also consider the pathway (bus) from MCU to the sensor, the actual wiring of these components. If the bus is not operational, the communication between MCU and sensor will not succeed even though the sensor itself might be functioning correctly. Figure 27 shows how the testing expands from the MCU outwards.

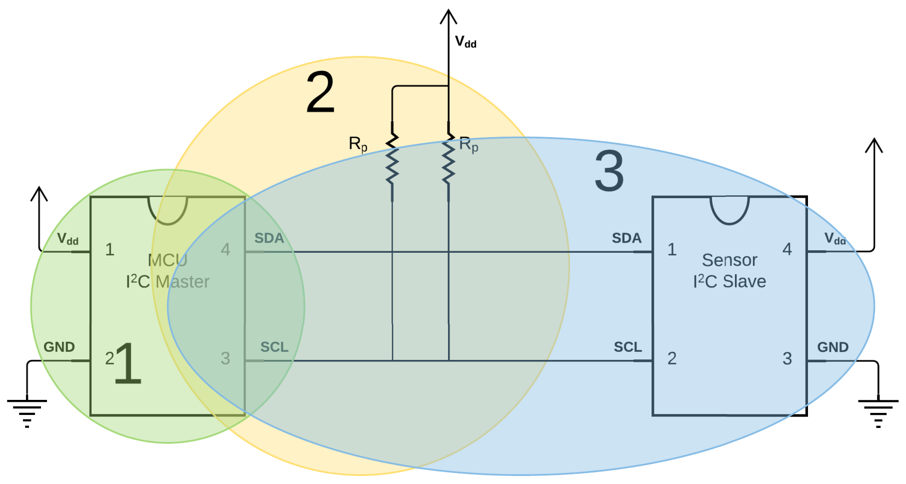


Figure 27. Test phases expanding from MCU towards sensor.

Figure 27 demonstrates how testing proceeds based on example hardware schematics. First testing phase is to ensure that MCU is operational and the communication with the device is working. This also covers the short circuit test mentioned before. The second phase is to verify if the connection (bus) between MCU and sensor module is functioning properly. The bus is not always testable, but the pull-up resistors connected to it make it testable in this example. If there were no pull-up resistors, the only way to verify if the bus is operational is to try to communicate with the sensor. If the communication succeeds, then also the bus must be operational. But if the communication fails, either the bus is not operational, or the sensor is not operational. For this reason, the bus should be tested separately from the sensor if possible as it would pinpoint the problem more accurately.

It is not obvious how the bus can be tested using the pull-up resistors, so short overview of the test is given here. MCU pins 4 and 3 are configured as serial data (SDA) and serial clock (SCL) when communicating with the sensor. But these pins can be configured as general-purpose input-output (GPIO) pins also. If both pins are configured as GPIO inputs and the value of the pins is read, they both should be high as the pull-up resistors should pull the GPIO state up. If either of the pins is low it is likely that the wire is cut or the pull-up resistor is not connected. In this case, testing the sensor does not make sense as the problem is identified before the sensor is reached.

The third phase is to communicate with the sensor. Commonly the module id register is read first. Id register has a static value that is easy to verify against the sensor specification. If reading the id register succeeds, then the actual sensor value is requested. Verification of the sensor value happens against predefined limits.

The above-mentioned method was applied to all components on the example product to reveal how each component can be tested. Following Table 7 summarises the planned tests based on the schematics and component specification to cover needed functionality. Production tests must be designed to cover all of these tests during test automation run.

Table 7. Planned tests based on test requirements.

no	Component	Test	Interface
1.	LED	Turn on test	GPIO
2.	LED	Turn off test	GPIO
3.	Button	Press test	GPIO
4.	Button	Release test	GPIO
5.	RTC clock	Pull-up resistors test	GPIO
6.	RTC clock	Get id register value	I^2C
7.	RTC clock	Set time	I^2C
8.	RTC clock	Get time	I^2C
9.	7-segment display	Pull-up resistors test	GPIO
10.	7-segment display	Get id register value	I^2C
11.	7-segment display	Turn all segments on	I^2C
12.	7-segment display	Turn all segments off	I^2C
13.	Air quality sensor	Get id register value	SPI
14.	Air quality sensor	Get temperature value	SPI
15.	Air quality sensor	Get humidity value	SPI
16.	Air quality sensor	Get pressure value	SPI
17.	Air quality sensor	Get carbon monoxide value	SPI
18.	OLED display	Get id register value	SPI
19.	OLED display	Turn all pixels on	SPI
20.	OLED display	Turn all pixels off	SPI

Each component of the example product was studied individually, and planned test cases were added to Table 7. When planning test cases, the verification of the test result was considered but left still open, it is often more productive to first concentrate on what should be tested and how and then focus on the verification method. If tests are planned based on what verification methods are available, it has a tendency to skew the test case design. Next, the verification method is considered for each test case, and initial acceptance criteria are set. Both the verification method and criteria are subject to change during product development. Previously collected data is valuable in order to set limits for measurements.

Table 8. Verification for planned tests.

no	Test	Method	Criteria
1.	Turn on test	Visual	LED on
2.	Turn off test	Visual	LED off
3.	Press test	Value	Matches schematics
4.	Release test	Value	Matches schematics
5.	Pull-up resistors test	Value	Matches schematics
6.	Get id register value	Value	Matches specification
7.	Set time	Exit code	Verify that operation succeeded
8.	Get time	Value	Set time + elapsed time in seconds
9.	Pull-up resistors test	Value	Matches schematics
10.	Get id register value	Value	Matches specification
11.	Turn all segments on	Visual	All segments on
12.	Turn all segments off	Visual	All segments off
13.	Get id register value	Value	Matches specification
14.	Get temperature value	Value	18-30 celsius
15.	Get humidity value	Value	50-80 %
16.	Get pressure value	Value	1005-1020 hPa
17.	Get carbon monoxide value	Value	0-5 ppmv
18.	Get id register value	Value	Matches specification
19.	Turn all pixels on	Visual	All pixels white
20.	Turn all pixels off	Visual	All pixels black

Table 8 presents what is the planned verification method for each test and on what the acceptance criteria is based on. Acceptance criteria are based primarily on hardware schematics and component specification. Actual values for specification and schematics are not presented here as they offer little insight. Visual inspection is used to verify the tests for LED (1-2), 7-segment display (11-12) and OLED display (19-20). Limits for air quality sensor values are set based on the normal readings measured indoors. Information is taken from Finnish Meteorological Institute website (2021).

Acceptance criteria can be loose at the beginning of the development, and it can be tightened over time. Even the loose acceptance criteria force the test requirement designer to think about the test result verification. If the verification is ignored altogether, it poses difficulties during test implementation. As already mentioned Myers et al. has stated that system testing is impossible without a set of written and measurable objectives for the product (Myers et al., 2012).

5.4 Test order

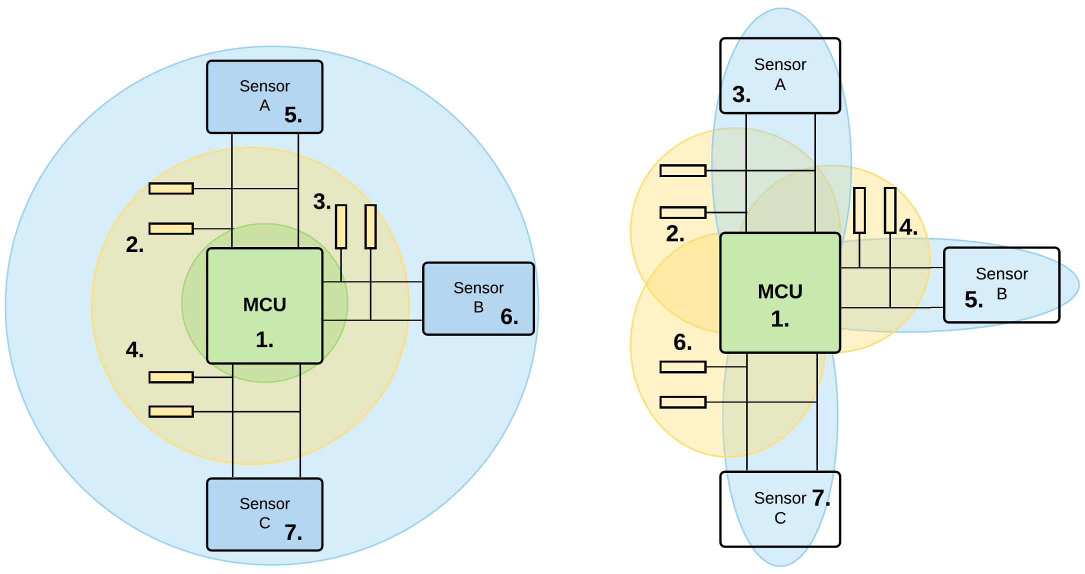
Test order can have a significant impact on test execution time which in return affects production testing efficiency. Execution times are not nearly as important during development as they are when entering mass production. For high-volume manufacturing batches, fraction of second are significant. There are several different ways how the test order can impact the test execution time.

