Tampere University

Riku Pääkkönen

AN APPROACH FOR MATURE SOFTWARE TESTING IN INDUSTRIAL SYSTEMS

Factory Automation
Examiners: Professor Jose Martinez Lastra,
University Instructor Luis Gonzalez Moctezuma
Master of Science Thesis
Feb 2021

ABSTRACT

Riku Pääkkönen: An approach for mature software testing in industrial systems
Master of Science Thesis
Tampere University
Master's Degree Programme in Automation Engineering
February 2021

The role of software in industrial systems is constantly increasing: there is more software, the complexity of the software increases and there are more dependencies between different software. As the role of software increases, so does the number of software errors and the adverse impact of the errors. This creates a growing need for software testing. The importance of testing is often neglected, and testing is often the first phase of work that gets overlooked when schedules are tight or resources run out. This does not make the need for testing disappear and can backfire when the project is mature and should be released.

The literature review indicates that industrial software testing does suffer from low automatization levels, lack of regression testing, and poor organization of testing. The testing of industrial systems has some special characteristics that should be covered when testing is planned for industrial systems.

This thesis describes a three-phase approach for implementing testing as a mature software project in industrial systems. The first phase considers the project before testing: economic feasibility, resource mapping, and management support. The second phase includes project analysis, planning of tests and test prioritization, and development of tests. The main component of this phase is a model for analyzing the project by feature and assign priority for the testing of each feature. The third phase concludes the approach to maintenance and continuity of testing in both technical and organizational standpoints.

The approach was used in practice in an industrial software project that is in a mature phase of development. With the help of the approach, the project was successfully analyzed and an initial testing plan with prioritization was created. The implementation of testing based on the plan was started during the writing of this thesis.

Keywords: software design, industrial software, software testing, automated testing, project management

TIIVISTELMÄ

Riku Pääkkönen: Lähestymistapa kypsien teollisuusympäristössä toimivien ohjelmistojen testaamiseen
Diplomityö
Tampereen yliopisto
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Helmikuu 2021

Ohjelmistojen rooli teollisuusympäristössä kasvaa jatkuvasti: ohjelmistoja on enemmän ja ne ovat monimutkaisempia sekä riippuvaisempia toisistaan. Ohjelmistojen roolin kasvaessa myös ohjelmistovirheiden määrä sekä niiden vaikuttavuus ovat kasvava ongelma, mikä luo tarvetta paremmalle ohjelmistojen testaamiselle. Testaamista ei välttämättä aina nähdä tärkeänä ja se on usein työvaihe, jota ei oteta vakavasti ja jota laiminlyödään ensimmäiseksi resurssi- tai aikataulupaineiden alla. Tarve testaamiselle ei kuitenkaan katoa ja testaamisen tärkeys voi korostua vasta projektin julkaisuvaiheessa.

Aiempi kirjallisuus aiheeseen liittyen osoittaa, että teollisuusohjelmistojen testaus kärsii alhaisesta automatisaatioasteesta, regressiotestauksen puutteesta ja huonosta testauksen organisoinnista. Teollisuusohjelmistojen testaamiseen liittyy myös tiettyjä erityispiirteitä, joita on otettava huomioon testauksen suunnittelussa.

Tämä diplomityö luo kolmivaiheisen lähestymistavan, jolla testaus voidaan tuoda osaksi jo olemassa olevaa ohjelmistoprojektia. Lähestymistavan ensimmäinen osa koostuu tilanteesta ennen testaamista: projektin taloudellinen analyysi, resurssikartoitus ja johdon tuki. Toinen vaihe käsittelee projektin analysointia, testaamisen suunnittelua ja priorisointia sekä testien kehitystä. Tämän vaiheen pääkomponenttina esitellään malli, jonka avulla projekti voidaan käydä läpi ominaisuus kerrallaan ja määritellä testien prioriteetti. Kolmannessa vaiheessa käsitellään testaamisen jatkuvuutta: ylläpitoa sekä teknisestä että organisaation näkökulmasta.

Esiteltyä lähestymistapaa sovellettiin käytännössä teollisuusohjelmistoprojektissa, joka on kypsässä kehitysvaiheessa. Lähestymistavan avulla projekti pystyttiin analysoimaan ja luomaan suunnitelma testauksen implementointia varten sekä implementoimaan ensimmäisiä testejä suunnitelman pohjalta.

Avainsanat: ohjelmistosuunnittelu, teollisuusohjelmistot, ohjelmistojen testaus, automaattinen testaus, projektihallinta

PREFACE

I've been waiting for this moment since spring 2017. That's when I finished my studies besides the Master's thesis and thought that graduation was just around the corner. That was not the case. The writing of this thesis begun in January of 2020. The process had some hindrances, like the global pandemic and self-isolation, but nothing major. It was frustrating and challenging, took longer than expected, but as a whole, the process was interesting and also quite rewarding.

I am grateful for the support from my examiners Professor Jose Martinez Lastra and University instructor Luis Gonzalez Moctezuma. Especially the structure of this thesis, posed by Prof. Jose Lastra, was clear and easy to follow. I only had to worry about the research and writing.

I also want to thank Atostek Oy for giving me the possibility to write this thesis as a part of my daily work and Mitsubishi Logisnext Europe Oy for the support and ideas. Special thanks to Klaus Förger and Antti Anttonen for the guidance through the process.

Thanks to my loved ones for the support throughout my studies.

Nokia, 19.2.2021

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AGV	Automated guided vehicle
API	Application programming interface
E2E	End to end (testing)
ERP	Enterprise resource planning
GUI	Graphical user interface
LTP	Level test plan
MES	Manufacturing execution system
MTP	Master test plan
OTA	Over the air
ROI	Return of investment
SUT	Software under test
SAFe	Scaled Agile Framework
UI	User interface

.

1. INTRODUCTION

1.1 Background

During the last few decades, software has become a critical part of industrial systems. The use of software has grown rapidly in each part of the ISA-95 automation pyramid (Figure 1) model and the levels have become more dependent on each other. It is also expected that the use of software will become even more common as the industry seeks effectiveness from automation.

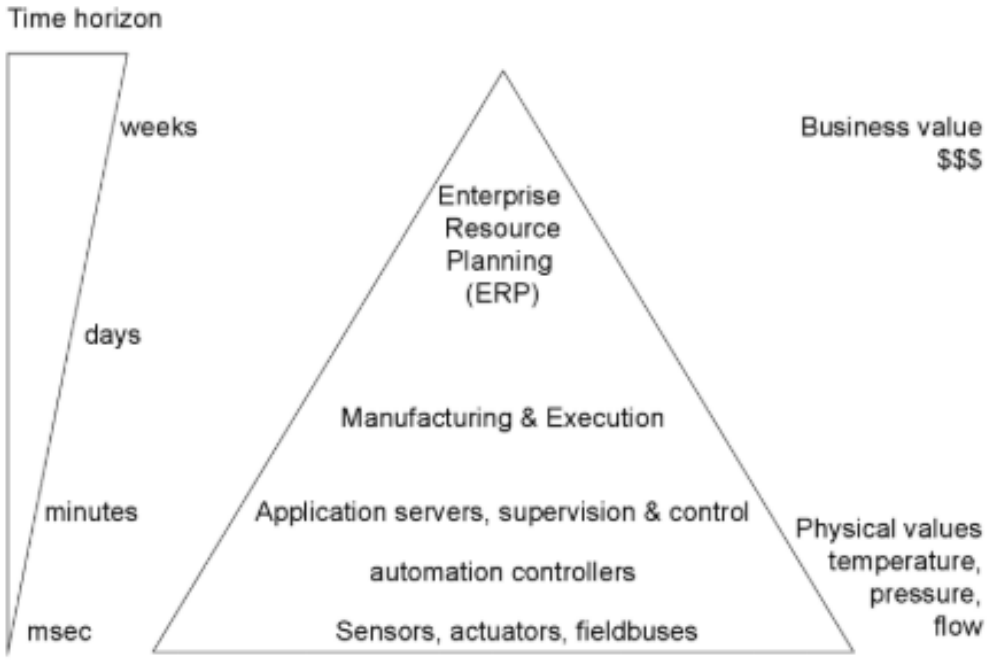

Figure 1. Automation pyramid *(Hollender 2010)*

At the same time, the complexity of the software has also increased. This provides unique challenges to maintain the integrity of the connected software in different levels of production systems. For example, the configurations made at the MES level have a direct impact on the devices at the field level. The nature of modern, directly connected systems means that even minor errors, such as misspellings or wrong datatypes at the high level can cause major issues at the field level.

This problem has created a growing need for testing and quality assurance. A 2011 study by Pierre Audoin Consultants found out that companies invest up to 50 billion

dollars in testing and quality assurance annually and it one of the fastest-growing areas in IT services (Pierre Audoin Consultants GmbH 2011). Also, a 2013 study by the University of Cambridge estimated that the total cost of debugging software accumulated to $312 billion per year and that failure to adopt proper debugging tools cost the economy $41 billion worth of programming time annually (University of Cambridge 2013).

Testing can be done manually or with automated tools. In manual testing, the tests are executed by humans according to some set of actions and expected results and by general visual observation of the interfaces. With complex and interconnected views, manual testing can require a lot of work and can lead to false-positive results, if the tester forgets or fails to identify deficiencies. Automated testing is done with tools such as scripting and testing frameworks tests a pre-written to perform the same set of actions and to expect the same set of results each time. This eliminates human errors, omission errors, and leaves more time for developers to focus on other tasks. Automated testing requires planning. Creating a testing plan requires defining the coverage, testing technologies, test cases, and testing methods. (Ammann and Offutt 2008)

1.2 Problem definition

Ideally, the testing practices are crafted at the beginning of a new software project. This way tests can be developed gradually along with the software, allowing developers to refine test cases and learn to prioritize the most critical areas to test, thus making the development of the software easier to maintain. However, this is always not possible, and testing might be neglected for different reasons e.g. when prioritizing feature development or when lacking a coherent testing plan. According to a survey study by Torkar and Mankefors, 60% of developers said that testing (verification and validation) was the first neglected thing when something had to be discarded due to timeline restrictions (Torkar and Mankefors-Christiernin 2003, 164-173).

The lack of automated testing becomes an especially heavy burden if the software is released as a stand-alone software with an offline installer, which is often the case for industrial systems. Providing software updates and bugfixes to an offline application can be difficult and usually requires installing a new version. Also, as the complexity of the software accumulates, so does the risk of creating a bug and not catching it before the release. This usually requires help from technical support and is very cost-intensive (Figure 2). The severity of a bug can also be difficult to measure. A bug can for example complicate or block the use of software, but it can also be cause physical damage to people or the environment.

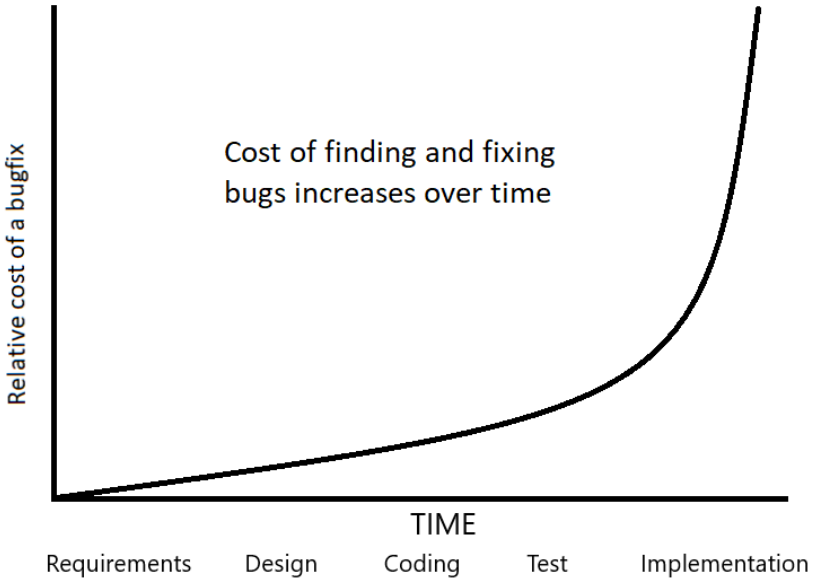

Figure 2. Cost of a bug *(S.M.K and Farooq 2010)*

This thesis focuses on creating an approach for adopting testing practices, efficiently developing tests, and maintaining the testing practices in a mature industrial software project.

1.3 Objectives

The objective of this thesis is to answer questions related to the development and implementation of a testing plan for software that is in a mature stage of development as well as considering the benefits of testing on the related systems in the software ecosystem. The research work aims to answer the following questions which also define the scope of this thesis:

- How the testing can be introduced to be a persistent and maintainable feature of development to a mature project?
- How to identify the most critical parts of the software that should be tested in industrial systems?
- What are the factors that affect the overall design and efficiency of the tests and what are the challenges in the industrial context?

1.4 Limitations

The scope of this thesis is to provide an approach for adopting testing practices in a software project. The technical details regarding test development are out of scope for this thesis.

The testing in the scope of this thesis is limited to *functional* testing. Usability-, scalability-, performance-, accessibility-, security- and other forms of *non-functional* testing are not discussed in this thesis.

1.5 Outline

The structure of this thesis is as follows; Chapter 1 introduces this document and established the problem and the objectives as well as the limitations of the thesis work. Chapter 2 defines the background and defines the current state of the art in software testing. Chapter 3 presents a proposed method for the problems discussed in chapter 1. Chapter 4 describes the implementation of the methods proposed in Chapter 3. In chapter 5 the results and conclusions are reviewed and some reflections and possible further work are considered.

2. STATE OF THE ART

In the current field of software development, testing is seen as an integral part of the development process. Testing is a large field that includes everything from unit-testing of a simple function to an acceptance testing of the whole software. It is a cost-intensive process, done either manually or automatically. The goal is to execute the software with the intent of finding defects, which is important because the goal steers our actions. A goal of not finding errors would lead towards testing with data that would likely not produce errors. The goal is to find the errors, but the main purpose of testing is not to find all existing defects but to provide value to the bottom line. Everything cannot be tested, so the approaches, methods, etc. should be selected to make a fitting testing plan for the program which is to be tested. If the testing is not demonstrated to be adding value, it will most likely be overlooked. (Myers and Sandler 2004)

The ten principles of software testing according to The Art of Software Testing (2nd edition) by Glenford Myers (Myers and Sandler 2004) are:

1. A necessary part of a test case is a definition of the expected output or result.
2. A programmer should avoid attempting to test his or her program.
3. A programming organization should not test its programs.
4. Thoroughly inspect the results of each test.
5. Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
6. Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
7. Avoid throwaway test cases unless the program is truly a throwaway program.
8. Do not plan a testing effort under the tacit assumption that no errors will be found.
9. The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
10. Testing is an extremely creative and intellectually challenging task.

This set of principles gives some general guidelines on how testing should be approached in any organization in an ideal environment. But the reality is often limited with time and resources that dictate the actual approach. For example, the number 2: "*A programmer should avoid attempting to test his or her program*", is often not feasible for the white-box type of automated tests.