If test case setup or teardown requires time, ordering test cases in a way that requires a minimum amount of setups and teardowns to be executed optimize time taken by them.

This kind of ordering is important to optimize the time it takes for a fully functional device to pass all test phases. In addition to this, there is attempt to catch faulty devices as early as possible from the test process. So tests can also be ordered by failure likelihood based on design, statistics, and existing manufacturing data. Optimizing the test order should be considered from these two points, what is the quickest way for the functional device to pass testing and what is the quickest way to capture a faulty device. Optimization based on functional devices is easier. It can be tested in practice using so-called golden units, which are known to be fully functional. Optimization based on faulty units is more complex as there is always the variable of which component is faulty. Statistics help with this, once available.

In production testing, there are stronger dependencies between test cases compared to traditional software testing. In fact, software testing actively tries to minimize dependencies between test cases, for example by using random test ordering to reveal hidden dependencies. In production testing, this is not feasible as there is always dependency on the target hardware. In most cases the tests should be executed in the same order as they were designed, demonstrated in section 5.3.3 in figure 27 (p.46). Tests can be freely ordered given that other testable components and pathways on route to target component are tested first. For example, in order to run RTC clocks Get time test, the pull-up resistor test (7) and id register read (8) should be run first. In addition, to verify the returned time reliable the Set time test (9) should be executed beforehand to compare time against a known value. Because of the dependencies between test cases, they cannot be ordered individually, rather they should be ordered as sets.

Two commonly known search strategies can be applied in the production testing context to order testing. They are known as breadth-first (BFS), and depth-first (DFS) search strategies, and they are shown in Figure 28.



(a) Breadth-first **(b)** Depth-first

Figure 28. Testing order presented by Breadth-first and Depth-first

Figure 28 demonstrates testing of three I^{2}C sensors using breadth- and depth-first strategies. In breadth-first strategy, testing expands from MCU as waves, pull-up resistors for all three sensors are tested before the first sensor is tested. In a depth-first strategy each sensor is fully tested, including pull-up resistors before testing continues to the next sensor.

Which strategy fits better is heavily impacted by the hardware design. In some cases the depth-first strategy can reduce the amount of needed test setup and teardown phases. If there are multiple components connected to the same bus, the breadth-first strategy might be a better approach. The test-order optimization is usually relevant when product is nearing the mass production phase. Test ordering in the scope of this thesis is not important. The depth-first strategy is used in this thesis to order tests as it resembles the process of how the test cases were designed from the hardware schematics, presented in Figure 27 (p. 46).

5.5 Test suite design

The role of a test suite is to order testing in practice. Test suite is also a way to collate similar tests, for example, those that have dependencies between them, as was discussed in the previous section. Test suites can include other test suites and test cases. For this reason, one test suite can be used to order testing at the high level and other test suites to order test cases that have dependencies between them.

Test cases presented in Table 7 were divided to test suites based on the interface they use and dependencies between test cases.

Table 9. Planned tests divided to test suites.

Suite	Component	Test	Interface
1.	LED	Turn on test	GPIO
1.	LED	Turn off test	GPIO
1.	Button	Press test	GPIO
1.	Button	Release test	GPIO
1.	RTC clock	Pull-up resistors test	GPIO
1.	7-segment display	Pull-up resistors test	GPIO
2.	RTC clock	Get id register value	I^2C
2.	RTC clock	Set time	I^2C
2.	RTC clock	Get time	I^2C
3.	Air quality sensor	Get id register value	SPI
3.	Air quality sensor	Get temperature value	SPI
3.	Air quality sensor	Get humidity value	SPI
3.	Air quality sensor	Get pressure value	SPI
3.	Air quality sensor	Get carbon monoxide value	SPI
4.	7-segment display	Get id register value	I^2C
4.	OLED display	Get id register value	SPI
5.	7-segment display	Turn all segments on	I^2C
5.	OLED display	Turn all pixels on	SPI
5.	7-segment display	Turn all segments off	I^2C
5.	OLED display	Turn all pixels off	SPI

Table 9 divides test cases into six separate test suites, and the number of test suites indicates in which order those test suites are planned to be executed.

Test suite 1 is a collection of all GPIO tests, including the RTC clock and 7-segment display pull-up resistor tests. LED, button, and both display tests require input from human as their automation in the scope of this thesis is not feasible. First LED is turned on, and the user is asked for verification if the LED is turned on. After the LED is turned off, and user is prompted again to verify if the LED is turned off. Then user is instructed to press the button. The state change of the button press and release can be detected directly from the device, so user verification is not needed. After the button is pressed, testing can continue automatically. Pull-up resistor for both I^{2}C modules is checked to verify the bus functionality.

Test suite 2 contains RTC clock tests ordered based on their dependencies. First, the id register value is checked, after which the time is set, and clock started. Then the time is read back and verified against the time set plus elapsed time since the clock was started. Suite 2 does not contain other components, so that test suite can be ordered later in sequence. Only restriction for reordering is that suite 1 must always be before suite 2.

Test suite 3 is similar to suite 2. First sensor id register is read, and then values for each integrated sensor are read. Values are checked against normal atmospheric values in indoors.

Suite 4 contains module id read tests for both display units. It is meant to verify that both displays are connected correctly. Suite can be reordered as long as suite 1 is run before it.

Suite 5 contains display tests that require verification from the user. All the segments from the 7-segment display and pixels from the OLED display can be turned on simultaneously so that they can be checked at the same time. As last test both displays are turned off, and user is prompted to verify that they have been switched off.

5.6 Test case design

Individual test cases follow the Four-Phase Test method. Each test has setup, exercise, verify, teardown phase (Meszaros, 2007). Used Robot framework allows defining setup and teardown functions for individual test cases and test suites which allows controlling test execution on both levels. The purpose of setup and teardown is to guarantee that they are executed when a test case is run, setup before, and teardown after. Especially the teardown is important in production testing. If instrumentation is used during testing and it is initialized during setup phase, it needs to be deinitialized even if test execution fails. In Robot framework there is special tags for setup and teardowns, and framework guarantees that these functions are called before and after the test is executed.

For each test case, the setup and teardown functions were added even if no functionality were added to these functions. This was done to ensure that execution for each test follows the Four-Phase method.

TODO: Mention about the common interface!

PTSW API was developed to create a common interface between the testable device and test automation for production testing purposes. This interface is meant to be implemented for new IoT products, and production tests should exclusively be implemented using it. The device is running a simple command-line interpreter which reads the commands sent via UART, executes them, and returns exit code along with the requested data. Tests designed and implemented in this thesis have to interact with the device using the current version of PTSW API.

PTSW API defines three basic operations: configuration (C), read (R), and write (W). These operations are tied to a used hardware interface, for example, SPI (S) and I^{2}C (I). Hardware interface and operation base the syntax for commands.

Table 10. Example of the PTSW API command structure

Command base	**Description**	**Interface**
`IC:<bus>:<enable>(:speed)`	Configure bus on/off	I^{2}C
`IW:<bus>:<address>:<data>`	Write data to register	I^{2}C
`IR:<bus>:<address>(:bytes)`	Read (n bytes) from register	I^{2}C
`SC:<bus>:<enable>(:speed)`	Configure bus on/off	SPI
`SW:<bus>:<data>`	Write data to register	SPI
`SR:<bus>:<bytes>`	Read (n bytes) from register	SPI

PTSW API contains additional hardware interfaces which are supported but not present in the Table 10. For example, there are interfaces for GPIO, LED, Memory, and Controller Area Network (CAN) bus. They have slightly different syntax but are based on the same configure, read and write concept. Only relatively low-level operations are supported in PTSW API so that implementation of those operations is easier for the device software. Also, there is a conscious effort to move the test logic from the device to test automation. Performing complex tasks using this command interface would be tedious to do manually, but it fits well in to the test automation environment.

5.7 Support for Multiple products

Support for multiple products is the main focus for test automation design and implementation in this thesis. SPLE literature was used to find techniques and methods for finding variation points and also known solutions to handle variation in the application domain. Test implementation for example products started after the method for finding variation points was defined. Tests were grouped to test suites described in Table 9. Test suites were executed to verify that test results matched requirements defined in table 8.

SPLE literature defines domain engineering and application engineering. Domain engineering focuses on general or common assets, while application engineering focuses on creating products based on common assets. Pohl (2005) define domain engineering and application engineering in following way:

> *Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realised.*
>
> *Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artefacts and exploiting the product line variability.* (Pohl, 2005, p.21)

The work in the thesis is combination of both. Thesis does not focus solely on the creation of common assets. Test automation and test cases also need to be implemented and executed in practice. This belongs to the application engineering context.

SPLE literature highlights the importance of eliciting common and variable requirements of the product line. However, the process of defining commonalities and variations described in the literature does not fit the scope of this thesis very well. Some of the processes from the literature are aimed at development team, rather than an individual. Pohl (2005) also mentions that variability can be added to almost anywhere, but adding it to places where it is not needed can be costly. It increases the implementation effort without adding any value, and unused variation most likely will be removed later on (Pohl, 2005).