2.1 Software testing approaches and methods

The testing approach considers how the program that is meant to be tested is viewed. There are many ways to categorize the field of testing approaches, but in general, the types of testing approaches can be divided into black-box testing and white-box testing. The testing method considers the way tests are performed: which is either manually or automatically. (Myers and Sandler 2004; Sawant, Bari, and Chawan 2012, 980-986)

2.1.1 Black-box testing

Black-box testing is a data-driven approach: the program to be tested is viewed as a *black box* and the tests only focus on the inputs and outputs without considering the structure or any internal functionalities of the program. (Figure 3)

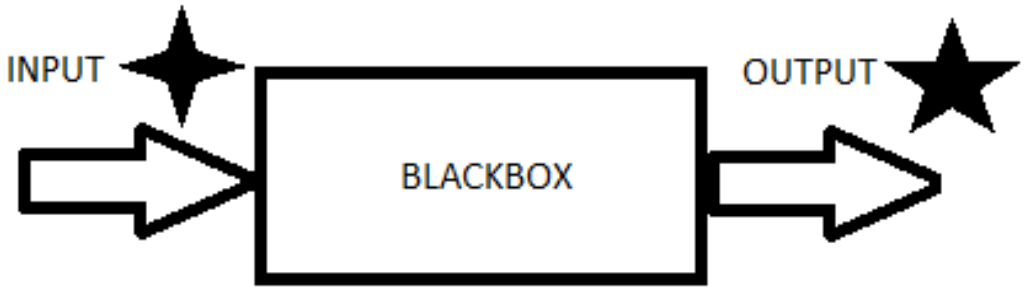

Figure 3. Black-box principle.

The purpose of this approach is to find if the inputs are accepted and result in outputs that meet the requirements. The advantages of this approach link with the principles of software testing: the writer of the test can be anyone and independent from the development of the code. This approach is also useful for larger entities of code and to code that relies on third-party software because the internal functions can become very complex or not be available at all. (Myers and Sandler 2004; Sawant, Bari, and Chawan 2012, 980-986)

The issue with black-box testing is that it creates a problem called *exhaustive input testing*: to ensure correct functionality of the program, not only valid inputs, but all possible inputs should be tested. To avert this issue, an *equivalence partitioning* methodology is used to define the test cases. The two considerations for a good test case should be:

1. It should reduce the number of other test cases that need to be developed to cover a test scenario.
2. The selected inputs should cover a wide set of test cases. Both the presence and the absence of errors in the outputs should be examined. (Myers and Sandler 2004)

2.1.2 White-box testing

White-box testing is a logic-driven approach: it involves analyzing how the system processes the input to create an output. This is beneficial because the code can be also validated to meet design patterns and possible bugs can be found before they cause any issues. (Sawant, Bari, and Chawan 2012, 980-986)

White-box testing can be static or structured. The static testing is done with code inspections and walkthroughs, in which the programmers or lint-tools analyze the code for errors. Structural tests, like coverage testing, path testing, or flow testing aim to identify errors in the decision logic in the code. (Nidhra and Dondeti 2012, 29-50; Myers and Sandler 2004)

2.1.3 Manual testing

In manual testing, the execution and validation of tests and results are done by a human. The field of manual testing is wide and the main methods could be categorized into formal and informal methods. In a study by Ramler and Wolfmaier, manual testing methods were found to be effective in situations where the software is tested for novel errors (Ramler and Wolfmaier 2006, 85–91). These scenarios could be for example:

- the project is still in the early stages of development when the code mutates a lot and maintaining automated tests would be difficult.
- Complex testing scenarios like acceptance tests.

Manual testing exists in many different forms. Some manual testing methods are explained in The Art of Software Testing (Myers and Sandler 2004), which describes methods like code inspections, walkthroughs, desk checking, and peer ratings. But the field of manual testing is much wider than this and often the methods depend on the

testers and local practices. A special case of manual testing is *pull requests*, which are an important peer review-based manual testing method in the pull-based development model.

Inspections and walkthroughs usually revolve around a team of three to five people and follow a set of error-detection techniques and guidelines for a group code reviewing session. In both methods, the team members are given roles (moderator, programmer, tester, senior/junior developer, etc.) and the session is structured to revolve around these roles. In inspections, the code is reviewed against a checklist. Walkthroughs are more focused on the use cases of the software and the cases are walked through the logic of the code. Both methods are effective tools to find errors but are quite time-consuming as they require a dedicated session of several people and reading the material in advance before the session. (Myers and Sandler 2004)

Desk checking is not as structured as the inspections and walkthroughs and can be done ad-hoc. Desk checking is usually performed by the programmer or by his/her peer. This method is can be less productive because of the lack of discipline and especially if the programmer is testing his/her software, which is against the principles of testing. Peer reviews, where the programmer self-evaluates his/her work and randomized peers review can provide indirect help for testing. The peer-reviewing process does not help with the testing itself but allows programmers to self-assess their skills. (Myers and Sandler 2004)

A study by Itkonen, Mäntylä, and Lassenius describes how testers approach manual testing tasks. They identified that most of the different practices the testers used to manage the testing session management and testing execution were exploratory and relied largely on experience and tacit knowledge. Most of the strategies to manage the session (the overall structure of testing work) were exploratory: e.g. browsing through the UI trough, but testing an individual feature based on subjective evaluation or identifying weak areas based on tacit knowledge about potentially weak points of the software. The documentation-based session management relied on e.g. checklists and pre-set lists of tests to perform. Most of the test execution strategies (testing an individual feature) were also exploratory, where the testing was based on hypotheses and assumptions on how the feature might cause errors. The other execution strategies were comparison (e.g. comparing the performance of similar functions) and input (e.g. testing boundaries). (J. Itkonen, M. V. Mantyla, and C. Lassenius 2009, 494-497)

Pull requests are a widely used mechanism in the pull-based software development model to make changes to the codebase. Pull requests combine manual testing methods from the desk checking and peer-reviewing process: the pull request, which can be e.g. a new feature to the software, is first sent to be reviewed by other developers. The reviewers can run the code, make remarks, or update the code. Eventually, the code is either rejected or accepted to be merged to the main software (Figure 4). The pull request process is therefore also an iterative desk checking and peer-reviewing testing method.

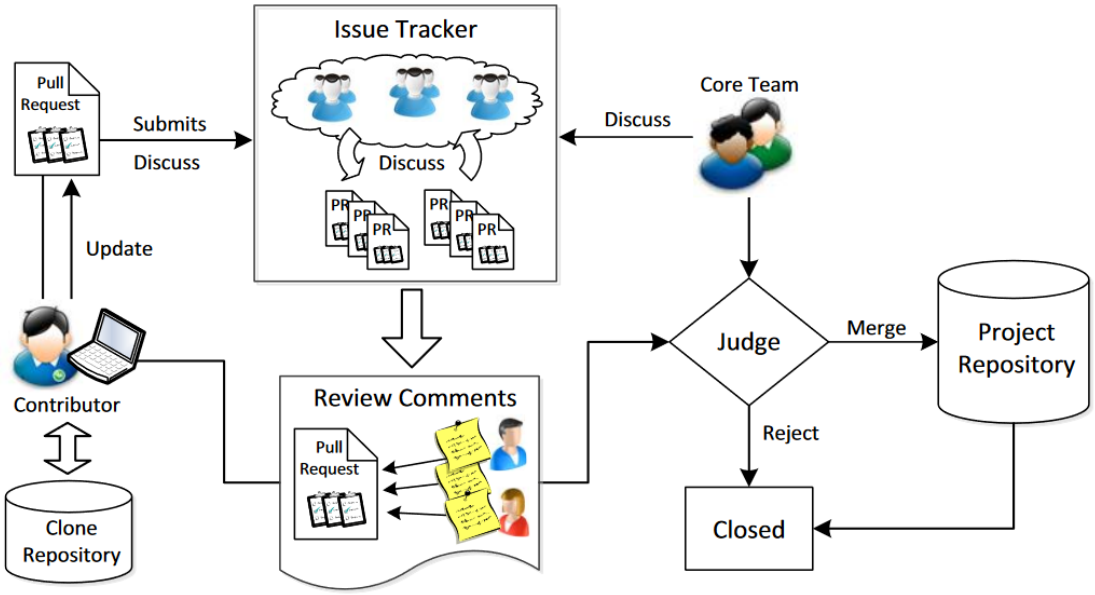

Figure 4. Pull request mechanism. *(Y. Yu et al. 2014, 609-612)*

The problem with manual testing methods is that these methods can leave systematic errors even tests are done with discipline because human error is always present. The second problem is the scale: as the software grows, the number of tests accumulates and becomes unpractical to do manually. (Sahaf et al. 2014, 149–158)

2.1.4 Automated testing

In automated testing, once the test has been set up, the test execution and result validation are done by testing software (script runner, testing framework, etc.) without any human intervention. Automated tests are particularly useful in repetitive testing cases, like unit tests or in scenarios that are difficult to execute manually, which could be for example a time-critical test scenario. Fully automated testing is usually not a desirable scenario: 94% of developers do not agree that automated testing can fully replace manual testing. (Varma 2000; Dudekula Mohammad Rafi et al. 2012, 36-42)

In contrast to manual testing, automated tests require a lot of work setting up but running the tests requires little or no action. Automated tests are run with script runners or testing frameworks and can be set up to run periodically or related to any activity. This is useful as the software can be tested often with little or no effort. The tools required to run the scripts and frameworks depend on the software that is being tested: while some of the more common programming languages have readily available tools, support, and documentation often for free, proprietary languages may require very specific testing software and training. This is also explained in the survey study by Ng et al. where the main hindrances in adopting automated testing tools were found to be the monetary cost of use and time consumption (Ng et al. 2004, 116).

Automated testing methods can be generalized into two types: end-to-end testing (E2E) and API/component testing. In end-to-end testing, the execution and validation are done by simulating the use of the program GUI usually without considering at all what is happening in the internal system. It can be described as a black-box type of testing. The point of end-to-end testing is to ensure that the end-users' point of view functions correctly. These tests are usually created with testing frameworks that must be able to interpret and interact with the GUI. API testing methods use directly the internal modules, classes, and functions of the program and can be categorized as white-box testing. These types of tests are usually written in the same language as the main program. The tests are run by a script runner that passes pre-set or random values to the test, which inputs the values to a module/class/function and expects a certain value as an output.

Different types of tests offer a different kind of coverage. This has some implications when the test efficiency (most coverage with minimal effort) is critical. Vice president of Cypress (developer of testing tools) Gleb Bahmutov implied that for GUI applications, system-level end-to-end testing is more efficient than e.g. unit testing in terms of offering testing coverage. He argued that with end-to-end testing, not only the application logic (e.g. CRUD operations) are covered, but also the whole application and its components need to be rendered, validating the integrity of the GUI as well. The interaction with the full application makes the end-to-end type of tests highly efficient. (Gleb Bahmutov 2020)

The benefits and limitations of automated software testing are described in an article by Dudekula Mohammad Rafi et al. (Dudekula Mohammad Rafi et al. 2012, 36-42). The study collected empirical findings and experiences of testing automation from literature and surveyed how the found benefits and limitations of testing automation were seen in the industry. Survey also pointed out that the overall satisfaction with the automated

test was high: 84% of participants were satisfied or highly satisfied with automated testing. The most notable benefits of automated testing that were agreed upon in the survey were:

1. Test reusability makes automated testing productive.
2. Repeatability of tests, which allows running more tests in less time
3. Better test coverage improves product quality.
4. Automated testing saves time and cost as it can be re-run without extra effort.
5. Automated tests improve the ability to meet deadlines and provide more confidence in the product.
6. Correct testing tools can reduce the effort needed by the developers.
7. Complete automation reduces the cost overall cost and facilitates continuous test development.
8. Automated tests reduce the amount of effort but are not guaranteed to find more complex bugs. (Dudekula Mohammad Rafi et al. 2012, 36-42)

The most agreed limitations according to the survey were:

1. Compared to manual testing, the cost is higher, especially in the beginning.
2. The design and maintenance of the tests require extra effort.
3. Test developers should be skilled enough to build automated tests.
4. Automated testing requires a high investment in tools and training.
5. Automated tests require more effort from developers but leave complex bugs untested.
6. Testing tools can be incompatible and do not provide the needed functionalities.
7. Automated tests do not replace manual testing needs (Dudekula Mohammad Rafi et al. 2012, 36-42)

The article by Taipale et al. also points out the observations that facilitate and hinder the use of automated tests at an organizational level. The facilitating factors were generic or similar products, low need for human involvement, standardized technology, and internal customers. Hindering factors were the opposite of facilitating factors: customized or complex products, high need for human involvement, rapid changes in technology, and external customers. (Taipale et al. 2011, 114-125)

2.1.5 Software testing types

Different kinds of tests can be categorized into testing types. The testing type spectrum is wide and some implementations fit both testing approaches. Table 1 presents a general categorization of the testing type spectrum.

Table 1. Testing type spectrum. (Nidhra and Dondeti 2012, 29-50)

Testing type	Opacity	Specification	Scope	Method
Unit	White-box	Low-level code structure	Small units of code	Automated
Integration	White-box and black-box	Low- and high-level design	Multiple classes or	Both
System	Black-box	Requirements	For the entire program in the representative environment	Both
Acceptance	Black-box	High-level design	For the entire project in the customer environment	Usually manual
Regression	Black-box and white-box	High-level design	Any of the above	Usually automated

Unit testing is a type of testing where individual components of the software are tested. It is the smallest testable part of the software (e.g. a single UI element or function with few outputs and single output). (Ammann and Offutt 2008)

Integration testing is a type of testing where the single units of code are combined as a group and the expected functionality of the group of individual components is tested. This can be for example testing the correct functionality between two UI components or UI components with backend services. (Ammann and Offutt 2008)

System testing is where the whole software is tested against the set specifications. The assumption is, that the individual components are operating as intended and the software is tested as a single unit. The goal is usually to find design or specification issues, and the tests should be performed by testers, not programmers. (Ammann and Offutt 2008)

Acceptance testing is supposed to determine if the program meets the set functional and business requirements. The testers should have a strong knowledge of the set requirements and domain of the product. Acceptance testing must be done with the end customer, as the goal of the testing is to validate that it meets the customer's needs. (Ammann and Offutt 2008)

Regression testing is a type of testing where typically the whole system or large components of it are tested to validate that each change made, even the small bug fixes, etc., did not break something elsewhere in the system. Regression test cases can consist of a large set of different types of tests (unit, system, integration, etc.). These test sets are often large and can be time-consuming even when automated. Therefore, regression test sets should be automated and run periodically, preferably at night when development is halted. Maintaining regression tests can be difficult. (Ammann and Offutt 2008; IEEE 2008, 1-150)

The tests can also be divided into deterministic and non-deterministic tests (Figure 5). The deterministic test has a very narrow scope, and each test run will always yield the same set of outputs for the same set of inputs. Nondeterministic tests are the opposite of this and can produce different outputs for the same set of inputs. (Ammann and Offutt 2008)

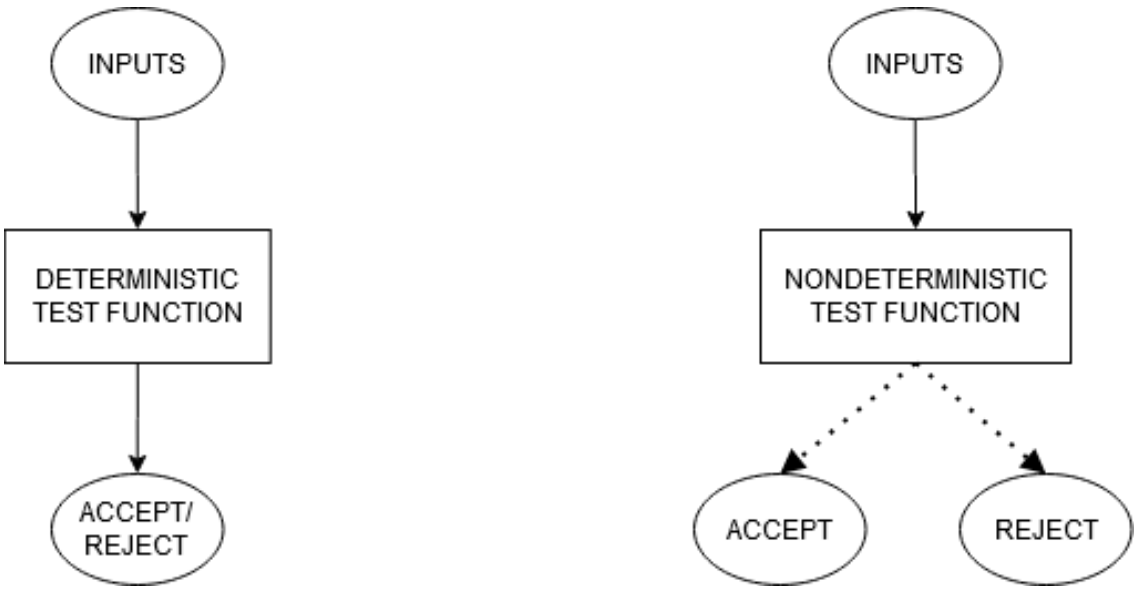

Figure 5. Deterministic and non-deterministic test.

An empirical study by Luo et al. points out, that non-deterministic tests should be avoided as these tests create different results intermittently, which causes confusion and exhaustion among developers. The study reveals that the main causes of non-determinism are asynchronous functions, concurrency, and test order dependency. (Luo et al. 2014, 643–653)

2.1.6 Testing plan

A testing plan is essentially a description of why the testing is needed, what and when is required to be tested, and how the tests are conducted (Ammann and Offutt 2008). The testing plan is described in ANSI/IEEE standard 829-2008 in two ways (IEEE 2008, 1-150):

- "A document describing the scope, approach, resources, and schedule of intended test activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning"
- "A document that describes the technical and management approach to be followed for testing a system or component. Typical contents identify the items to be tested, tasks to be performed, responsibilities, schedules, and required resources for the testing activity. The document may be a Master Test Plan or a Level Test Plan."

Master test plan (MTP) should provide a general test plan and test management rules and requirements across the project or multiple projects. The role of MTP according to ANSI/IEEE is to set the objectives for each part; manage time, resources, and interrelations between parts; identify risks, assumptions, and workmanship of parts; define the controls for test effort; to confirm the objectives set by quality assurance plan. It should also define the integrity, the levels to test, tasks that need to be performed, and the documentation requirements. (IEEE 2008, 1-150)

A level test plan (LTP) specifies the scope, approach, resources, and schedules for each level of testing. Each level defined in MTP should be specified with an LTP named after the test. For example, if MTP defines unit, integration, and acceptance testing levels, there should be documents named "Unit test plan", "Integration test plan" and "Acceptance test plan". The differentiation to individual plans derives from the fact that the different levels of testing have a different set of requirements, methods, resources, and tools. (IEEE 2008, 1-150)

The ANSI/IEEE 829-2008 provides outlines for the MTP and LTP. The outline provides only a template for the document and needs to be modified to meet the needs of an individual project. The document is usually in written form, but the format may depend on the organizational/project needs (ISO/IEC/IEEE 2013, 1-138).

Testing plans can be sometimes mixed to *testing strategy*, which according to Ammann and Offutt, lacks a proper definition in testing literature (Ammann and Offutt 2008). Although in the ISO/IEC/IEEE standard 29119-3-2013 test strategy is defined as: "*Part of the Test Plan that described the approach to testing for a specific test project or test sub-process or sub-processes*". This is a very wide description and can be used to describe any of the following: the test practices used; the test sub-processes to be implemented; the retesting and regression testing to be employed; the test design techniques and corresponding test completion criteria to be used; test data; test environment and testing tool requirements; and expectations for test deliverables (ISO/IEC/IEEE 2013, 1-138). This is why the use of the term is avoided in this thesis.

2.2 Economics and management of software testing

The economics of test automation is perhaps one of the most important things to emphasize in a project that is lacking sufficient (automated) testing. A study by Taipale et al. points We found that the main disadvantages of test automation are the costs, which include implementation, maintenance, and training costs (Taipale et al. 2011, 114-125). As described in the introduction, the economics of testing are closely linked to the support from management, which is the most important factor to succeed in the automatization process (Graham and Fewster 2012). If the testing is not seen to provide value, it can easily be neglected in favor of feature development. Therefore, perhaps the best generic way to justify the investment in automated testing is to demonstrate the return of investment (ROI) that automated tests will provide. This can be achieved by tracking the right metrics and by using the approximation models for calculating the return of investment of different scenarios.

Graham and Fewster have collected experiences of test automation from 28 different cases. Nine out of these cases reported collecting some metrics of ROI and almost all results were positive. These findings are also supported by the multivocal literature review by Garousi and Mäntylä, where the reported ROI for automated testing was between 40-3200%. Metrics used in the Graham and Fewster case studies were in general:

- Time consumed

- Quantifiable savings
- Satisfied customers

The time consumption was mentioned in at least five cases. The reported benefits were accelerated testing, reduced development cycles, reduced time spent on testing related activities, and reduced the number of required testers. Several cases reported quantifiable savings, as costs per release, costs per test, and comparison between calculated costs before (manual testing) and after (automated testing). Some cases also reported that customer satisfaction increased significantly when quality improved and the development was faster. (Graham and Fewster 2012; Garousi and Mäntylä 2016, 92-117)

The case study by Sahaf et al. assesses the ROI of different testing setups (fully manual, fully automatic, and combinations of each). In the study, a system dynamics model (SD model) of software testing was created. The model was tested using simulation for each different testing setup. The SD model considered several important parameters, like the number of new testing cases, testing cycle time, the productivity of designing/scripting/execution/reporting/updating, fail-/pass rating, correcting, maintaining, and the number of employees involved.

The simulation results of the study conclude that manual testing has a short set-up period and in short term, it is more productive than automated testing (Figure 6, scenario 1). But after a setup period of automated testing, automated testing will provide better results than manual only testing (Figure 6, scenario 2). The study also pointed out, that the results will be improved by introducing better tools, which make reporting and evaluation more effective (Figure 6, scenario 3) or adding more manpower, which will reduce the setup time (Figure 6, scenario 4). (Sahaf et al. 2014, 149–158)

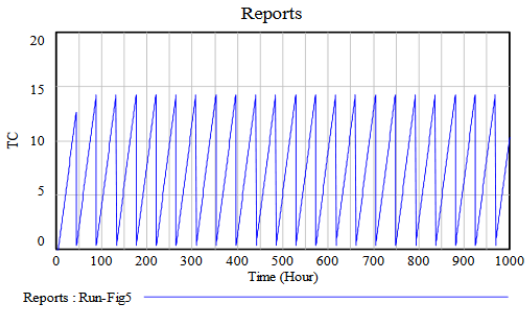

Figure 5. Scenario 1

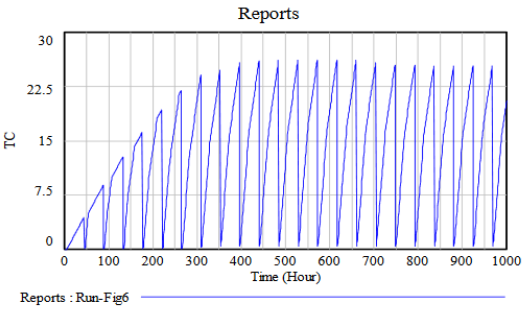

Figure 6. Scenario 2

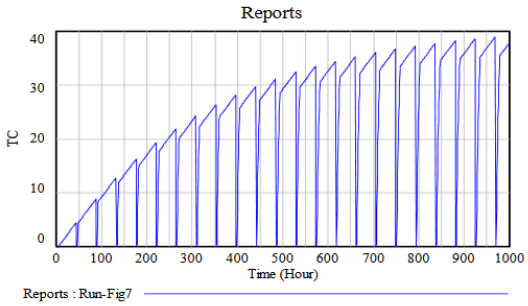

Figure 7. Scenario 3

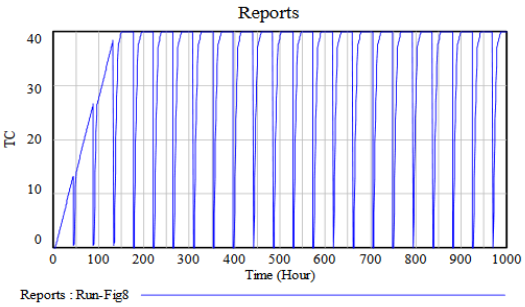

Figure 8. Scenario 4

Figure 6. Amount of testing cases per hours worked: manual (upper left), automated (upper right), automated with tools (lower left), and automated with more manpower (lower right). *(Sahaf et al. 2014, 149–158)*

Although the study points out that the specific results cannot be directly generalized, the simulation results demonstrate that in general, automating the tests will be more productive in the long term and that the results will depend on many outlining factors of the project. This finding is also supported in an article by Kumar and Mishra, where they analyze the economic side of automated testing from cost, quality, and time to market perspectives. In their study, they analyzed three different software concerning the three perspectives. The results of the study indicated that cost and time perspectives were improved in every software and the quality (in terms of the number of failures found) was improved in most of the cases (Kumar and Mishra 2016, 8-15). The findings regarding timing and cost are significant, but the improved quality can be debated. The article only specifies the quality by the number of failures with automatic versus manual testing. This is listed as a common pitfall of automated testing in the article “Establishment of automated regression testing at ABB: industrial experience report on 'avoiding the pitfalls'” (C. Persson and N. Yilmazturk 2004, 112-121).

The economics of automated versus manual testing are discussed from a different perspective in the article by Ramler and Wolfmaier (Ramler and Wolfmaier 2006, 85–91). The article describes a test automation opportunity cost model as well as considering

other influencing cost factors regarding software testing. They argue against the simplistic model of “universal formula” (Figure 7), which acts as a strong argument for test automation.

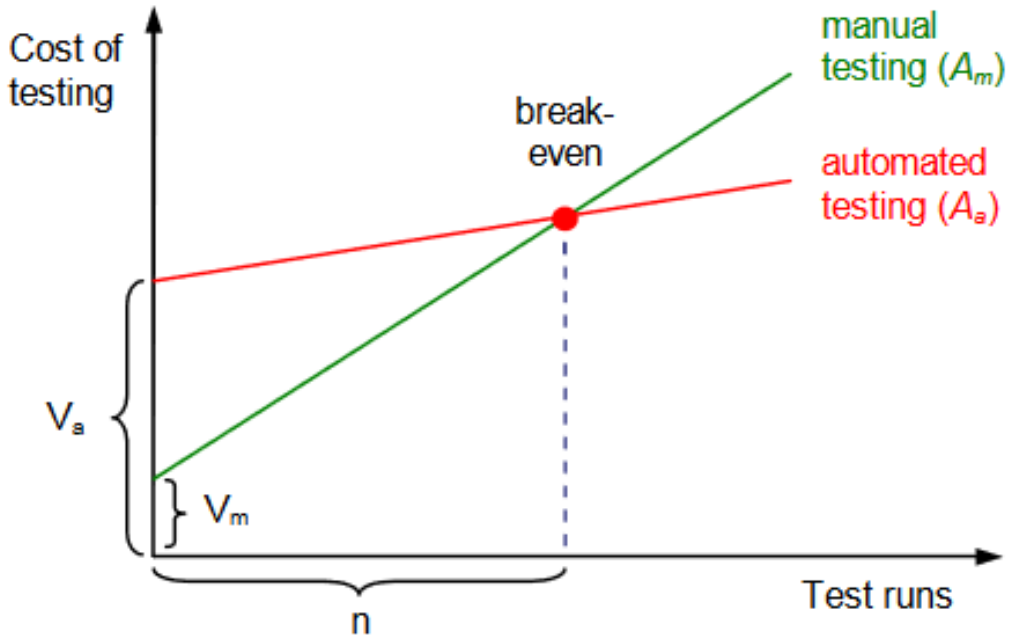

Figure 7. Break-even point for automated testing in a simplistic model. *(Ramler and Wolfmaier 2006, 85–91)*

Their criticism against this model is that it only calculates the costs without considering the different kinds of benefits of each approach. They also point out, that the manual and automated methods are incomparable: the output of the tests are different, and the real value of test runs are not equal; manual testing can lead to finding new defects which is valuable. They also criticize that the project context or budget is not considered in the simplistic model as well as pointing out that the simplistic model is missing additional cost factors, like tools and training costs.

Their alternative model aims to provide balance to the “production possibilities frontier” in software testing, a trade-off between higher upfront costs of automating tests and the opportunity cost of losing time to do manual testing. The proposed model suggests determining the benefits of each test case based on the estimated mitigation of risk it provides, so the most critical parts should be emphasized. In their model, the benefits of manual test cases and automated test cases are different, so those are calculated separately. The model formula (Figure 8) for finding the optimized number of tests takes to account the budget restrictions. The model can provide support and alternative quick sketched scenarios for a testing plan. (Ramler and Wolfmaier 2006, 85–91)

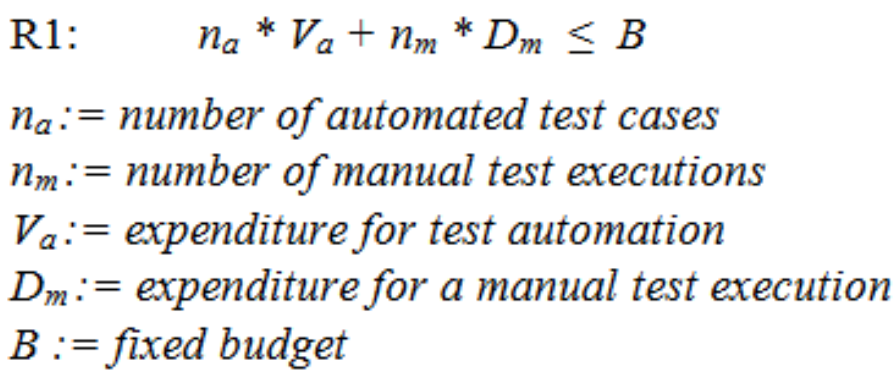

Figure 8. Optimizing balance between automated and manual testing. *(Ramler and Wolfmaier 2006, 85–91)*

2.3 Software testing implementation

Setting objectives and planning is crucial. As described in chapter 2.1.2, the different testing types and methods offer a different kind of testing coverage. Regression tests are good at testing the integrity of existing features but do not offer much help in finding new bugs or covering the edge cases. With manual testing methods, the opposite is often true. Therefore, it is important to carefully scope the testing objectives and form a testing plan with the development and management team. The most important questions regarding the automatization are: “what tests should (and should not be) automated?”, “when a test should be automated?” and “how it should be automated?”. Although there are no definitive answers to any of these questions, the testing literature does good general practices to consider.