In order to avoid introducing unnecessary variation to prototype design and to avoid labor intensive design process alternative way was used to find required variation points. The used method can be summarised by following pseudo algorithm.

```
LOOP until all tests PASS:                                          (1)
    Implement test cases for the first example product
    Execute tests using first example product

FOR i TO n:
    LOOP until all tests PASS:
        Execute tests using example product i
        Mark failures as variation points
        Implement needed variation to fix failures
```

The method presented in pseudo algorithm 1 has its benefits and weaknesses too. Variation is not introduced until it is needed. This addresses the issue of overusing variation in places where it is not needed (Pohl, 2005). Two example products were designed so the second loop could be executed, but only once. This has limitations on how many variation points are discovered by this method. However, this method can be used to discover more variation points later on as the PTSW has been already ported to over ten different hardware platforms. Alas, this work will be outside of the scope of this thesis. Regardless of the downsides, this method for finding the needed variation points was used as it enabled the implementation work to begin early on.

The used method in the context of DSR can be seen as ”design loop”. In the case of the first example product the test cases are implemented, and they are evaluated by running test automation and checking test results against test requirements. When all test cases pass (test requirements are met) for the first example product, test automation is executed for the second example product. Failures are marked as variation points, and required variation is implemented. It is again evaluated by running test automation and checking test results against test requirements. In this thesis, there are only two example products used, but the same technique can be used for additional hardware platforms in the future in order to detect additional variation points not discovered in this work.

5.7.1 Implementing tests for the first example product

Test implementation for the first example product was straight forward. Table 9 (p.50) shows lists of tests needed to be implemented to cover all hardware components required for planned functionality (covered in Table 6. p.44). Test results were verified against criteria defined in Table 8 (p.48). Expected outputs were defined directly in test suites and values read from device were compared against these.

The need for variation was not taken into account while implementing tests for the first example product. Hardware-specific values such as I^{2}C bus numbers and component addresses were directly inserted into test cases even if it was clear that these values would potentially change for the second example product. Still, there was a conscious attempt to avoid introducing variation before there was a valid reason for it, namely a failing test case. After each test was implemented, the test automation loop was executed to verify that tests passed based on its acceptance criteria (Table 8. p.48). When all test suites were implemented, all test requirements were met, and all tests passed, the first example product was accepted.

5.7.2 Executing tests on the second example product

When implementation for the second example product started, all the test cases were already done. Example products share the same functionality, so no new tests were needed. Test suites could be executed for the second example product to detect failures that indicate the need for variation.

Failures are summarised here with an analysis of the problem. The test run was executed without `exit on failure` option, which would stop execution immediately at the first failing test. Without this option, all test cases were executed regardless of other results. Test results are visible in Table 8 where test number matches test cases in tables 7 and 8.

Table 11. Summary of test results for the second example product

no.	Error	Interface
1-5	FAIL	GPIO
6-8	PASS	I^2C
9	FAIL	GPIO
10-12	PASS	I^2C
13-20	FAIL	SPI

Surprisingly the I^2C tests seemed to be working, but that was a lucky coincidence. I^2C components can be wired in multiple ways; there can be one or several components on each bus. By coincidence, both example products had only one component per bus, and the bus index was mapped equally in both products: bus 0 for RTC clock and bus 1 for 7-segment display. Tests for these components worked, but the test suites would not have passed if the test option `exit on failure` had been used. The pull-up resistor test failed for both components as the GPIO pins had been changed. This would have caused the first test suite to fail due bus verification, and the execution would have been stopped there.

All GPIO tests were analysed to be broken because the pin mapping (port or pin) had changed. GPIO port&pin were identified as the first variation point. Components on the SPI bus were analysed to have two problems, one that caused failures and one that would cause failures in the future. The evident problem was the bus indexing, even though it was not behind the failures. Similarly to I^2C the SPI components can be wired multiple ways to hardware, one or several components per bus. SPI busses were mapped similarly in both example products, bus 0 for air quality sensor and bus 1 for OLED display. But unlike the I^2C tests, SPI tests all failed. This was discovered to be because of the chip select (CS) signal, which is used to select the component. Chip select is GPIO signal, and it was wired to a different pin compared to the first example product.

Even though the GPIO port and pin mapping was identified as the only reason for failures, the I^2C and SPI bus mapping was added to the variation list also. The wiring could have been changed so that bus 0 would be bus 1 and vice versa, and this would have caused the I^2C and SPI tests to fail, but problem was evident without doing so.

5.8 Finding the needed variation

In order to fix the failed test cases, the variation for GPIO, I^2C and SPI needed to be designed and implemented. Variation was studied for each interface separately to find commonalities and differences. The simplest way to fix test failures would have been to create a configuration file, that would contain named variables for each failing test case, in the style of `LED_GPIO_PORT=A`, `LED_GPIO_PIN=3` and so on. This solution would be easiest to implement, but it is brittle and cumbersome. Test cases need to use these variables directly, and changing the test case becomes harder. If the next product has seven LEDs instead of one, new variables for each LED would have to be added and inserted into the test case. This approach was discarded as it is not generalizable.

Early candidates for configuration format were discovered as:

- using component as basis
- using bus interface as basis

These approaches are presented in Figure 29.

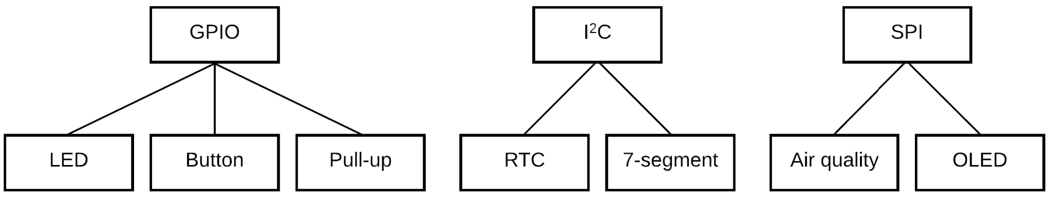


Figure 29. Hardware interfaces and components

If Figure 29 is read from top to bottom, the configuration is interface based, where root nodes represent hardware interfaces. If Figure 29 is read from bottom-up, then the root nodes are the hardware components. There is no difference information-wise in which approach is used. Mainly this is a concern for how the configuration data is constructed in practice. Using interfaces as the root nodes seemed a more natural way to organize configuration data, so interfaces were chosen as root nodes.

Next, all the test cases were reviewed to collect all attributes needed to perform the test case successfully. Attributes were categorized according the hardware interface they used. Source, where this data should be configured, was considered. Result of this analysis can be seen on figure 30.

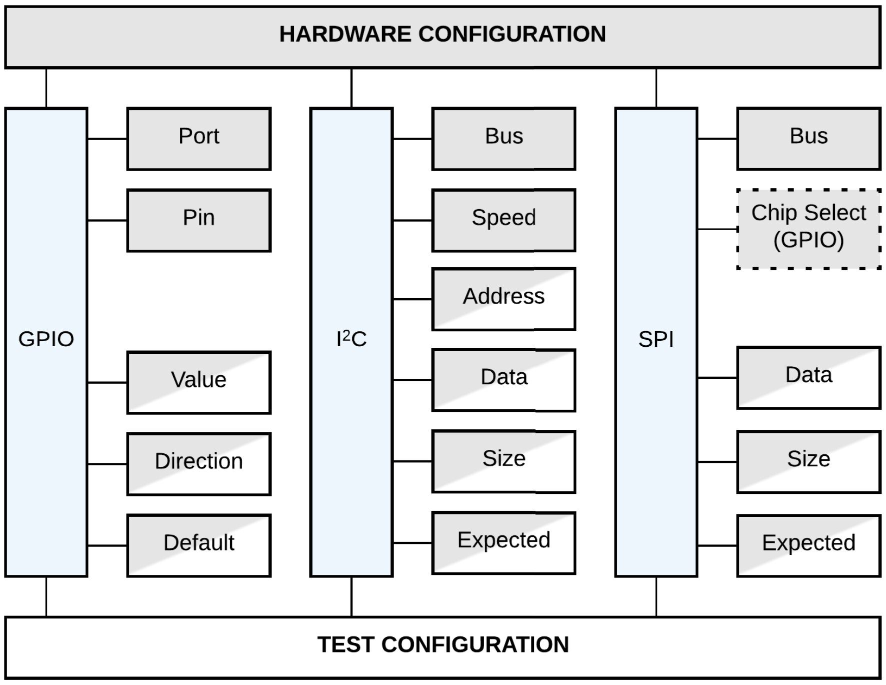


Figure 30. Required test attributes divided between hardware and test configuration.

Hardware interfaces are presented in Figure 30 as long vertical bars, and their configuration (required attributes) can come either from hardware configuration or test configuration. Attributes belonging to hardware configuration are marked with a solid color. These attributes have the tightest coupling to hardware. Other attributes required to execute tests can be configured via the test case. The *chip select* in the SPI interface is not strictly related to it, and it could be moved to GPIO interface. However, SPI bus cannot be successfully configured if *chip select* signal is not handled correctly. For this reason, it is included in SPI interface rather than GPIO.