2.3.1 Testing methods and approach

The level and timing of test automatization will always depend on the environment, project management, testing objectives, etc. A multivocal literature review by Garousi and Mäntylä and the case studies by Graham and Fewster provide answers to these questions.

The literature review by Garousi and Mäntylä split the question of what and when into five different categories (Figure 9): SUT-related–, test-related–, test-tool-related –, human and organizational– and cross-cutting factors. (Garousi and Mäntylä 2016, 92-117)

- SUT-related factors: the system should be mature and stable enough before implementing the automated testing. Automating tests for features under the early

development phase or re-implementation will cause broken tests (false negatives and false positives). Test automatization is also not recommended for a product with a short life cycle.

- Test-related factors: the focus on test automation should be on the tests that need repetition (e.g. regression testing) are deemed critical for the SUT or that are hard to perform manually (e.g. performance testing).
- Test tool-related factors: the selected tool should be carefully researched for combability for the SUT. Progressing with test automatization should be halted without suitable tools for the task.
- Human and organizational factors: maturity of the organization (being able to adapt to change), resource allocation (prepared to invest more time upfront), the current skill set of the team (testers should have programming skills and/or developers should have testing skills) are the most important factors when automatization is being considered. Management is as important: being able to communicate the benefits and handle the change resistance is crucial.
- Cross-cutting and other factors: the economical aspect, i.e. cost-benefit analysis of automatization needs to be considered (more in chapter 2.2). Also, the automatability of the SUT (how easy the automatization would be?) and the development model should be taken into account before the decision to automate testing.

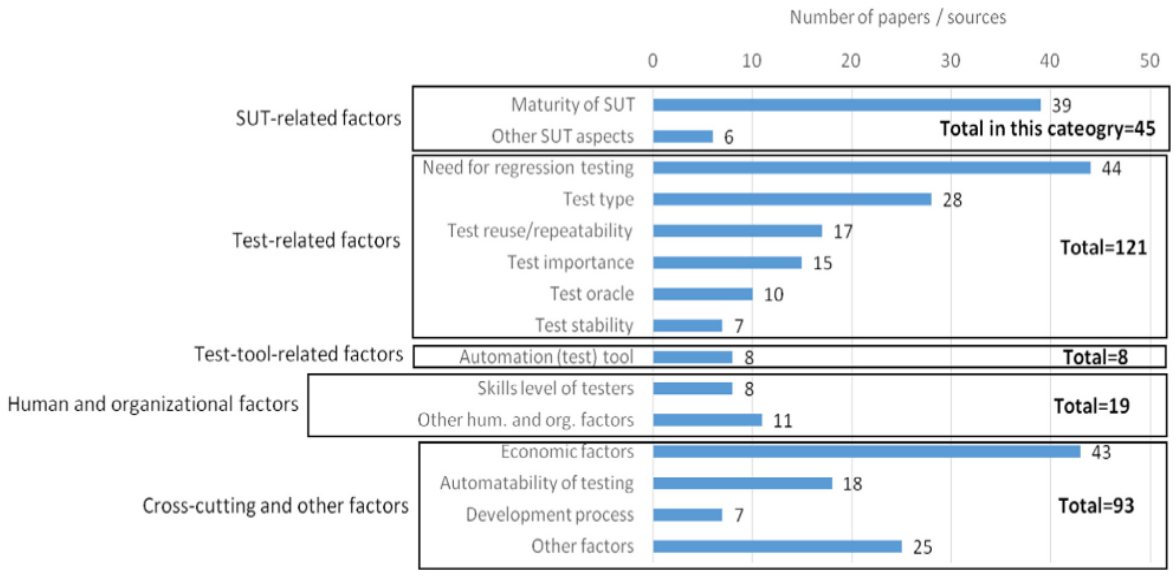

Figure 9. Factors affecting what to-and when to automate questions. *(Garousi and Mäntylä 2016, 92-117)*

The case studies by Graham and Fewster correspond and offer some additional points to the findings by Garousi and Mäntylä. One of the main things these cases point out is that the economical aspect of decision-making needs to be brought to the level of individual tests. There is no point in automating tests that do not provide value and further, only the tests that need to be run frequently, should be automated (Graham and Fewster 2012). This can be refined in terms of which types of tests are considered useful to be automated: regression testing is mentioned in many cases as the most important part of testing (Graham and Fewster 2012). This corresponds with the test-related factors above. The study by Rafi et al. also points out, that automation is the superior choice especially when several regression testing rounds are needed (Dudekula Mohammad Rafi et al. 2012, 36-42).

The quality and maturity of software are mentioned in one case: if the design and implementation of the SUT are unstable or not well written, the automated tests will cause chaos and high maintenance costs (Garousi and Mäntylä 2016, 92-117). This corresponds with the SUT-related factors above.

Graham and Fewster emphasize that standardization of testing related issues is critical when test automation is developed. The way tests are developed, naming conventions, documenting the tests, etc. needs to be planned with the people involved with the testing. (Graham and Fewster 2012)

The case studies by Graham and Fewster also point out, that testing and test development should be treated as a separate skill. This does not mean that the tasks require different individuals, but different roles for the same individual. But the development of automated tests usually requires programming skills, depending on the level of abstraction the testing tools provide. (Graham and Fewster 2012)

The importance of planning and goals are also mentioned in the case studies by Graham and Fewster. They point out that the field of test development is never problem-free, and the role of the testing plan should rather be a guideline that adapts to the current situation than something permanent. The testing goals were also found out to be helpful. A good goal has realistic expectations and a tangible schedule. Early results were seen to promote success. (Graham and Fewster 2012)

2.3.2 Test prioritization

Prioritization is an important aspect of test development. As described in chapter 2.2, the test development and testing always involve working with finite resources, that need to be used as efficiently as possible. The prioritization problem can be divided

into two categories: what tests should be developed first, and which tests should be run most frequently, once the number of tests grows?

A case study article by Srikanth and Banerjee describes a PORT (prioritization or requirements for a test) model for testing prioritization. The model consists of four factors, which were seen to improve the failure detection rate and business value: customer assigner priority, developer-perceived implementation complexity, fault proneness, and requirements volatility.

- Customer assigned priority.

  Focusing on the factors that hinder the everyday use of the software by the end-user has been shown to improve customer perceived value and satisfaction. They argue that there is a lot of potential to increase the business value by prioritizing testing for the features with the highest priority for the customer. (Srikanth and Banerjee 2012, 1176-1187)

- Developer-perceived implementation complexity.

  There is evidence that the complexity of the code is related to the number of errors in the code. Therefore, the developers should have the authority to prioritize the testing of some features over others based on the perceived complexity of the code. (Srikanth and Banerjee 2012, 1176-1187)

- Fault proneness.

  Srikanth and Banerjee argue that the testing priority should be assigned based on the history of errors for the feature in question (Srikanth and Banerjee 2012, 1176-1187). The study by Ostrand and Weyuker also verifies this: their study indicated that for late releases of the software, 7% of the files contained 100% (Figure 10) of the faults (Ostrand and Weyuker 2002, 55–64). Later findings of Ostrand, Weyuker, and Bell from two case studies concluded that 20% of the files that were identified to be the most problematic contained 83% of the errors in the software (Ostrand, Weyuker, and Bell 2004, 86–96).

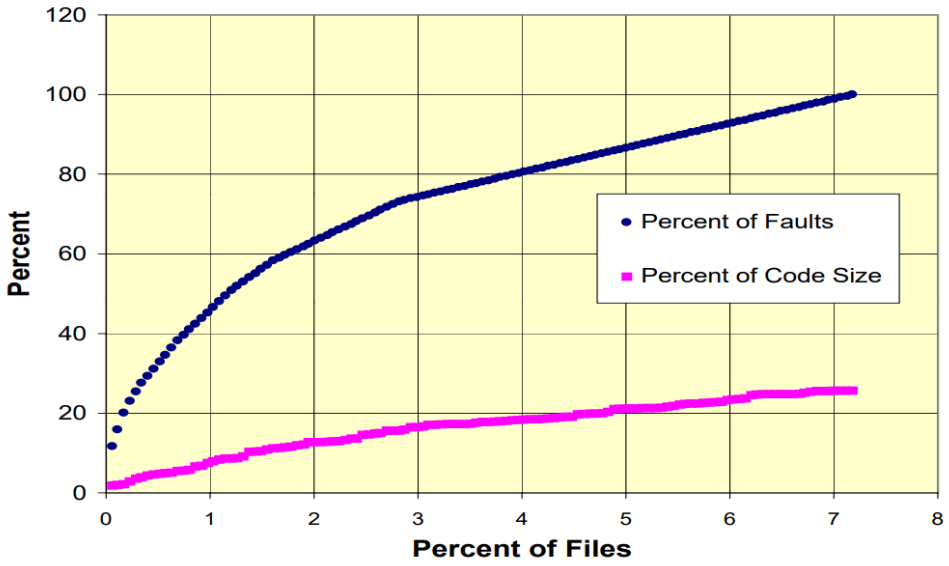

Figure 10. Distribution of faults in software *(Ostrand and Weyuker 2002, 55–64).*

- Requirements volatility.

  Several studies show that the high requirement volatility or incomplete requirements can be a significant cause for failures in the software. Therefore, the features that have been subject to changing requirements, should be prioritized in testing. (Srikanth and Banerjee 2012, 1176-1187)

2.4 Maintaining software testing and result analysis

The literature review by Garousi and Mäntylä offers some guidelines for the maintenance of automated tests. They analyze the questions in SUT-related–, test-related– and test-tool related factors (Garousi and Mäntylä 2016, 92-117):

- SUT-related factors: the complexity, customization, and 3rd party dependencies affect the automatization effort. The tests that need to be run in customized environments or are dependent on external software are harder to maintain and therefore the automatization should be avoided.
- Test related factors: the tests that produce unpredictable outcomes or that fail often should not be automated. The results can be hard for a human to analyze and hard to maintain.
- Test tool-related factors: before selecting a tool, the tool-related risks should be mapped. Open-source tools are often a good option compared to paid options, as the cost of using open-source tools is usually low. But the lifecycle of the tool, suitability, and features should also be considered.

The case studies by Graham and Fewster correspond and offer some additional points to the findings by Garousi and Mäntylä. The case studies emphasize the importance of good result reporting practices: a good result should tell what was expected and what was the outcome - a true/false result of a test can leave room for *zombie-tests* that do not explain *why* the test has passed or failed (caused by non-determinism, chapter 2.1.5). Some cases also reported positive outcomes on making the testing reports visible for management as well: this links to the economic aspects of testing (chapter 2.2). Keeping the results visible demonstrates the usefulness of testing by itself. The reporting should be tailored to the people viewing the results: a less technical report is better for the management level and a detailed version for the developers. The case studies point out, that it is important to invest the time to create understandable and consistent result reports. This helps with the maintenance of the tests and decreases the time needed for briefing new developers. Graham and Fewster emphasize the continuous development of testing practices. The tools used, development style, documentation, reporting, etc. are important to re-evaluated occasionally. (Graham and Fewster 2012)

2.5 The current state of software testing in safety-critical industrial systems

Industrial software in general can be described as a system of systems. These systems often consist of multiple independent but connected systems that need to collaborate, creating dependencies between the software (Crnkovic 2008, 57–60; Boehm 2006, 12–29; Carlshamre et al. 2001, 84-91). Considering the interdependencies between software requirements, an industrial survey by Carlshamre et al. concluded that the was majority (75%) of interdependencies was caused by 20% of requirements and only 20% of requirements did not have interdependencies. (Carlshamre et al. 2001, 84-91)

The dependencies among the safety-criticality requirements caused by high-risk environments are the two main things that define industrial software development. This also affects the testing of industrial software.

The study by Taipale et al. explored the current state of testing in five different companies in different industries and sizes. Company A, -C, and -D in the study were large international organizations that were operating with MES systems (Company A), process automation (Company C), and electronics manufacturing (Company D). The state of testing in Company A was focused on integration and system testing. The products were mainly tested manually because the customers required customized products that were hard to simulate and test automatically. Internal products, like development tools, were partly being tested automatically. Company A reported that the development of

automatization was part of their testing plan, but lack of time and resources was a hindrance. Company C also reported problems with automatization. Automated testing was seen to be difficult to develop and maintain for their products and manual testing was the primary method in their testing plan, which involved mostly system testing and end-to-end testing. The testing plan at Company D emphasized automated testing mostly at the system testing level. Automated testing was seen to improve the quality of products but also caused some problems because the tests required frequent maintenance and a comprehensive test set was hard to create due to lack of specifications. (Taipale et al. 2011, 114-125)

An industrial survey by Causevic et al. mapped contemporary aspects of testing among software developers and testers. The goal of this survey was to find differences in testing practices among five different categories: development process (agile vs. other), distribution of development, safety-criticality, amount of testing performed, and product domain. The main findings of this study were:

- The domain of the product (e.g. web vs. embedded/industrial software) did not affect the available time for testing much. The types of tests emphasized varied among the domains: web development emphasizing unit testing, embedded development emphasizing system testing.
- The developers using agile methods were unhappier with the testing practices and the non-agile developers were unknowingly using agile testing practices.
- The developers creating safety-critical software were more resistant to interact with the customer during the development of the software.
- Most of the organizations reported to have defined types of testing: unit, integration, regression, and system testing were the most preferred types of testing. The responsibility of developers was usually the white-box testing and the responsibility of the tester was the black-box testing.
- In general, open-source tools were used for unit testing and proprietary tools were used for higher-level testing. Some respondents reported using in-house developed tools. (A. Causevic, D. Sundmark, and S. Punnekkat 2010, 393-401)

One important finding to highlight from Causevic et al. is the fact that automatization was not widely adopted according to the open questions in the survey. This is also supported by a survey study by Lee et al. where they found out that 74% of participants reported automatization level below 50%. The test execution, test reporting, and defect management were the only parts of testing that were automated to even some extent.

(Figure 11) (J. Lee, S. Kang, and D. Lee 2012, 275-282; A. Causevic, D. Sundmark, and S. Punnekkat 2010, 393-401)

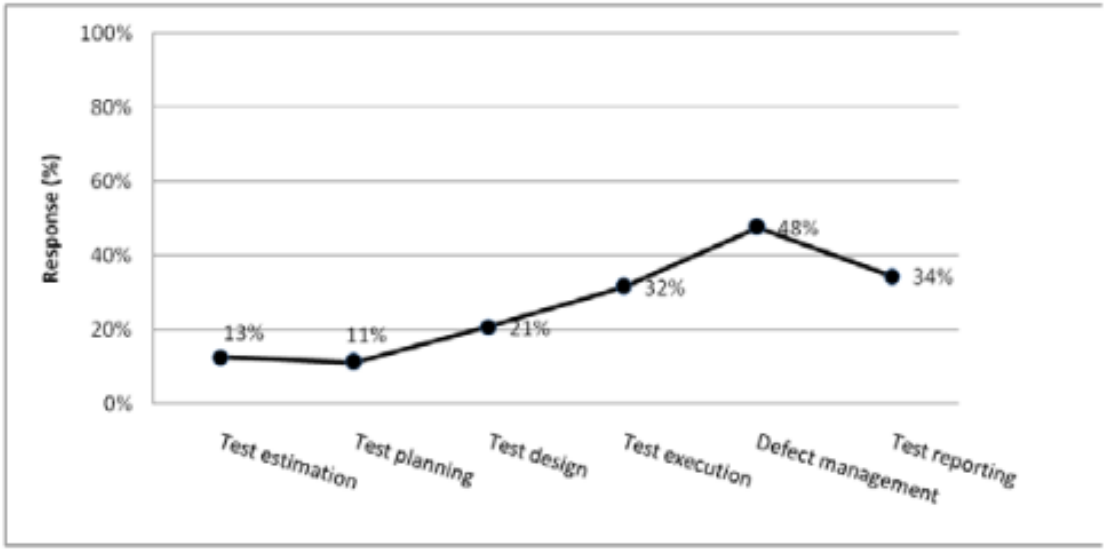

Figure 11. Level of test automation. *(J. Lee, S. Kang, and D. Lee 2012, 275-282)*

Another industrial survey by Mohammed Kassab compared testing practices between safety-critical and non-safety-critical systems. The key findings from this study were:

- Compared to non-safety-critical systems, testing is regarded more as a distinct phase in development.
- Testing practices in safety-critical systems were more applicable in the later stages of development.
- The effectiveness of testing is seen to be higher in safety-critical systems. (Kassab 2018, 359-367)

The survey also verifies the findings that system, integration, unit, acceptance, and regression testing are widely used types of testing in the industry (Figure 12).

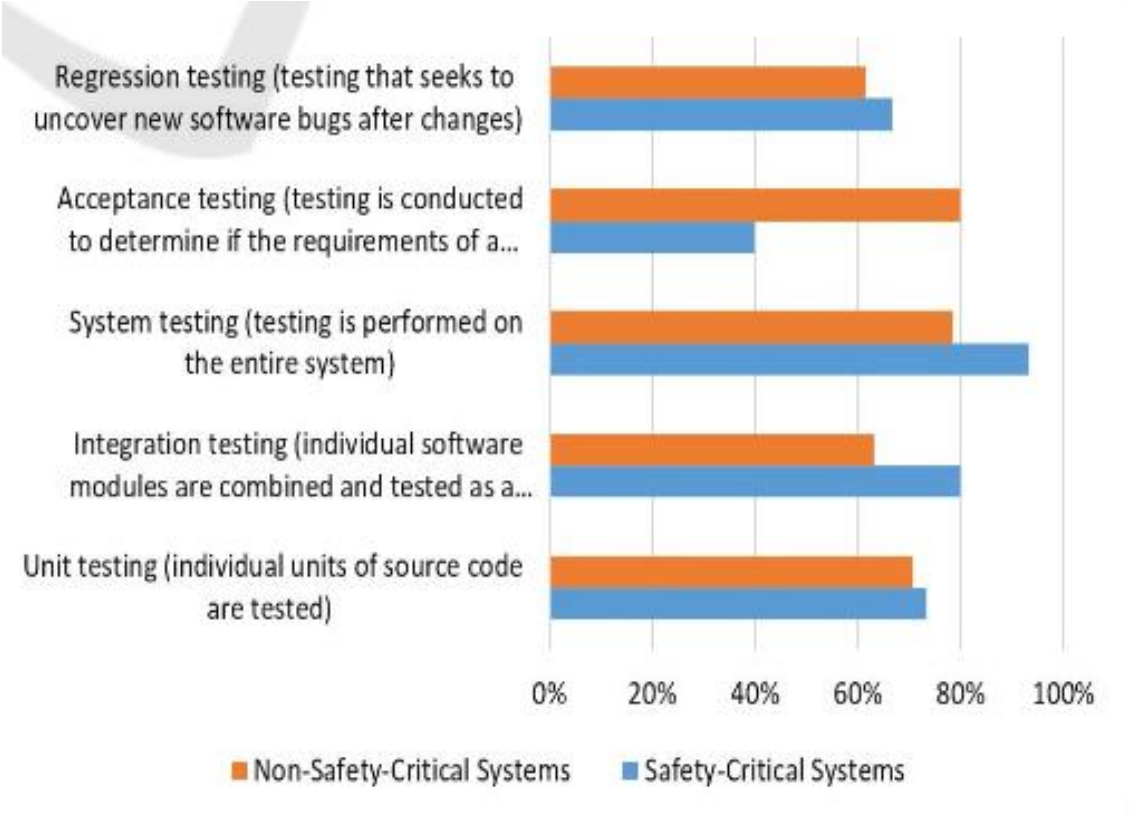

Figure 12. Types of testing implemented. *(Kassab 2018, 359-367)*

The main differences between safety-critical and non-safety-critical systems are, that acceptance testing is not seen as useful in safety-critical systems, but all other types of testing are more used in safety-critical systems than in non-safety-critical systems. (Kassab 2018, 359-367)

A study by Mäntylä, Itkonen, and Ivonen analyzed testing practices in three different companies that developed industrial software (business software with customer integrations, engineering software with customer integrations, and engineering software with directly safety-critical properties). The study reviewed how testing-related practices were distributed in the organizations of these two companies: who does the testing, how effective they are etc. The key findings were:

- Testing is not a specialist task, but a team effort. A significant proportion of the defects was found by non-testers and non-developers (management, sales, etc.). (Mäntylä, Itkonen, and Iivonen 2012, 145-172)
- The end-user is experience is more important than aiming for zero-defects. In a highly specialized field of engineering, the experiences from actual users were indispensable knowledge for testing purposes. (Mäntylä, Itkonen, and Iivonen 2012, 145-172)

One additional finding was linked to the organization of testing. Their results indicated that delegating the testing for a specialized group of testers was linked to the product development model: the testing group was seen necessary to ensure the integrity of the periodical internal mainline releases to offer a reliable baseline for customer projects. In contrast, if the external product releases were periodical, the testing was done by the project development team. (Mäntylä, Itkonen, and Iivonen 2012, 145-172)

A literature review on testing of embedded software by Garousi et al. mapped out what are the domain-specific characteristics for testing embedded software. The key findings that are related to the scope of this thesis were:

- Level of testing. System testing was the most used level by far, followed by unit and integration testing. Regression testing was mentioned in some papers, but not very often (Garousi et al. 2018).
- Testing activities. Test automatization was mentioned in 47% of the papers, but the management of testing was mentioned only in 8% of the papers, although according to Graham and Fewster, management is the most important factor to succeed in the automatization of testing (Graham and Fewster 2012; Garousi et al. 2018).

The findings by Garousi et al. indicate that embedded software, a subset of industrial software, is most often tested as a whole in the representative environment (system testing), but there is a lack of discussion about regression testing and management of testing in the industry.

3. PROPOSED TESTING PRACTICES

The research points that that test automation is a powerful testing tool. In many cases, it is often cheaper in the long run, more efficient in finding errors, and improves the overall quality of the software. But determining when, what and how to automate, finding out the correct level of automation, managing the automatization process, analyzing the results, and maintaining the testing are complex subjects. These subjects are talked about a lot in testing literature, but there is a lack of guidelines on how to approach the problem in mature industrial systems efficiently. This chapter is divided into three sub-chapters which aim to provide a general approach to the testing process from the beginning to implementation of tests and maintaining the tests.

Developing tests for mature industrial systems has some caveats that the test developers and testers should be aware of. Safety-criticality, along with interdependencies between different systems are aspects that often define testing challenges with industrial software. The goal of this proposal is to allow the efficient and fast implementation of testing with good coverage and longevity with a balanced ROI. Comprehensive technical guidelines for test (case) design, in general, are out of scope for this thesis.

3.1 Feasibility analysis economics and management

Testing - especially automated testing - is a cost-intensive process and needs support from the management. Lots of time and resources need to be allocated towards testing related development which is why it is important to recognize if and where the automatization can provide value.

The rationality for test automatization is highly dependent on project details: life cycle, the resources available, and technical details of the project. Some programming environments have proprietary and expensive solutions as others have free and general tools available. The development team might have expertise on the subject or need additional training. The only general way to approach the problem is to research the topic and available tools in the context of the specific project. At least the following topics should be mapped before beginning the process (Figure 13):

- Product life cycle.

  The life cycle of the product should be the first thing to recognize. Automating testing for software that has a short life cycle can be unprofitable and even for a

long-term project, there are always has some opportunity costs involved (chapter 2.2). Approximating with the SD model scenarios 2 and 3 compared to scenario 1 (chapter 2.2), it can be roughly estimated that automating the testing will become profitable if the amount of time used for manual testing is used towards automating 3 - 7 times (depending on the tools and talented manpower available). For example, A project where 10 hours/month time is allocated for testing is likely to be profitable to automate if the project lasts more than 5 months.

- Resources available and resources needed.

  The current available manpower and knowledge must be mapped before proceeding with planning. Testing and test development are separate skillsets (chapter 2, 2.1.3, 2.1.4) that need to be present in the team to perform comprehensive automated and manual testing. Testers need to be available to perform manual testing tasks (usually at least acceptance testing). Developers who do not have any experience in test scripting, result analysis, etc. need training on the tools and techniques to become test developers. This could have a prolonging effect on the time the automatization becomes profitable. If the tools needed are also undefined, it adds complexity to the equation as the time needed to train test development can be difficult to estimate.

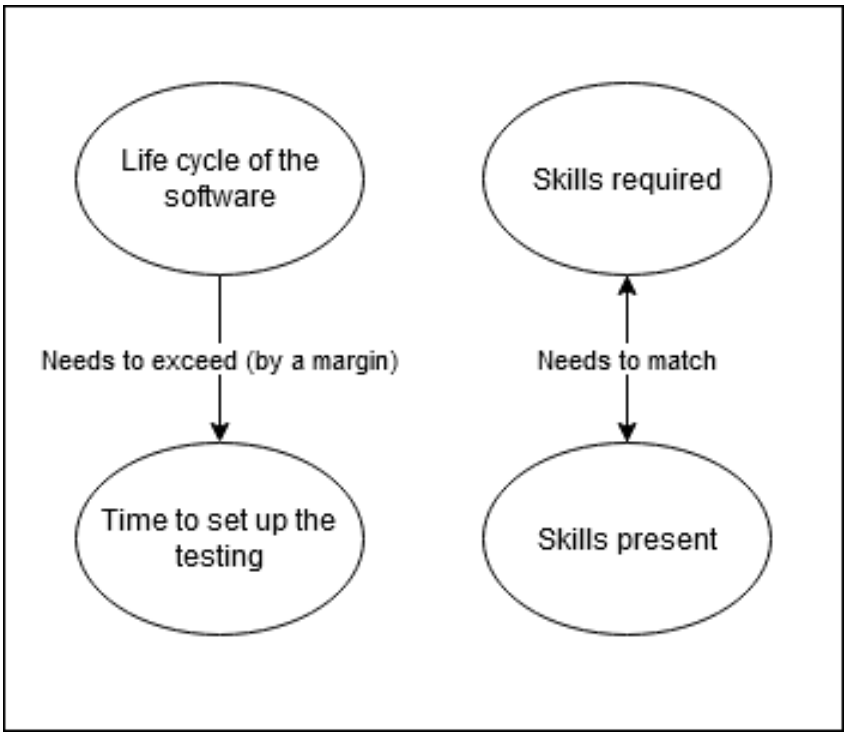

Figure 13. Feasibility of setting up testing

If automatization is seen as a rational investment in general, The best way to get the full management support is to start by collecting metrics from present development

practices and sketching estimates based on the current situation and what could be archived with better test automation (e.g. SD model or another modeling). The metrics could be, for example, calculating the amount of time spent on manual testing, tests run daily, errors discovered, bugs found from SUT, etc.

The good way going forward with the sketching is carrying out proof-of-concept type experimental setups which are good at demonstrating the ROI value of better testing. This could be e.g. developing a minor automatized test set for some features in the developed software that were previously relying on manual testing. The sketching could also be a model for estimating the cost of a bug in the released version of the SUT. This is an especially valid argument in industrial systems, where a software bug in the released version of the software can cause major downtime or even fatal accidents.

Based on chapter 2.1, the sketches and metrics are valuable data to make an argument that automating the tests is financially a good idea: communicating the benefits to the management level is easier with tangible data. In smaller organizations, this might not be as necessary because the management is usually closely linked in the development process and are often more educated in the subject. I also suggest, that even if not needed by management, gathering metrics can be a beneficial task to carry out and continue indefinitely. Metrics are a great way to prove (or disprove) which things have improved the situation (e.g. when later comparing the situation before and after automated tests were implemented) and to find out which things did not work for the current problem. This can help the organization to learn and make better decisions in the future. This data is also useful when test development is in the maintenance phase (chapter 3.3).

3.2 Implementing the tests

Once the organization is committed to set up testing practices, the implementation phase can be started. The most important phase when setting up the testing infrastructure is planning. The whole process should iterative and needs to be re-evaluated when organizational or technical changes occur.

3.2.1 Planning the testing

Planning can be divided into several sub-phases: SUT review, human resourcing, tool selection, and writing the plans (MTP and LTP). Any sketches and proof-of-concept work made in the before testing -phase (chapter 3.1) is valuable and should be revisited.

The planning phase should always begin by evaluating the SUT environment and setting the goals for testing. This is important because the external and project-specific factors like the development phase, safety-criticality, dependencies, and customization needs affect the testing implementation. These factors can be overlooked but will materialize when tests are being implemented. This step can be started by evaluating the following points which affect the decision of what to test automatically and what should be tested manually (Figure 17). These factors should be noted when writing the MTP and affect the decisions on LTP (feature level test planning). Especially in industrial software projects, the second part of the analysis should also involve the end-user or someone else who has experience in the specific field of engineering (chapter 2.5) and can offer invaluable information regarding the features of the SUT.

- Development phase.

  The implementation of automated testing is most efficient when the SUT is in a mature enough development phase. The testing effort should be primarily manual testing for a novel SUT (or new feature of SUT) because it is still in the rapid development phase. This phase is known for frequent braking changes and automated testing would lead to testing errors (false negatives and false positives), which requires high test maintenance. Automatization of testing should be started when there are features of the software that are not going under large structural or breaking changes. Unit testing is the exception to this rule as it helps to catch errors already in the early development phase. Unit tests are also relatively quick to implement, and because of the narrow scope of the test, unit tests rarely break unexpectedly.

- Safety-criticality.

  The safety-criticality of the software must be mapped with a risk analysis to be aware of the potential risks and how the software can affect its environment. The depth of analysis should correlate with the complexity of the software and the environment it affects. Basic risk analysis for a small or intermediate project can be made using a risk assessment matrix with the development team but more complex systems should be analyzed using professional risk assessment services. The object is to discover which are the most harmful and frequent errors the software can cause. Errors in the released version of SUT which can lead to fatalities or serious injuries (with a moderate probability) must be prioritized at testing. This information should later be used when the feature level testing is planned. Safety criticality can be direct or indirect (Figure 14). Mapping out the

indirectly safety-critical parts requires a lot of knowledge about the role of the SUT in its actual operational context. The most common case is shared data between different software.