Some attributes seemed to be related to test logic itself rather than hardware. GPIO attributes *direction* and *value* were recognized to be part of test logic. These attributes depend on whether the test case is changing the value of the GPIO (write) or inspecting its value (read). If GPIO value is changed its *direction* has to be set to output and *value* defines which value is written. If GPIO value is read, the *direction* has to be set input (and *value* parameter can be ignored). *Default* parameter for GPIO is not related to controlling the hardware, but it is needed in some test cases because of command format. For example, to check if the GPIO value has changed as expected. For I^2C and SPI interface, the *data* is written to the device, *size* tells how many bytes are read from the device, and the *expected* value for read operation can be configured in the test case.

Diagonally half-colored attributes in Figure 30 could be configured in the test case, or those could be added to hardware configuration also. For the example products used in this thesis the *data*, *size* and *expected* attributes remain the same for both products. These attributes are related to used components which are the same in example products, and they could be used in new products that use the same components. Based on this realization, the component configuration was added. Figure 31 shows the changes based on analysis.

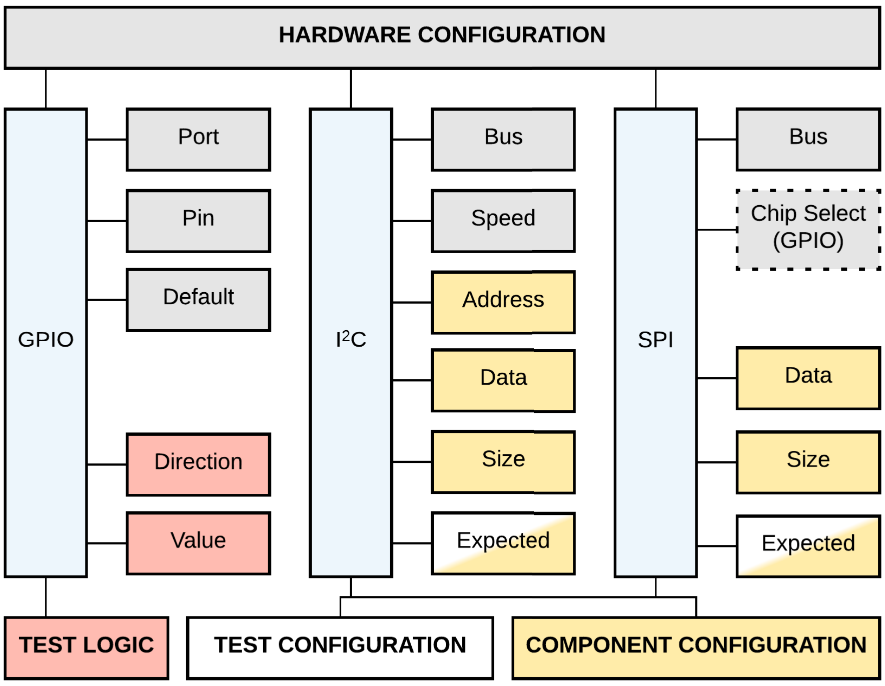


Figure 31. Required test attributes divided between test logic, hardware, test and component configuration.

In Figure 31 the test logic and component configuration were added to reflect findings from the first configuration model. Test logic itself dictates some of the attributes used in testing. In the example products, this can be seen in the way GPIO pins are controlled. These attributes rarely require variation. For I^{2}C and SPI components, the attributes come mainly from component specification, excluding those that are directly related to hardware wiring. Component specification dictates what registers & *addresses* to use, and what *data* can be written to them, as well as the *size* for written and read data. The specification also has static register values, such as component id register, which is used as *expected* value for id register read test. Sensor values are related to the environment in which the tests are executed. For these tests the *expected* value comes from test configuration which is based on test requirements (Table 8. p48).

The addition of the component configuration helps to separate the hardware dependencies from the test case. For example, if the I^{2}C RTC would be changed for other manufacturers module, it is likely that only the component configuration would need to be changed while the test logic would remain the same. RTC modules have a similar interface to set time and to retrieve it, so changes in test logic might not be needed. If changes were required, that would indicate that a new variation point is needed. If RTC module with an existing component configuration is used in another product, the component configuration can be reused. Component data will remain the same even though the wiring of the component to the new product would be different.

Hardware configuration has a very low reusability value. It is directly tied to hardware schematics, which are subject to change even for variant products from the same product platform. For this reason, the hardware configuration should contain a minimal amount of the configuration data that is directly tied to schematics. This enables the reuse of other configuration data, which is more generic in nature.

Based on the analysis done for the attributes used in testing, four different sources were identified which define the attribute value. They are presented in the Table 12.

Table 12. Identified sources for attribute values ordered by their reusability

Reusability	Source
1.	Test logic
2.	Component configuration
3.	Test configuration (test requirements)
4.	Hardware configuration

Based on the information summarised in Table 12 and in Figure 31 the attributes related to test logic were decided to be left to test cases as no reason could be identified where those values could change. Other attributes were divided into configuration files based on their identified source.

5.9 Implementing the required variation

After the variation model was outlined, the hardware, component, and test configuration files were added to test resources.

Test cases were modified so that each attribute used in the test case was acquired from one of the three separate configuration files. Implementation continued until all the test cases in the test suites passed, and test requirements were met, after which the second example product was accepted.

After the second example product was accepted, the production tests were performed again for the first example product. There were several failures in execution, but they were related to missing configuration values. The hardware configuration file was not yet implemented for the first example product. During initial implementation, these values had been inserted directly to test cases. These values were removed when configuration files were taken into use.

After the hardware configuration file was created for the first product, all the test cases passed again. A fair amount of exploratory testing was done to discover hidden problems related to configuration files or test execution. When no major issues were detected, the prototype was accepted ready for evaluation.

6 Evaluation

This section focuses on the evaluation of the designed and implemented artifact, the production test automation prototype. First, the design phase of the artifact is evaluated in the context of the production testing domain. The implementation itself is not covered in the evaluation. Instead, focus is on evaluating the final version of the artifact used in the demonstration.

Gregor & Hevner (2013) state that artifact should be evaluated in terms of criteria that can include the validity, utility, quality, and efficacy. Validity is evaluated by determining if the artifact is working in its intended purpose: does it achieve its goal?. The utility is evaluated by assessing if the artifact has value outside of the development environment (Gregor & Hevner, 2013). Quality can be evaluated against the quality factors of the design process as well as the produced artifact (Hevner, March, Park, Ram & others, 2004). Efficacy can be evaluated to which degree the artifact produces its desired effect without addressing situational concerns (Venable, Pries-Heje & Baskerville, 2012).

In this thesis, the utility and validity of the artifact have the highest priority. The quality of the design process is considered to have higher priority than the artifact itself. Artifact is prototype implementation, and high quality is not expected for it. Evaluating the efficacy of the artifact is somewhat problematic as the artifact produces pass-fail information of the test run. In the scope of this thesis, the efficacy of the design has more value.

Authors Venable et al. (2012) note that there is evaluation gap present in DSR research. They propose different strategies for conducting the evaluation to improve its effectiveness (Venable et al., 2012). However, these strategies were not applied in this thesis to conduct the evaluation. The evaluation was done by assessing validity, utility, quality, and efficacy based on the design and functionality of the prototype against the research question without using an evaluation framework. This poses some limitations for the quality of the evaluation. The evaluation strategy and rationale behind it are presented next in order to mitigate the lack of an evaluation framework.

6.1 Evaluation strategy

This section describes the evaluation criteria set for the prototype and the overall strategy for evaluation. First, the requirements for the prototype are presented, then the rationale for evaluation. Later each requirement is evaluated separately.

Set evaluation criteria: Evaluation criteria for to prototype can be seen from Table 13.

Table 13. Requirements for prototype.

no.	Description	Linked to
R1.	All test cases can be executed	Table 6 (p.44) & 7 (p.47)
R2.	Test cases pass or fail based on test requirements	Table 8 (p.48)
R3.	Verification can be seen from test report	Section 5.2 (p.40)
R4.	Test cases can be executed for multiple products	Research Question

Rationale behind the evaluation Requirements in Table 13 is presented next.

The purpose is to evaluate if the prototype can test multiple products (R4). In order to evaluate multiple products, the production test automation system must first be proven valid using one product. Test automation is verified through requirements R1-R3. Criteria for R1-R3 are set based on minimal requirements for a production test automation system in a real production environment. This ensures that the prototype is valid for its intended usage and environment.

R1 was set to evaluate if target hardware is fully covered during testing. Each production test is designed to verify some hardware functionality from that device, so skipping tests would mean that some hardware functionality is not tested. R1 evaluates the functionality of the system and the scope of testing.

R2 was set to evaluate the accuracy of the test cases against limits set in test requirements. This is set to verify that tests accurately measure what they are supposed to. R2 evaluates the validity of the test cases.