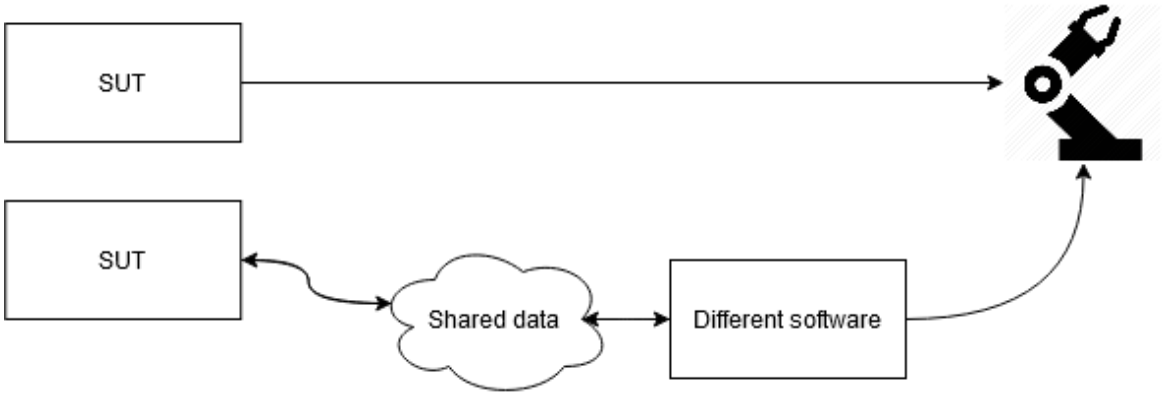

Figure 14. Direct (up) and indirect (below) way features affect e.g. machinery.

- Dependency on third-party applications.

  Dependencies on 3rd party applications (Figure 15) should be evaluated before making the move for broader automatization of testing. If the software has features that require 3rd party applications to function, it is important to investigate if and how these applications can be tested in a realistic environment or mocked. Researching this before beginning will prevent deadlock situations later as there is a risk that these features cannot be tested at all or will require proprietary tools, which affects adversely the maintainability of the tests. Also, the 3rd party features can be subject to sudden changes from outside actors, which can also cause issues with the maintenance of the tests.

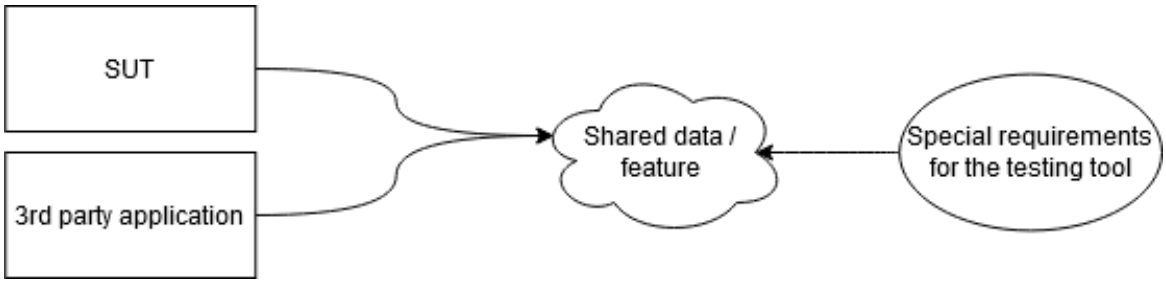

Figure 15. The testability of SUT is affected by the dependencies.

- Customization needs.

  Possible software customization needs should also be recognized. Testing customized features usually require customized testing tools and testing scripts (Figure 16), even though modular development does allow testing of software in smaller, standardized chunks. But e.g. regression and integration testing will be

harder if the software has customized features. Extensive automatization of a widely customized software can lead to high maintenance costs (chapter 2.4) which is why customization usually induces a need for manual testing.

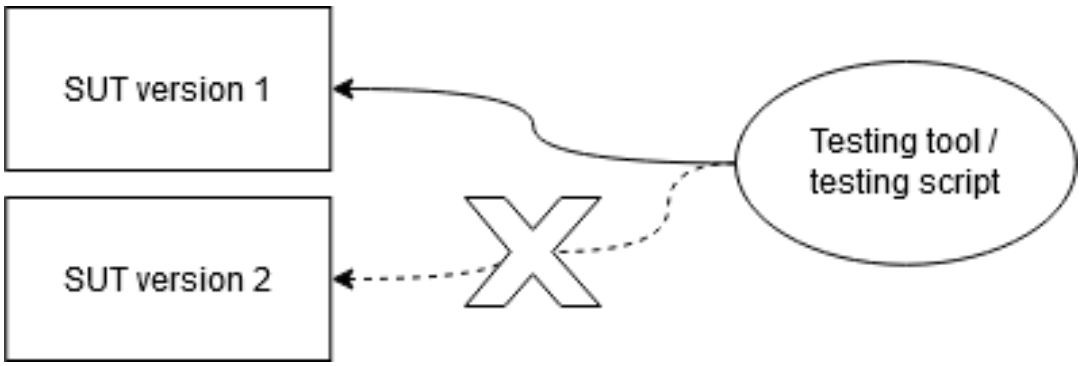

Figure 16. Customization of software can affect the testability.

- Distribution of the software.

  The method of distribution should affect the type and amount of testing needed and the schedule of testing. If the software is distributed with a standalone installer, a bug in the software can have a significant and persisting effect on the use and the rollback of the software can be difficult or even impossible. A web-based software or software that can be updated over the air is generally more tolerant to bugs in released versions, as the updates can be distributed quickly. The time needed for testing should increase when the difficulty of rollback or patching increases and the role of acceptance testing should be emphasized.

- Testing goals.

  Setting testing related goals is a way to make the implementation process more tangible. Based on the research in chapter 2.3.1, the goals should be easily measurable and appropriately sized. Unreachable goals cause disgorgement and will be easily neglected.

Once the external and project-specific factors have been distinguished, the software should be reviewed on a feature specific level. The batch of questions along with the process charts (Figures 17, 19, and 20) creates an approach to find out what and when the features should be tested, what method of testing should be considered, and whether to include the test into a regression test set. Because test development is a time-consuming process, there must also be a prioritization system for developing the tests. This approach uses aspects of the PORT-model (chapter 2.3.2) along with the industrial aspect to create a simplistic model for assigning priority for tests. This creates the base for writing the LTP and forming the test cases.

- Feature maturity and testability.

  If the feature falls into any of the pitfalls of automatization (3rd party dependencies or customization), the testing should be done manually. Spending lots of time-solving problems with customization or 3rd party applications involves a significant opportunity cost. Also, if the feature is still undergoing probable breaking changes, testing should be limited to manual testing only (Figure 17). Automatization of testing for a feature that is still under development would very likely be a waste of resources.

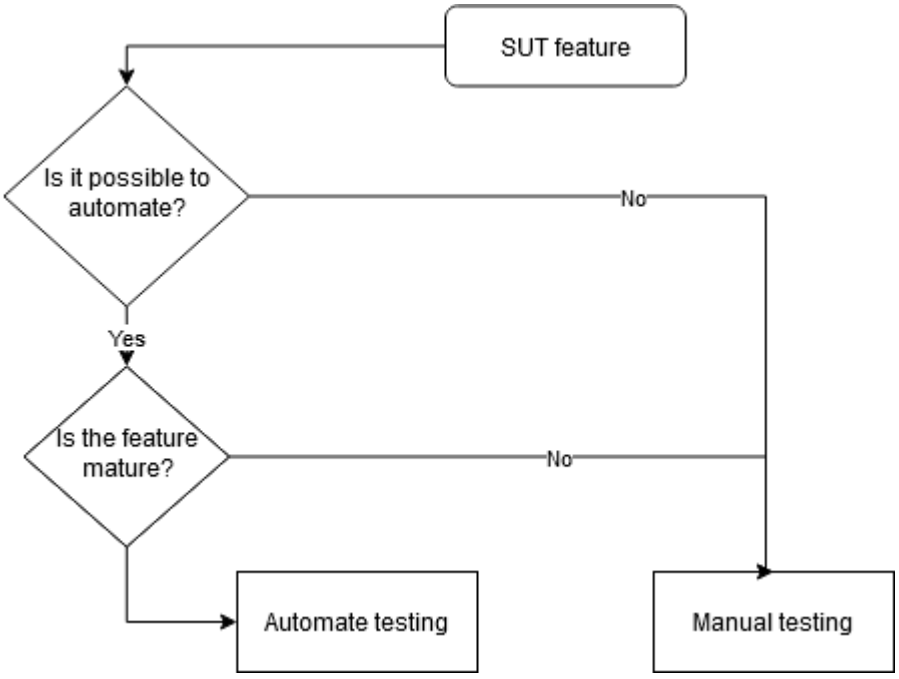

Figure 17. The method for testing.

- Release and development cycle.

  The speed of development is an important factor to consider when testing is being planned because it affects the need, type, and method of testing. The need for acceptance testing is dependent on the frequency of releases as well as on the schedule of a release. Rapid release cycle and short notice time on releases promote the need for automated testing. This is also true for development speed: a rapidly changing source code increases the risk of breaking features that should not have been affected. The most effective way to detect this kind of error is regression testing with broad coverage. Regression test runs are almost impossible to do effectively without automatization in a rapidly changing source code. Ideally, there would be a 100% regression test coverage, but in practice, the order in which the tests are developed needs to be prioritized. The importance of tests should be judged in a case by case approach to save time at

the beginning of test implementation: how critical the feature is and/or how likely it is to fail?

This also relates to the PORT-model fourth theory (requirements volatility), which roots from the fact that changing requirements is one of the main causes for errors.

- Feature criticality.

  This should be looked at from two different perspectives: operational and safety criticality, which relates to the PORT-model first theory (customer-assigned priority).

  The operationally critical feature is a feature that must be working properly to enable the use of the software. These could be e.g. GUI rendering, back-end services running, and connections between the two workings. Operationally critical errors should not end up in the development or released versions of the software and therefore should be tested every time a change is made to the source code, creating a need for automated regression testing.

  The directly safety-critical features (Figure 14) are important for every version of the SUT and promote the need for automated regression testing. The indirectly safety-critical features may not be as critical in the development version of the software but depending on the distribution method, can be extremely critical. These features need to be tested at least in acceptance testing, but the need for a rapid release cycle promotes the need for automated regression testing of the indirectly safety-critical features as well.

- Effect on other features.

  This corresponds with the second and third theories of the PORT-model (developer perceived implementation complexity). It is important to map out the features that have interdependencies (Figure X). These could be e.g. functionalities that use the same data in a shared state or nested structures where the child affects the parent or vice versa. As the complexity of the software increases and the amount of interdependencies grows, testing quickly becomes unpractical to perform manually, which is why the dependent features should be tested with automated regression testing. These features should also be prioritized on test development, as an error in a feature with shared data, interface, etc. will likely cause errors elsewhere as well. (Figure 18).

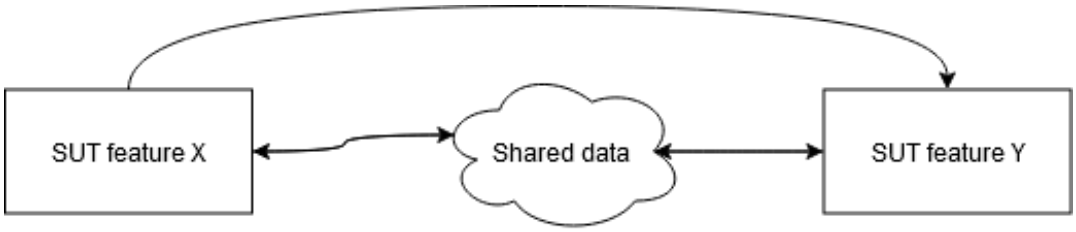

Figure 18. Features can have dependencies through shared data or directly.

- History of errors.

  As mentioned in PORT-model theory three, the development history and real-world user stories are an important source for prioritization, because some features are more likely to contain errors than others. Emphasis should be given to the features which have been found out to cause errors in history. This source of information is especially important to consider in mature projects. It also adds validity to chapter 3.3 and verifies why it is important to collect metrics and results from testing.

  As described in chapter 2.5, finding errors is a team effort. Especially in highly specialized industrial applications, where the usage of the application requires expertise in e.g. mechanical engineering, the experiences of actual users are indispensable information for testers and developers who may not be familiar with mechanical engineering. There should be a way to collect and document the errors and defects that users other than developers or testers (sales, marketing, customer, etc.) find.

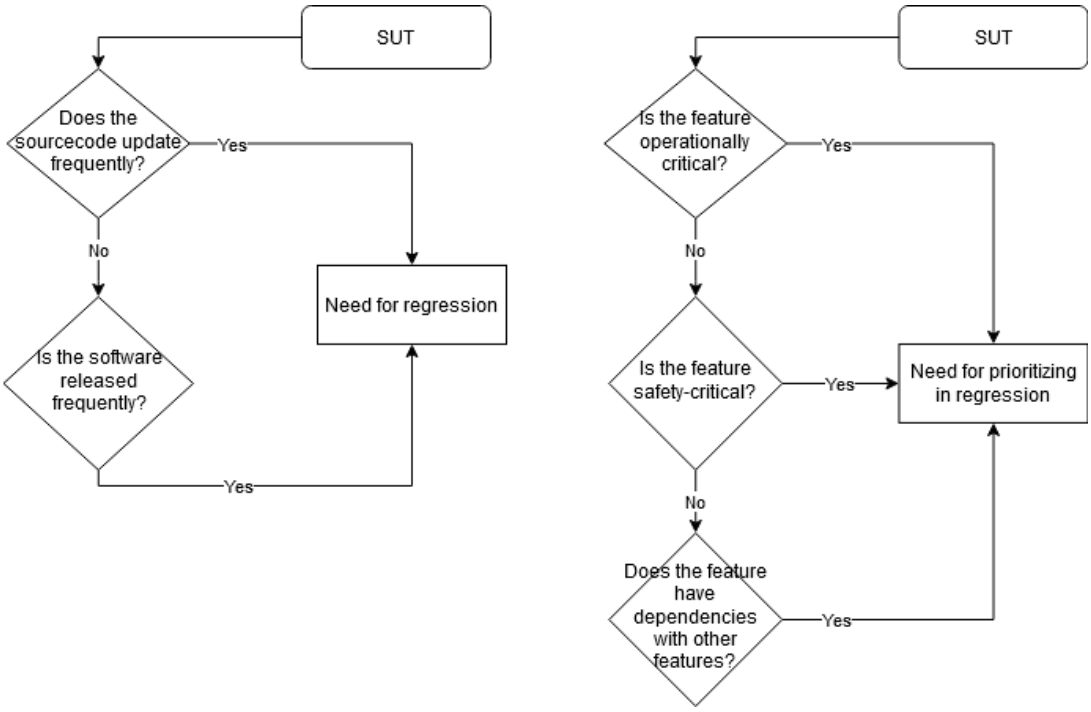

Figure 19. Factors affecting the need and prioritizing of regression testing.