R3 was set to evaluate common automation requirements for reporting. For pass or fail test cases, the test report has to show the verification step, for example, what value is compared against what limit. In production test environment, the reason for either passing or failing has to be demonstrated. R3 evaluates that the prototype is suitable for its intended purpose.

R4 was set to evaluate if the prototype answers the research question set for this thesis. When the functionality of the production test automation and test cases have been evaluated through R1-R3, the prototype can be evaluated against R4.

6.2 Evaluation of test automation (R1)

The evaluation of R1 is meant to verify the prototype execution in practice. However, this requirement on its own does not state anything about the quality of the production test automation system or the designed test cases. The design of test automation and test cases are evaluated in this section even if a direct requirement for the design was not set for evaluation. The author believes that the quality of the test automation and test cases are prerequisites for the evaluation of R1.

At the beginning of the design process the generic software testing framework demonstrated by Alsmadi (2012) was taken as model for the test automation architecture. Generic software testing framework is presented in Figure 8 (p.17). For the purpose of this thesis the framework was separated in two halves, seen in Figure 25 (p.40). Purpose was to divide the framework between work done manually and by using automation.

First design section covers the process of building the formal model seen in the generic software testing framework (Figure 25, p.40). Process for defining production testing requirements was done based on the target hardware. The design section describes the target hardware and the intended functionality of the two example products used in production testing.

Target hardware and its intended usage forms the basis for designing what must be tested. Target hardware is presented in Table 4 (p.43) and used components in Table 3 (p.42). The intended usage of the example products was defined in Table 5 (p.43) and it forms the basis for test requirements: what must be tested from the example product to verify operational status of the hardware to provide required functionality. Planned tests based on test requirements are presented in Table 7 (p.47). Aforementioned design process is comparable to one used for real products. This was done to ensure that the scope of testing was realistically set.

After the scope of testing was set, the test cases were designed. The method used to define test cases based on the target hardware is similar to that used for real products. This method is described in Section 5.3.3 (p.44). Tests were designed realistically to avoid creating them just for demonstration purposes.

After the test cases were designed acceptance criteria for defined tests were set and source for the criteria presented, as seen in Table 8 (p.48). This verifies that there are clear criteria set for test verification.

Test requirements, test acceptance criteria, hardware schematics and specifications form the formal model seen in the generic software testing framework (Figure 25 p.40). A formal model was used to set test inputs for test cases and expected output values for verification. The design of the formal model covered the manual work done to implement production test automation based on the generic framework.

The remaining part of the generic software testing framework (Figure 25 p.40) was implemented using Robot automation framework. Robot automation framework was the engine used to carry out the testing process, verify the test results, and to report them. Even though the production test automation implementation is at the heart of the designed artifact, the implementation itself offers little to the topic and is not covered in this thesis.

The author's opinion is that prerequisites for the evaluation of R1 have been satisfied through the design and documentation summarised here.

Evaluation of R1 1 was verified by executing the implemented tests for the example product 1. The test report was compared against the Table 7 (p.47) to verify that all tests had been executed, thus reaching the 100% of the test coverage set by the test requirements.

6.3 Evaluation of test asset validity (R2)

After the R1 had been verified, the next step was to verify the validity of the test cases. This was done by comparing the verification results from test report to acceptance criteria, seen in Table 8 (p.48). All the test cases had been passed according to the acceptance criteria.

To verify that test cases also fail according to the acceptance criteria, the test limits were changed to unreasonable values. For visual inspection tests the fail option was given when prompted by test automation. All the test cases produced fail result, thus verifying the validity of implemented tests.

6.4 Evaluation of test report (R3)

Test report was already used in verification of R1&R2, which partially verifies that the report contains the required information. Required information from the executed test run in the production test context is:

- name of executed test
- command send to device
- response read from the device

- comparison values in verification
- test result

Test name, comparison values, and test result had already been used in the evaluation of R1&R2 and verified to be valid. Command send to the device was checked against the test implementation, and read responses were verified by executing commands manually. The required information was found from the test report, and it matched the test implementation and manual verification. This verified that reporting fulfills requirements in the production testing environment.

6.5 Evaluation of support for multiple products (R4)

After the validity of test automation and implemented tests were verified for the first example product, the design continued to verify the second example product. The process of how variation support was added is documented in Sections 5.7 & 5.8. When design was considered to be ready, a demonstration was held to show the functionality of the prototype system.

The demonstration setup contained one laptop that worked as a test environment. Test environment had Robot automation framework as test runner, and it contained the implemented test suites and cases as test assets. Both example products functioned as target hardware to be tested. The test environment contained one component configuration, one test configuration, and two hardware configuration files to be used with matching example products. Configuration files were passed via command line parameters to test automation. Also the parameter `exit on failure` was used to ensure that failures were not missed during the test run.

Test suites were executed for the first example product and verified that all test cases had passed. Then the first example product was replaced by the second product, and the hardware configuration file was changed to the command line parameters. Test suites were executed again and verified from results that all tests had passed. This demonstrated that the prototype system was capable of testing two example products without the need to modify test cases when the required variation was provided via a hardware configuration file. There was no need to change the component or test configuration file as the example products had the same hardware components and common acceptance criteria for the test cases. Figure 32 describes the test environment used in demonstration.

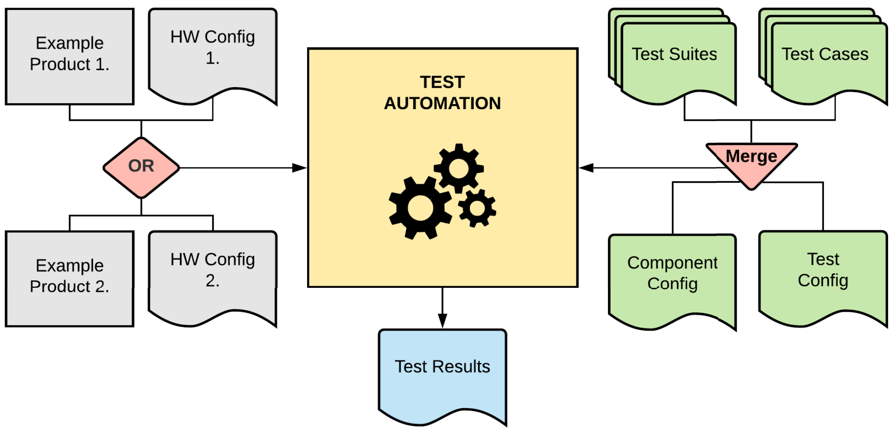


Figure 32. Description of the test automation system used in demonstration. Variation is provided via configuration files.

Product-specific input in Figure 32 is coming from the left. This is the example product and the matching hardware configuration for it. From the right comes more generic input. Test suites and cases are merged with configuration data coming from component and test configuration files. Test automation uses input from both sides during execution and generates the test results.

Even though the demonstration was done using simple example products, they contained many of the same interfaces and components found in real products. Artifact succeeded in demonstrating that hardware dependencies can be separated from the test logic and that test cases can be dynamically configured during a test run in a production testing context. Dynamic configuration during test execution makes it possible to test multiple products using the same test environment. Prototype fulfills R4 by demonstrating that multiple products can be tested by it, even though the ”multiple” in this thesis is precisely two.

Evaluation criteria R4 directly addresses the research question of this thesis:

> *How to create production test automation system capable of testing multiple products?*

The prototype system and its design do not solve the research question, as the problem domain for IoT device production testing are much larger than can be covered in one study. Instead, prototype and its design try to provide utility to study this problem further. How can the existing single system engineering approach be moved towards platform engineering, such as SPLE? Prototype and its design demonstrate one solution to separate dependencies between test assets and hardware.

6.6 Production testing generalization effort

The generalization of the production test automation is part of the greater generalization effort. Section 4.7 (p.35) mentions that PTSW API was required to be used in test automation to communicate with the device. While PTSW provides the API to be used, it is not the only goal for its development. Another goal is to improve portability to new hardware platforms. Development of PTSW was not in the scope of this thesis, but test automation used it to communicate with the device. In fact, without a common command interface between two example products, the generalization of test automation would not have been possible. Figure 33 presents both PTSW (at the bottom) and test automation (at top), which makes it easier to put this thesis in its context.

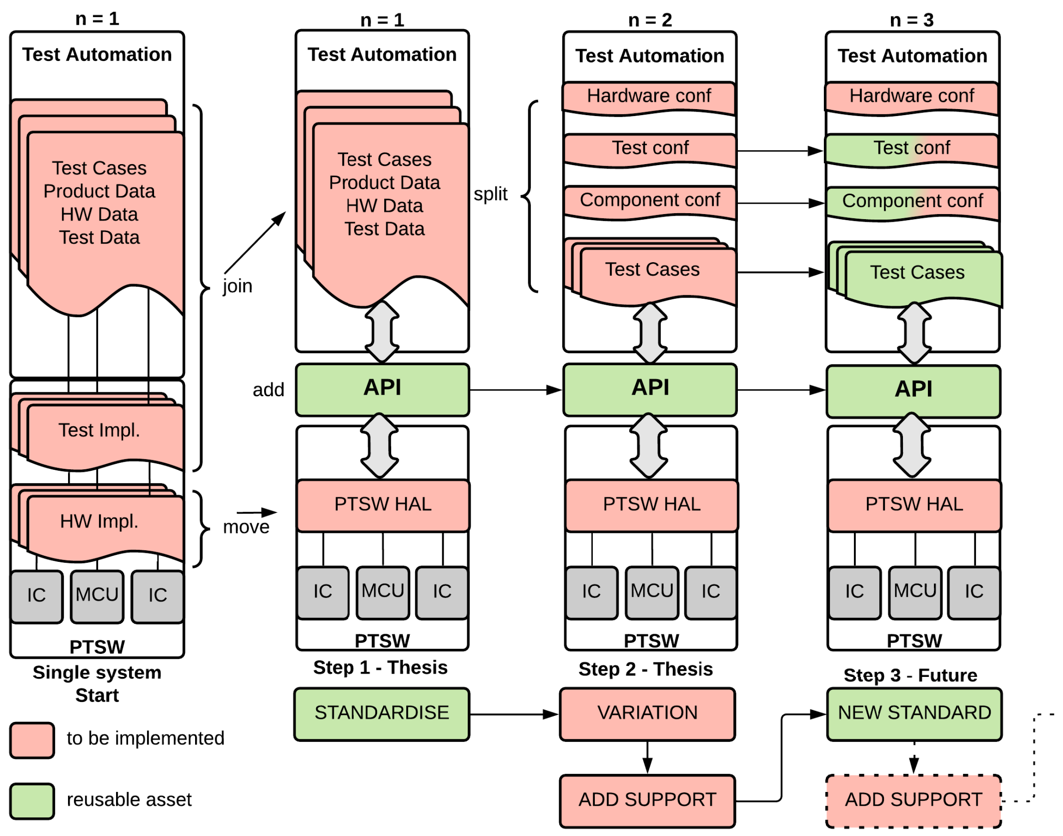


Figure 33. Production testing generalization effort

This thesis focuses on two systems presented in the middle of Figure 33. The leftmost system describes the common single system implementation of PTSW and test automation. There is no defined API between the two, and test cases are directly coupled to functions executed in the target hardware. The red color indicates the need for implementation. Each new system developed by a single system approach requires implementation of all of the assets needed, PTSW, test assets, and test automation. The rightmost system indicates the future, what is the goal beyond the work on this thesis. As such, it is not part of the evaluation done here but to give context for work done in this thesis. This is covered more in Discussion Section 7.

The second system from the left describes the implementation done in this thesis for the first example product. The PTSW API had already been designed by T. Räty, who was also the technical advisor for this thesis. PTSW HAL was implemented by Sankila (2021) as part of his own study, and it was extended by adding support for SPI interface also by him. For this reason, the API is colored with green color (reusable asset).

The third system from the left describes the implementation done in this thesis for the second example product. Reusing test cases from the previous step required the separation of hardware dependencies from the test logic. This was achieved by introducing hardware, test, and component configuration file. During this phase, test automation assets could not be reused as the separation of test logic, and hardware dependencies had to be implemented.

The rightmost system shows the desired goal of increasing the reusability of test assets, but it cannot be evaluated at this point.

Where PTSW API&HAL has succeeded on-device software implementation is the abstraction of hardware dependencies. PTSW API operates on a low hardware abstraction level, required by the changing hardware. In order to minimize the implementation effort for new hardware, the test logic needed to be extracted from the device to test automation in its entirety. When tests have been moved to test automation, naturally, there are benefits in reusing them from product to product.

This thesis succeeded in creating one solution to separate test logic and hardware dependencies from each other and to demonstrate this using two example products. The solution is certainly viable, but it is too early to evaluate whether it can solve all variability issues for a wider range of hardware platforms. Ability to solve variability and reusability of test assets can be evaluated after the prototype is used for additional hardware platforms. For this reason, any claims of prototypes efficacy are not stated. The method for finding variation points (pseudo algorithm 1, p.53) can be used in future work together with the prototype. This demonstrates that artifact also provides utility to practice.

6.7 Evaluation of design science guidelines

Design science guidelines presented in the research problem and methodology section are revised to evaluate how well these guidelines were followed in this thesis.

Guideline 1. Design as an Artifact.

Design science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation. (Hevner et al., 2004)

Prototype of test automation system capable of testing two different example products was implemented. Its design has been presented in the design section of this thesis, Section 5 (p.37).

Guideline 2. Problem relevance

The objective of design science research is to develop technology-based solutions to important and relevant business problems. (Hevner et al., 2004)

Problems of developing production testing in single system approach has been presented in this thesis to highlight the relevance of more generic approach, Sections 4.5&4.6 and 5.

Guideline 3. Design evaluation

The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. (Hevner et al., 2004)

Evaluation of the prototype was done by demonstrating its functionality using two example products. Production tests developed for demonstration are comparable by their scope and functionality to those used for real products. Production tests fulfilled the test requirements for two different products, given that the product-specific configuration file was passed to test execution as a parameter.

Guideline 4. Research contributions

Effective design science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies. (Hevner et al., 2004)

Variation model is presented in design Sections 5.7 & 5.8 to demonstrate how hardware dependencies were separated from the test logic. This method can be used for other tests which were not used in this thesis to increase the reusability of test assets. Prototype implementation provides utility to test additional hardware platforms or products.

Guideline 5. Research rigor

Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact. (Hevner et al., 2004)

Design Section 5.2-5.9 covers test automation architecture, defining requirements for production testing, test asset development and demonstrates how needed variation was successfully introduced in the scope of this thesis. Evaluation Section 6 covers its evaluation.

Guideline 6. Design as a search process

The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment. (Hevner et al., 2004)

Existing knowledge was used to form the base for the test automation framework. The test automation framework was created based on the existing generic software testing framework model. Test requirements were defined following practices used in the production testing domain. The solution to the problem of how the required variation could be implemented was searched through experimentation and problem analysis.

Guideline 7. Communication of research

Design science research must be presented effectively to both technology-oriented and management-oriented audiences. (Hevner et al., 2004)

Domain overview Section 4 and the problem description in the Section 5 is aimed for management-oriented audience. They highlight the problem relevance without the need to understand the technologies behind production testing. Design Sections 5.2-5.9 together with the produced artifact is aimed for technology-oriented audience.

7 Discussion

This thesis focused on the creation of a test automation system capable of testing multiple products using common test assets. The idea for this work came from the practice where the increasing number of production test automation systems was recognized as a growing technical challenge. Each new IoT product increases the amount of test automation systems needed when developed in a single system engineering approach.

Literature was reviewed for existing solutions for creating test automation systems for embedded hardware testing. Literature specifically to the production testing domain is quite limited, if not non-existing. While most of the literature focuses on testing the software using the target hardware, the embedded system testing literature provided insight into testing the hardware itself.

Example products used in this thesis were designed in collaboration with Sankila, who was working on his Master's Thesis related to the software used in the device during production testing (Sankila, 2021). Assembly and the implementation of the PTSW, for example products, were done by him. The construction of these example products was beneficial for test automation development as the target hardware to be tested was readily available. Example products shared same components that can be found from the actual products. This was done to ensure that production testing would happen in a realistic setting.

7.1 Comparison to prior work

Literature for production testing context is scarcely available. Manufacturing-related data is often categorized as business sensitive, critical or classified data. Unfortunately, this also affects the production testing and how it can be communicated. Comparison and publication of production testing information that would reveal development or process-related specificities are difficult. This affects both finding such data and also the possibility to publish it.

This left open the possibility to compare experiences to those published in the literature. Literature related to techniques and challenges for variation management in SPLE was a primary interest. Material needed to be later further limited specifically to the embedded system domain.

While there is an abundance of material for variability management in SPLE in the software domain, it is not directly comparable to the embedded system domain. Article *Managing Variability in Software Product Lines* highlights many of the challenges faced in variability management based on information gathered by interviews and systematic literature review (Babar, Chen & Shull, 2010). Few of the main challenges mentioned in the article are related to finding variation, modeling it, and managing the scale of variation (Babar et al., 2010).

Authors Park, Kang & Sugumaran give an example that illustrates the problem of variation scalability in the software domain: combinatorics of 216 boolean features is comparable to the number of atoms in the universe, 33 boolean features compare to human population on earth (Park et al., 2010). In this thesis, variation points were analysed case-by-case. While this can be achievable in the embedded system domain, especially for quite simple IoT-devices, it is not applicable for software product lines that can contain over hundred variable features. On the other hand, the modeling techniques used in the software domain can be excessive for the embedded system domain.