Based on the factors above, I propose a model to form a level test plan quickly. This model does not create an in-depth testing plan nor does it specify the level of testing each feature needs to be included in but offers a fast way to proceed to the implementation of testing coverage for the most important parts of the software. The model helps to decide how the feature should be tested and when it should be tested: it is based on two general yes/no -questions about the project and seven yes/no -questions about the feature which results in a proposal about the testing method (automatic/manual), testing priority (0-3) and the need for regression testing. The questions are:

a. Is the feature mature?
b. Is the feature operationally critical?
c. Is the feature safety-critical?
d. Does the feature have dependencies with other features?
e. Does the feature have 3rd party dependencies?
f. Does the feature have customized properties?
g. Does the feature have a history of bugs?
h. Is the project released often?
i. Does the source code of the project change often?

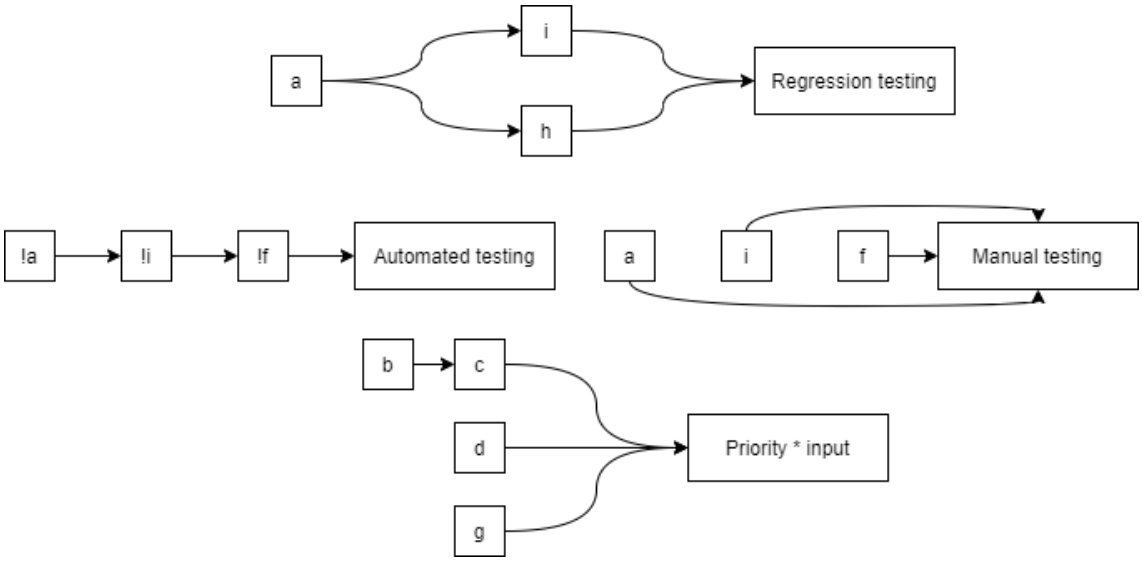

Figure 20. Model for forming a basis for LTP and assign priority.

Tools selection is an important decision in the testing implementation and usually requires some compromises with the coverage, costs, ease of use, etc. and often more than one tool is needed. The selected tools must meet the requirements mapped in the first phase of planning and cover the needs for feature testing planned in the second phase. The maintainability of the testing tool must also be considered, as the selection can have repercussions for the whole life cycle of the project (chapter 2.4). The decision of which tool is the best is always situational and the tool should be evaluated for at least the following points (Figure 21):

1. How well does the tool cover the needed functionalities?

   The most important aspect is to select a tool that meets the requirements. The most common pitfalls in tool selection are related to insufficient functionalities. Testing the possible 3rd party dependencies must also be evaluated. If multiple tools are needed, the aspect of code reusability and other synergy benefits should be considered.

2. Does the tool have longevity and technical support?

   There are a lot of examples of software projects where the support and/or development has been suddenly been deprecated. This is also a risk with testing tools. The safe option is to choose an open-source tool that has an active community or a tool that is backed up by a stable company. Open-source solutions are a safer option in regards to possible customization needs or continued use after the deprecation.

3. How much does this tool require additional training?

The comparison should be made against the current skill set of the team, the SUT, and other considered testing tools. Automated test development especially is essentially software development and using tools with similar development practices is beneficial.

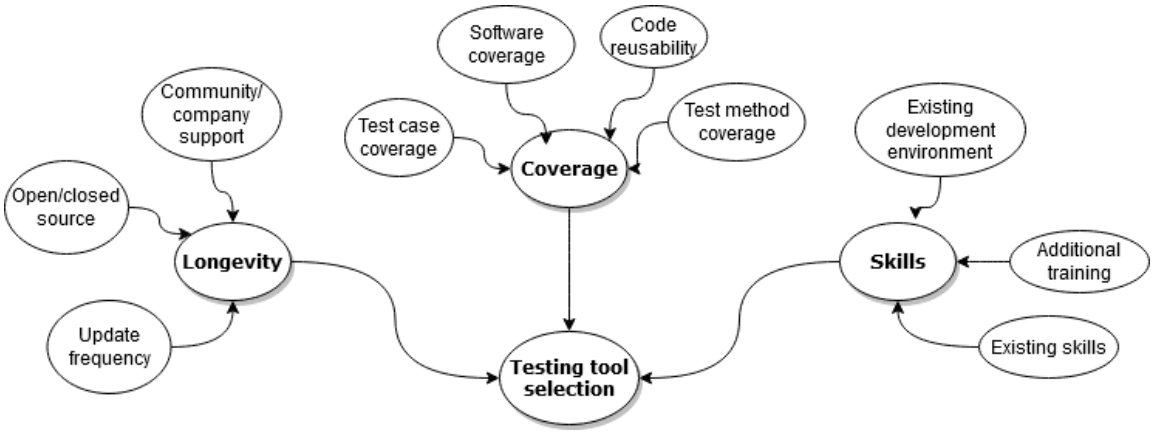

Figure 21. Factors affecting testing tool selection.

3.2.2 Designing the tests

The characteristics of a good test vary between the method and the type of test in question. The need for coverage for a single test is always dependent on the bigger picture and decisions made during planning. Technical details of a test development are not in the scope of this thesis, but the aim is to provide guidelines for test development practices in general and to cover test design caveats in industrial systems. The scale of coverage among the different types of tests is large. This should also be noted in the terms of prioritization to offer as wide as possible coverage with minimal effort. Based on chapter 2.1.5, there is a valid argument that end-to-end type system level tests can be the most efficient way of testing multiple features of code at once. Therefore, at least in GUI applications, the development of end-to-end testing should be the preferred method for creating tests if the goal is to achieve test coverage as efficiently as possible.

The test development should follow the basic principles: the use of standards, reporting, and reusability.

1. Standards and documentation.

   The studies show that at least in the case of manual testing (chapter 2.1.3), the testing practices largely rely upon tacit knowledge, which is against the basic rules of testing (chapter 2). Test development and testing are important to write

and perform using standards and documented properly: naming conventions, file locations, documentation, etc. need to be discussed and agreed with between the team. This will be critical as the project grows and/or new people are joining the team: using proprietary or divergent techniques will require more training and are harder to maintain (chapter 2.1.4). As with software development in general, well-written tests (descriptive variable/function/etc. names and readable structures) may not need additional documentation, but the testing environment, running the tests, etc. needs to be documented properly.

2. Reporting.

   The tests should always be written to support easily understandable and configurable reporting or in the case of manual testing, documented properly. The test result should indicate what has been tested, what kind of inputs has been used, what kind of outputs were expected, what was the actual outcome and in the case of an error, what was the part that failed (Figure 22). The use of deterministic tests (chapter 2.1.5) should be preferred whenever possible as it allows for a much more explicit output validation. For a non-deterministic test, the expected versus actual output validation should be done by a human.

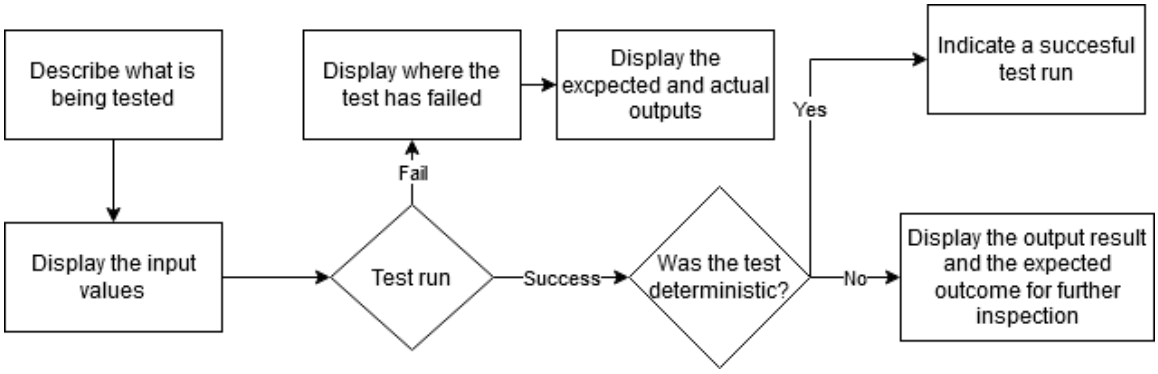

Figure 22. Test design practices to allow good result reporting.

3. Reusability.

   Especially with automated testing, creating tests that are usable in different scenarios is beneficial and increases the profitability of the testing. I would also suggest that the reusability must not be forced with the cost of losing coverage or determinism. The risks of having a false-positive or false-negative test results outweigh the potential profitability benefits.

Based on general knowledge about software development, it can be hypothesized that requiring good reporting practices is also beneficial in the sense that it directs the test development to better practices by forcing the developer/tester to view the problem more broadly. Describing the test case and its results also require thinking about how the test connects to the system and what could cause the test case to fail. This, along with using deterministic testing functions, prevents creating tests that will result in false-negative and false-positive results (the latter potentially being especially harmful in an industrial context).

Testing in an industrial context may have some challenges compared to e.g., web-based software. The industrial systems often consist of several different software with interdependencies – often from different vendors. An additional challenge, safety-criticality, is created by parts of the system that interact with the physical environment by controlling machinery. The possible interaction with the physical environment also affects the input data. Based on the findings in chapters 2.1.5, 2.3.1, and 2.5, These requirements should be considered with each testing type (Figure 23).

Unit testing in industrial software does not differ much from unit testing software in general. The scope of the tests is very narrow, and tests affect a low-level structure. Testing features that depend on e.g. sensor data from the physical environment, requires some research on the possible scope of input data.

Like unit testing, integration testing in an industrial context does not have any major or specific challenges to consider. An integration test case that includes safety-critical features, or has dependencies to a safety-critical function, should be designed with a narrow scope. This is to avoid complexity, which can obstruct the determinism of the test. The possible interactions with the physical environment should also be recognized as with unit tests.

In system testing and acceptance testing, dependencies with other software, and the safety-criticality context across the whole system needs to be considered. The test case needs to be as realistic as possible for the features that are safety-critical or have an impact on the safety-critical features on the dependent systems. Mocking input data or functionalities should be avoided, especially in acceptance testing.

	Safety-criticality	Dependencies	Inputs
Unit	Not necessary to consider	-	Research may be needed to cover the edge cases.
Integration	Should be tested with narrow scope	Internal dependencies that affect safety-critical features.	Realistic data. Mocking required.
System	Should be prioritized	All possible dependencies should be included	Realistic data. Some mocking may be necessary.
Acceptance	Should be prioritized	All dependencies should be included	Real data. Mocking should be avoided.

Figure 23. Testing caveats in the industrial context.

3.3 Test maintenance

Testing should be a long-term commitment. Automated testing, as well as manual testing, needs to be maintained and although the maintainability of the tests is largely dependent on the tool selection (chapter 3.2.1) and test design (chapter 3.2.2), some post-implementation practices help to avoid pitfalls with the long-term commitment. Based on chapters 2.2 and 2.4, the most important things to consider are result reporting, collection of metrics, and revisiting tools, techniques, and plans.

- Result reporting.

  Based on chapter 2.4, the results of the test runs should be made and kept visible. This practice not only keeps the test results accessible and helps to speed up the acknowledgment and fixing of errors but also highlights the effort put into testing.

- Collection of metrics.

  The continuous collection of metrics has been found out to be beneficial (chapter 2.2). The purpose is to form a feedback loop for testing development and the benefits of testing (time and resource savings, time to market, customer satisfaction, etc.).

- Revisiting tools, techniques, and plans.

Software development is by default under continuous change. There will be changes in teams, development and testing tools, languages, etc. Therefore, it is vital to also make revisits to the testing related tools and practices.

4. IMPLEMENTATION

This chapter presents the implementation of the testing approach presented in chapter 3. This chapter is divided into four sections. The first section describes the background of the Configurator project. The second describes the feasibility and economic analysis of testing practices for the Configurator. The third consists of test planning and testing implementation for the Configurator. The fourth describes the testing maintenance work with Configurator.

4.1 Configurator background

The Configurator is an industrial tool used by specially trained system engineers to configure a fleet of AGVs and connect the fleet with related systems, like controller dashboards and runtime environment. It is also used to manage interface-settings, create tasks for AGVs, setting up error handling, etc. The software is a part of a larger context of industrial software. In terms of the *Automation pyramid* (Figure 1) the configurator falls into the MES level.

The Configurator project is currently in the stage where the second release candidate has been published and the first stable release is being planned. It is in a stage where the software is mostly quite mature and the features are stable, but the development is still continuous and new features are also being developed.

The Configurator is stand-alone software that uses cloud services for some administrative features. The front-end of the software is built with the Electron framework which acts as a container for the software. It is developed using the Vue.js JavaScript framework using TypeScript as the main language (some parts also developed with vanilla JavaScript). The software also uses a locally run backend, which acts as an interface between other systems and offers some background functionalities. The backend is developed with C#. The full architecture of the Configurator is presented in Figure 24

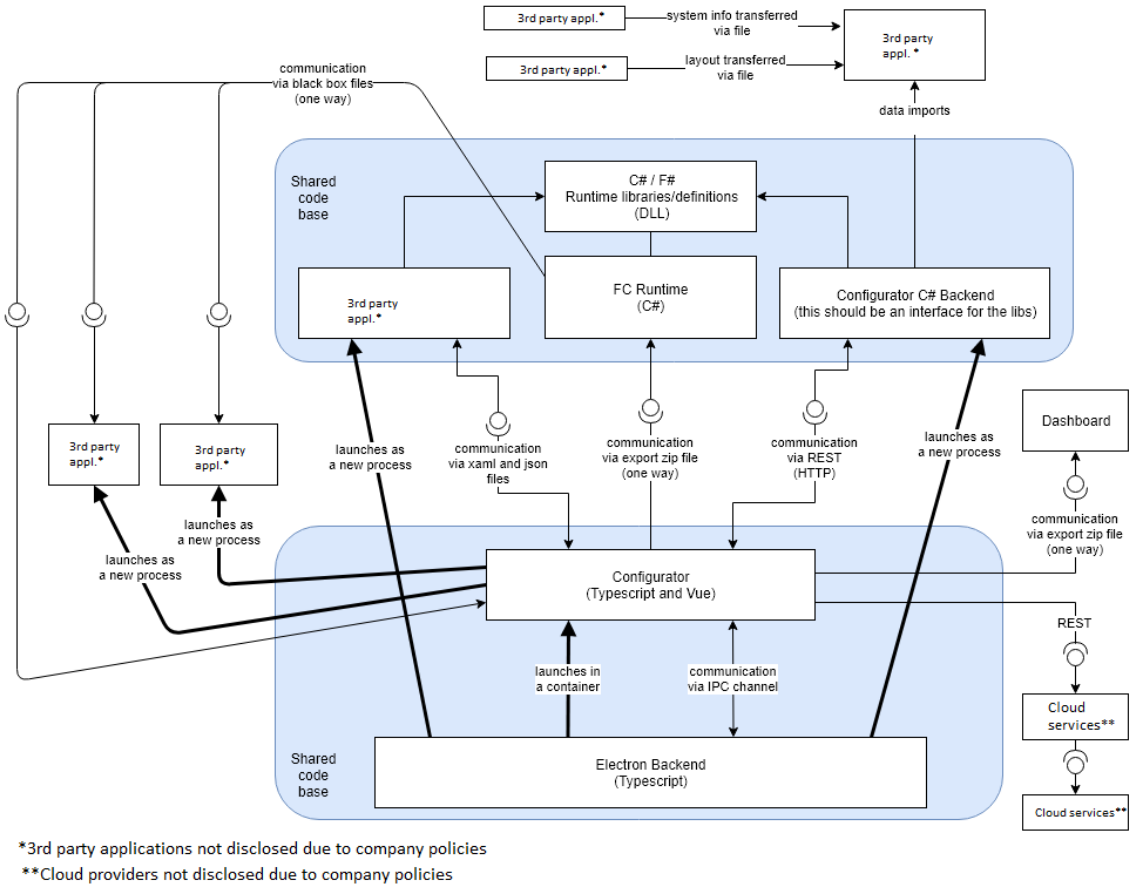

Figure 24. Configurator architecture (some parts masked due to company policies).