While many of the challenges mentioned in the article by Babar et al. (2010) does not closely relate to this work, it raises a good point that variation points should be kept to a minimum. This issue was also raised in article *Experiences with product line development of embedded systems at Testo AG*, in which the main challenge was reported to be introducing too many variation points early on, based on analysis and modeling (Kolb, John, Knodel, Muthig, Haury & Meier, 2006). Kolb et al. say that this led to the development of over-complex software components, an issue that was also raised by Pohl (2005). Unneeded variation was avoided in this thesis by adding variation only when there was an apparent reason for it. Either by failing production test case or the problem was evident otherwise.

The method used in this thesis can be applied to relatively simple systems, in which testing is also possible. Lack of tools and techniques to test variability was raised by Kolb et al. (2006) as one of the challenges of developing an embedded system using SPLE. Difficulty to test product highlights also the difference of end-user software compared to production test software. Testing is the core functionality the PTSW provides, while the end-user software focuses on intended usage, and testability is not its main concern. It can be stated that testability is guaranteed in PTSW as it is the core functionality it provides. For this reason, testability will not be a problem in production testing domain even if it is recognized as a challenge in the embedded system domain.

Testing and testability are often in focus concerning the SPLE development in the embedded system domain. *Testing environment for embedded software product lines* was the closest article to work done in this thesis: it describes the usage of simulation to test interconnected embedded systems (Kuroiwa & Kushiro, 2015). The example given in this article is the network of air conditioning system that includes several different components such as a control unit, fans, and remote control. Testing system as a whole is called system testing (Myers et al., 2012): category of testing that production testing also belongs to. Even though the topics are very closely aligned, the article by Kuroiwa & Kushiro (2015) focuses on how embedded systems can be used to simulate parts of air conditioning system that are not present in the test environment; it did not cover the testing of embedded systems themselves. Even that work in this thesis did not use interconnected devices, the approach demonstrated by Kuroiwa & Kushiro (2015) could be used in the future to replace systems that are not part of the production testing effort.

7.2 Answering the research question

To address the research question, a better understanding of the problem needed to be established first. This information was gathered in a somewhat simplistic manner even though it still proved to be effective. Tests were implemented for the first example product, and implementation continued until requirements for production tests were met. Then tests were executed for the second example product to see what breaks. Analysis of these failures helped to understand what factors are behind them. The obvious first factor was that the wiring between example products was different, so GPIO port&pin mapping had to be changed between products.

One of the key findings was not related to failures but to the question of why the I^2C tests had worked for the second example product. This was unexpected and surprising at first. Analysing the test logic and data exchanged between test automation and the device helped to separate the hardware-specific data from the component-specific data.

Hardware-specific data was related to how the bus was indexed in the PTSW implementation. There are no rules that dictate that RTC clock must be connected to bus index 0 and seven segment display to bus index 1. They were similarly indexed in both example products, which was seen as a coincidence. There are no guarantees that they would be similarly indexed for a third product. The I^{2}C API in PTSW hides the underlying hardware dependencies from the command interface. What GPIO pins are used for I^{2}C bus are handled by the PTSW HAL implementation, and therefore they are abstracted away from the API. Based on this fact, the bus index and speed attributes were placed to hardware configuration as they need to be defined per product.

SPI is similar to I^{2}C interface in that it abstracts the GPIO pins used for communication from the command interface. However, SPI uses a different mechanism for selecting the component to which MCU is communicating. The method for selecting an SPI device is a chip select signal (GPIO), and it is not included in the SPI API currently. Because the chip select must be separately handled for SPI, the configuration for I^{2}C and SPI could not be unified.

The rest of the data between test automation and device for I^{2}C tests also remained the same, but this was due to the fact that the used components were the same. What registers & values were written to the component were mostly based on the components specification. This data does not change between products if the used component is the same. Component-specific data was separated into its own configuration to make it easier to reuse.

The other two sources for data were recognized during the analysis. One reason was that the test logic directly dictated the values. Tests need to set GPIO direction based on the operation executed on them. This logic is coming from the GPIO specification itself, and no valid reasons were identified why this information should be moved out from the test implementation. It will remain the same as long as the GPIO specification does not change.

The last source for the data was coming directly from the test requirements. These were the acceptance limits for air quality sensor values. It was identified that this data should not be implemented in the test case or test suite as they are subject to change. For this data, the test configuration file was created.

Information discovered during analysis is not groundbreaking by any means, but it helped to generalize test assets in the context of production testing. Production testing, by definition, is always coupled to the hardware to be tested. But there are ways to manage this coupling, for example, by the use of configuration files. Configuration files enable dynamically configure tests during execution, which in turn helps in the creation of more generic test cases. Dynamically configured tests allow several products to be tested using the same test environment. This was successfully demonstrated using the prototype system in the evaluation phase of this work.

7.3 Future work

Several possibilities for future improvements and areas for further work were discovered during the design and implementation phase. A few of the most interesting ones are presented here.

7.3.1 Discovery API for PTSW

When the source for the needed configuration was considered during variation analysis, the question arose: why the required information is not asked from the device? Obviously, this functionality is not currently supported, but there is no technical reason why it could not be implemented based on current understanding.

The device itself would be a logical place to store the hardware and component configuration data. The device could store the information of what peripherals are attached to it and how they are connected to the MCU. This information would not have to be added to the PTSW HAL implementation; it could be appended to the PTSW binary.

It is almost certain that each device would have enough free flash memory to allocate a few kilobytes for the configuration data. Even a kilobyte could be enough if data is compressed. There are existing technical solutions that can ensure that configuration data would be flashed to the wanted memory address, and it would not interfere with the normal operation of the PTSW. Configuration data would be erased from the device when end-user software is flashed to the device.

Test automation could request the device to send its own configuration data. This data could contain both the hardware and component configuration. All that PTSW would need to know is how many bytes it needs to read from its memory and the address where it reads from. This functionality would keep the PTSW side of implementation trivial.

Test automation could identify the available interfaces and components from the configuration data it receives. This would make handling the configuration files easier. The idea could be taken even further. Test automation could identify what tests the device supports from the received data and generate the test suite that would cover all available interfaces. This would be a major step towards automated testing and automated test generation.

7.3.2 Using the artifact to verify new hardware platforms

Hardware components in the used example products were connected to movable prototype shield, known commonly as proto shields. Proto shield is a separate circuit board into which components can be wired or soldered. Proto shields can be connected to development kits and evaluation boards using standard pin headers. Arduino pin headers were used in example products used in this thesis. Proto shields make it possible to move components from one development kit to another in just a few seconds.

During the work in thesis, the components on the proto shield were already tested using two different hardware platforms. Designed artifact, the prototype test automation system, already contains a majority of the required test assets. The only thing missing for the new hardware platform is the hardware configuration file.

When the implementation of the PTSW starts for the new hardware platform, the proto shield could be connected to it. When the GPIO interface is implemented, the GPIO test suite could verify that the button and LED are working correctly. When I^{2}C has implemented the functionality of the RTC, and 7-segment display could be verified.

In this case, the purpose of using test automation would be to verify the implementation of the PTSW, not the hardware or the components. The functionality of the implemented hardware interfaces could be directly checked against a known working set of test cases, and results would be directly visible from the LED, 7-segment display, and OLED.

The concept that test cases are runnable before the implementation is done has been used for software testing for quite some time. For example, test-driven development (TDD) is largely based on this idea. Even though this concept is not new, it is rarely used in embedded systems programming with the actual target hardware. Code is usually tested by unit tests, and the testing is pure software testing by its nature. Using the proto shield and test automation prototype enables hardware testing to begin with the target hardware right at the beginning of the embedded software development.

7.3.3 Test case library together with components

As was briefly mentioned in Section 5.8 (p.55), when component configuration file is created for specific component, that data remains the same. The component configuration file can be reused when the same component is used in another product.

Also, the used test case for a specific component should be saved together with the configuration file. The test case would be already verified using one product before it is added to the library. This would ensure that the component library contains a verified version of the test case together with component configuration and would make it easy to reuse the test case in another project.

At first, test cases and configuration files need to be added often, but the reusability of test and configuration assets would increase over time. When the component library has grown large enough to contain a wide variety of components, the test design for a new product could become trivial. This would be the case if most of the used components could be found from the component library.

The creation of this library does not require much development effort. The component configuration file needs to be implemented during test automation development for a new component. It would be just a matter of storing this information to make searching and retrieving components from the library easy.

8 Conclusion

This Master's Thesis created a test automation prototype capable of testing multiple products using the common test assets. The research problem was raised from the practice where an increasing number of test automation systems developed for production testing purposes is seen as the growing technical challenge. Developing test automation systems per product is recognized to have two negative factors: development costs of the test automation and tests assets per product and the increasing maintenance effort of these systems. Work on this thesis aimed to find solutions to how test automation could be generalized to support testing of multiple products using the same test environment.

This study reviewed the literature to find known solutions to generalization. Test automation framework was designed based on the generic software testing framework by Alsmadi (2012). The process of defining test requirements for used example products is covered in the design section. Also, the method for defining the test cases based on the target hardware is demonstrated to give a better understanding of the core concept of production testing: verification of the operational status of the target hardware. Lastly, the design section covers by what means the support for multiple products was searched in this thesis and how the needed variation was introduced to the test automation system. Test automation design was operationalized as a prototype, and its functionality was evaluated by running production tests for two example products.

The prototype system succeeded in configuring test cases dynamically during execution, making it possible that both of the example products could be tested using the same test assets. The solution has its limitations due to the scope of this study, only two different hardware platforms were used, and the number of attached components is limited. However, the prototype succeeded in achieving its goal. It demonstrated the capability to test multiple products, directly addressing the research question set for this thesis.

Several possible avenues for further development were discovered during the design phase. The discovery API mentioned in the Section 7.3.1 has interesting prospects for both development and further research. If required configuration data for target hardware could be directly retrieved from the target device, it would enable new concepts for testing. Test automation could use the configuration data to adapt test suites to exclude or include additional tests based on the discovered interfaces and components. Configuration data read directly from device would help generalize the test automation further.

In addition, several possibilities were recognized where developed artifact could provide utility for practice. One of these possibilities is the ability to verify the PTSW implementation for new hardware. This concept is discussed in the Section 7.3.2. Implementation of the PTSW could be used as a programming exercise in embedded system programming training. Test automation prototype would provide an easy way to execute production tests for known components.

Bibliography

Alsmadi, I. *Advanced automated software testing: frameworks for refined practice*. Information Science reference, 2012.

American Society for Quality website. Quality Assurance vs Quality Control: Definitions & Differences. Retrieved 6.11.2020 from https://asq.org/quality-resources/quality-assurance-vs-control, 2020.

American Society for Quality website. What is Cost of Quality (COQ). Retrieved 6.11.2020 from https://asq.org/quality-resources/cost-of-quality, 2020.

Babar, M. A., Chen, L., and Shull, F. Managing variability in software product lines. *IEEE Software*, 27(3), pp. 89–91, 94, 2010.

Bailey, B. Balancing The Cost Of Test. Retrieved 16.6.2020 from https://semiengineering.com/balancing-the-cost-of-test, 2014.

Bose, S. Testing in Production: A Detailed Guide. Retrieved 13.11.2020 from https://www.browserstack.com/guide/testing-in-production, 2020.

Bourque, P. and Fairley, Richard E, e. a. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. Los Alamitos, CA: IEEE Computer Society Press, 2014.

Broekman, B. and Notenboom, E. *Testing embedded software*. Addidon-Wesley, 2008.

Chopra, R. *Software Testing: A Self-Teaching Introduction*. Stylus Publishing, LLC, 2018.

Cohn, M. *Succeeding with agile: software development using Scrum*. Addison-Wesley, 2013.

Dustin, E., Rashka, J., and Paul, J. *Automated software testing introduction, management, and performance*. Addison-Wesley, 1999.

Encyclopedia Britannica. Automation - Advantages and disadvantages of automation. Retrieved 22.11.2020 from https://www.britannica.com/technology/production-system/Important-considerations, 2020.

European Comission website. CE Marking - Manufacturers. Retrieved 10.12.2021 from https://ec.europa.eu/growth/single-market/ce-marking/manufacturers_en, 2021.

Fewster, M. and Graham, D. *Software test automation: effective use of test execution tools*. Addison-Wesley, 1999.

Finnish Meteorological Institute. Air quality. Retrieved 7.6.2021 from https://en.ilmatieteenlaitos.fi/air-quality, 2021.

Goericke, S. *The future of software quality assurance*. Springer Nature, 2019.

Graham, D., Fewster, M., and Copeland, L. *Experiences of test automation: case studies of software test automation*. Addison-Wesley, 2012.

Gregor, S. and Hevner, A. R. Positioning and presenting design science research for maximum impact. *MIS quarterly*, 2013.

Groover, M. P. *Automation, production systems, and computer-integrated manufacturing*. Pearson, 2015.

Gupta, S. and Starr, M. *Production and operations management systems*. CRC Press, 2014.

Hevner, A., March, S. T., Park, J., Ram, S., et al. Design science research in information systems. *MIS quarterly*, 28(1), pp. 75–105, 2004.

Hevner, A. R. and Chatterjee, S. *Design research in information systems: theory and practice*. Springer, 2012.

Hevner, A. R., March, S. T., Park, J., and Ram, S. Design science in information systems research. *MIS Quarterly*, 28(1), pp. 75 – 105, 2004.

IEEE Std 1450-1999. IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data. Standard, Institute of Electrical and Electronics Engineers, 1999.

IEEE Std 1505-2010. IEEE Standard for Receiver Fixture Interface. Standard, Institute of Electrical and Electronics Engineers, 2010.

IEEE Std 1547.1-2005. IEEE Standard Conformance Test Procedures for Equipment Interconnecting Distributed Resources with Electric Power Systems. Standard, Institute of Electrical and Electronics Engineers, 2005.

IEEE Std C37.121-2012. IEEE Guide for Switchgear - Unit Substation - Requirements. Standard, Institute of Electrical and Electronics Engineers, 2013.

ISTQB Glossary. International Software Testing Qualifications Board Glossary. Retrieved 16.6.2020 from
https://glossary.istqb.org, 2020.

Jorgensen, P. C. *Software Testing: a Craftsmans Approach, Fourth Edition*. CRC Press, 2013.

Kinsbruner, E. What Is Test Automation. Retrieved 18.11.2020 from
https://www.perfecto.io/blog/what-is-test-automation, 2019.

Kolb, R., John, I., Knodel, J., Muthig, D., Haury, U., and Meier, G. Experiences with product line development of embedded systems at Testo AG. In *10th International Software Product Line Conference (SPLC'06)*, pp. 10 pp.–181, 2006.

Kuechler, W. and Vaishnavi, V. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology, 2nd Edition*. CRC Press, 2015.

Kuroiwa, T. and Kushiro, N. Testing environment for embedded software product lines. In *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, pp. 1–7, 2015.

Linden, F. v. d., Schmid, K., and Rommes, E. *Software product lines in action: the best industrial practice in product line engineering*. Springer, 2010.

March, S. T. and Smith, G. F. Design and natural science research on information technology. *Decision Support Systems*, 15(4), pp. 251–266, 1995.

Martin, R. C. *The clean coder: a code of conduct for professional programmers*. Prentice Hall, 2014.

MCU Market History and Forecast 2016-2023 - HardwareBee. Retrieved 16.6.2020 from https://hardwarebee.com/mcu-market-history-and-forecast-2016-2023, 2019.

Merriam-Webster. End user. In Merriam-Webster.com dictionary. Retrieved 6.11.2020 from
https://www.merriam-webster.com/dictionary/end%20user, 2020.

Meszaros, G. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2007.

Myers, G. J., Sandler, C., and Badgett, T. *The art of software testing*. John Wiley & Sons, Inc., 2012.

Park, S., Kang, K. C., and Sugumaran, V. *Applied Software Product Line Engineering*. CRC Press, 2010.

Pohl, K. *Software product line engineering : foundations, principles, and techniques*. Berlin: Springer, 2005.

Pries, K. H. and Quigley, J. M. *Testing complex and embedded systems*. CRC Press, 2018.

Project Management Institute. *A guide to the project management body of knowledge (PMBOK® Guide)*. Newtown Square, PA: Project Management Institute, 2017.

Purao, S. Design research in the technology of information systems: Truth or dare. *GSU Department of CIS Working Paper*, 34, 2002.

Ramos, D. A Guide to Automation Frameworks. Retrieved 4.8.2021 from
https://www.smartsheet.com/test-automation-frameworks-software, 2021.

Robot Framework. n.a. Retrieved 6.12.2021 from https://robotframework.org, 2021.

Rungta, K. What is System Testing? Types & Definition with Example. Retrieved 9.8.2021 from
https://www.guru99.com/system-testing.html, 2021.

Sankila, Eero, I. Reusable firmware for IoT device production testing. [Unpublished master's thesis], University of Oulu, 2021.

Simon, H. A. *The sciences of the artificial*. MIT press, 2019.

Sugumaran, V., Park, S., and Kang, K. C. Software product line engineering. *Communications of the ACM*, 49(12), pp. 28–32, 2006.

Tech, S. *Quality Assurance: Software Quality Assurance Made Easy*. CreateSpace Independent Publishing Platform, 2016.

Techopedia. Test Automation Framework. Retrieved 11.4.2021 from
https://www.techopedia.com/definition/30670/test-automation-framework, 2021.

Vaishnavi, V., Kuechler, W., and Petter, S. Design science research in information systems. Retrieved 15.11.2021 from http://www.desrist.org/design-research-in-information-systems/, 2019.

Vance, S. *Quality code: software testing principles, practices, and patterns*. Addison-Wesley, 2013.

Venable, J., Pries-Heje, J., and Baskerville, R. A comprehensive framework for evaluation in design science research. In *International conference on design science research in information systems*, Springer, pp. 423–438, 2012.

Xiao, P. *Designing embedded systems and the internet of things (IoT) with the ARM® Mbed™*. John Wiley & Sons, Inc., 2018.