The project is developed using the Scaled Agile Framework (SAFe) approach, where the development is planned in increments, and increments are divided into sprints of two weeks. The project is part of a mono repository which includes the Configurator and other related software.

The testing of the Configurator is mostly manual testing done by developers (pull-based development). There is also an existing set of automatically run unit tests that mostly test the validity of the configurations that are made with the Configurator and some integration tests that validate the basic functionalities needed between the Configurator and runtime systems. Some proof-of-concept work has also been done to automatize GUI testing. Initial work on available testing tools has also been mapped.

The development of Configurator has been rapid, and the testing of the software has not been up to the speed of the development. There have also been some unsuccessful attempts setting up automated integration-/system level testing which failed due to incompatible testing tools. This, along with the built-up complexity, has led up to the situation where the team must use an ever-increasing amount of time for manual testing

of the software to keep the functionalities validated, which is a repetitive but time-consuming practice and is quite prone to human error.

The SAFe development approach also requires keeping the project ready to be released, which amplifies the need for retaining the development branch of the software stable. This compounds the need for automated testing with as much coverage as possible, but at least to keep the critical features always validated.

4.2 Setting up the testing practices for Configurator

The Configurator is a part of a larger software ecosystem, which serves projects with a long lifecycle. This means that the software needs to offer support to the projects for a long time. The development of the software ecosystem is also still in a relatively early phase. Therefore, it is reasonable to expect that the lifecycle of the software is long (at least years). This means that the economical fundaments for automatization of the testing exist.

The Configurator team currently consists of three full-time developers, who currently also perform the majority of the testing and develop the tests. But some designated testers are currently being introduced to the project, who will mainly be responsible for acceptance level testing. The developers are experienced in test development, but it is not the main field expertise for any of the developers. The team also has some formal knowledge in testing. The designated testers have enough expertise and training in formal testing methods. Currently, testing is mostly done *on-demand* based on the need along with the feature development. Although the overall situation is in terms of skillset and resources is decent, there still is some shortage of manpower and lack of systematical approach on testing.

To create a baseline before the study, the development team collected metrics to estimate the time used for manual testing (pull request reviews). The test period was an interval between 12.5.2020 – 4.6.2020 (roughly 1,5 sprints). The time used for testing was estimated by the developers themselves with a sensitivity of 15 minutes (Figure 25).

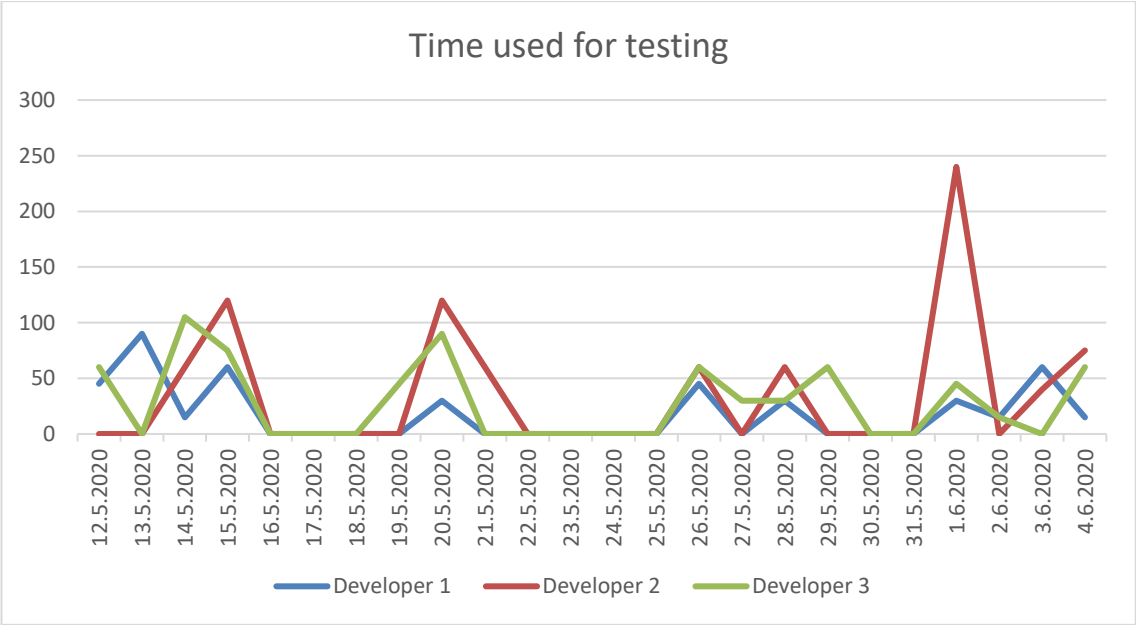

Figure 25. Time used for testing (in minutes) for each developer.

On average, the developers used 36 minutes per day for testing (the weekends ignored). The variation between developers ranging from 24,16 minutes to 46,38 minutes. Based on this value, the approximate amount of time used for manual testing per month (21 working days) is 12,6 hours.

With a rough approximation of the SD model scenario 3, it can be estimated that the time needed for automatization to become profitable in this scenario is 4 months. The approximation is not entirely valid, because this would require a full-time commitment to testing development and the pull-based development will always include some manual testing and cannot be fully automatized. But it does demonstrate that the ROI of testing will likely be positive in this project.

The emergence of the dedicated testing team is one indication that there is a long-term commitment from the management to support the test development and testing is currently seen as an important part of development. But although the general support is good, there is still some lack of resources: none of the developers or testers can commit full time for test development for the Configurator. This creates a need to emphasize the importance of testing, but mainly to prioritize the testing and test development to offer good coverage for the most critical features and fast as possible. During the writing of this thesis, a designated team of testers has also been introduced to the project to take part of the testing responsibilities. The role of the testing team is mainly in developing acceptance level tests to always keep the product ready for release.

4.3 Planning the testing for Configurator

The Configurator currently lacks formal plans for testing. There are some developer guidelines written on the development documentation, but the information is mostly incidental or technical details. The plan is developed using the approach for planning the testing (chapter 3.2).

The first step is to map the general testability and demand for testing for the Configurator. The general factors are the development phase, safety-criticality, 3rd party applications, customization, and distribution of the software.

1. Development phase.

   The Configurator is in the semi-mature phase. The first pre-release -versions for a closed group of users have been published and the first release candidate is currently in development. The structure of the software has been quite stable since the first pre-release-version was published. There are also no plans to make big breaking changes to existing features but develop new features and improve the old features.

   In terms of the development phase, there are no major hindrances for expanding the automatization of testing. Rather, the situation promotes the need for more automatized testing coverage, especially for regression testing, to verify the integrity of the old features as new features are added, and for acceptance testing, to verify the released versions. For the new features, manual testing should still be the primary method of testing.

2. Safety-criticality.

   The Configurator is a tool used to configure fleets of AGVs and the supporting systems. It does not have a direct impact on the physical environment, but there are some potentially harmful indirect effects if the configurations are defective.

   The most critical risks with the Configurator are related to the exported files that are used by the runtime- and other control systems. The potential outcome of the critical risks ranges from physical damage to the environment to production halts and economical damage.

3. Third-party applications.

   The Configurator does have some dependencies with 3rd party applications and services. It uses a .NET based external executable to create, read, and update a narrow part of the configuration. Two external executables are also used for

read-only purposes. Some features require local windows services that are not included in the installer to work properly and some features rely on cloud-based services. These dependencies have some effects on the testability of the Configurator.

Based on previous research done in early 2020, some of the third-party dependencies cannot be tested with the Configurator automatically with the testing tools that the developers currently use. The testing is also limited to black-box testing methods only, as there is no direct access to the source code of these applications. Although some of the 3rd party applications and services can be mocked and automatized with the current tools, most of these features still rely on manual testing on system and acceptance levels. The proof-of-concept work is done later (third step: tool selection) will determine the actual testability of the 3rd party applications.

4. Customization needs.

   Currently, there are no major needs for customer-specific versions of the software. Some features are controlled with user access levels which are granted based on the login data. This does not hinder the automatization as they do not require customized versions of the software.

   But although there are no customer-specific versions of the software, the challenge is that there can potentially be incompatible versions between the different software in the environment. This has a potentially adverse effect on testing, as the possible combinations of the different versions can grow out of hand. This promotes the need for automated regression tests.

5. Distribution of the software.

   The Configurator is distributed as a stand-alone desktop installer for Windows-based systems. The software does not support OTA (over the air) updates and everything from bug fixes to new features must be delivered with a new installer package.

   The current situation promotes the need for better overall testing coverage as the cost of fixing a bug in a released version can be high.

6. Goals.

   The short-term goal is to keep the current features validated with automated regression testing. This would improve the confidence among the developers if the

testing would ensure that the development work will not break the basic functionalities of the software. It would also allow more frequent releases of the software on shorter notice. In the long term, the aim is to also minimize the time spent on manual testing.

Based on the general analysis, the testability of the Configurator is good and there is readiness and a need for extensive automatization of testing. The biggest hindrances are related to the 3rd party dependencies. Although the Configurator does not directly affect any safety-critical systems, there are some features on the Configurator that have an indirect effect on safety-critical systems.

The second phase of the approach is to make the level test plans by breaking the features into adequately sized entities for the testing level in question. The features of the Configurator have a high emphasis on the GUI. This means that a logical place to start breaking down the features is by examining the different views and the functionalities in the views. Special emphasis is given to the features that were found to hold a risk affecting safety-critical systems.

Using the *model for forming a basis for LTP and assign priority* (chapter 3.2.1) the Configurator features were evaluated in a table format (Table 2, Appendix A). The result of the evaluation revealed eight features with a priority value of three that need regression testing and can be automatized. There was only one feature that based on the analysis, is not possible to test automatically, but needs regression testing with a priority value of three. The first set of tests are developed to cover these features and the plan is to continue implementation gradually in the order by priority. The full list of features to test is included in Appendix X.

Table 2. Features to be tested in the first set of regression tests.

FEATURE	TESTING MTD	Regression	Dependencies?	Historical errors?	Is critical?	Priority value (0-3)
Data import	AUTO	Yes	1	1	1	3
Feature switch	AUTO	Yes	1	1	1	3
Editing a model	AUTO	Yes	1	1	1	3
Data import	AUTO	Yes	1	1	1	3
Data generate	AUTO	Yes	1	1	1	3
Edit system rule	AUTO	Yes	1	1	1	3
Enable/disable maintenance order	AUTO	Yes	1	1	1	3
Export configuration	AUTO	Yes	1	1	1	3
Edit workflow	MAN	Yes	1	1	1	3

The third step in the planning phase of this approach is to select the tools for testing. The framework for unit testing already exists: Mocha has been used to execute unit tests and some integration level tests. It uses the same language as the main project (JavaScript/TypeScript) and is easy to set up with current knowledge. It is also an open-source project that is popular among JavaScript developers. For regression testing, the project already has an existing framework to execute scheduled test runs for different test sets with Jenkins. Manual tests are carried out without any specific tools.

Some proof-of-concept studies have been conducted to find good solutions for integration-/system testing. Based on the previously made studies on integration-, system- and acceptance testing, the possible candidates are:

- Spectron.

  Automated end-to-end testing framework for Electron applications that offer good coverage for both integration and system testing levels: the scope of the tests can be small entities of UI (integration tests) or larger tests that verify the integrity of the whole UI (system tests). It is an open-source project that is maintained by the community, but the project is quite mature and has an active userbase. It is also the most used end-to-end testing solution for Electron applications and likely will be maintained in the future. The tests for Spectron can be written with JavaScript and run with Mocha, which is familiar with the current development team.

- Robot framework

  The Robot framework is a generic test automation tool that is used by the testing team in previous projects. It has a long lifespan and has been used since 2008 and has remained active ever since. The most recent release is from 2019. It offers wide coverage and is suitable for different types of software and could potentially solve some of the issues with 3rd party dependencies (accessing background software during the test), but there is still a lack of proof of concept work with this tool in this project.

- Vue Test Utils

  Vue test utils is an automated unit testing framework for Vue.js. It offers unit and integration testing capabilities for Vue-based projects. It is an official tool backed up by Vue-organization and is, therefore, a good tool in the perspective of longevity and maturity. The development team has some experience in using it, so the learning curve should be relatively low. There is a potential issue with

the testing coverage in the Configurator context: because it is a Vue testing tool, it would narrow the non-Vue parts out of the testing scope. Currently, Spectron is seen as a better option because it offers wider coverage.

- Cypress

  Cypress testing was used for system-level end-to-end testing at the beginning of the project, but it was not suitable for the Electron framework. It required a lot of mocking which made the tests unstable.

Based on the research and previous knowledge, Spectron is chosen as a tool for integration- and system-level testing in the development environment. The testing team, who are mainly responsible for acceptance testing, has made some initial research on the suitability of the Robot framework for the Configurator and will develop a proof-of-concept test set with it.

The chosen tools for testing are:

- Unit testing: Mocha
- Integration testing: Mocha and Spectron with Mocha
- System testing: Spectron with Mocha
- Acceptance testing: Robot framework and Spectron

Some other things were also considered during the analysis phase. The importance of collecting errors from different sources and harness the end-user experiences has been noted. Currently, the majority of the reported errors and defects are found by developers and other stakeholders that are directly related to the development of the software. They already have a direct channel to report errors and defects. Other stakeholders rely on indirect messaging or support tickets. There are currently plans to implement a method for creating error/defect reports directly from the Configurator and it is likely to be implemented as a feature in the near future.

4.4 Designing the tests

Developing the system- and acceptance level automated end-to-end testing was prioritized in the initial phase of test implementation. This was partly due to the lack of such tests (previously testing was done only on unit-/integration level) and partly because of the wide coverage offered by end-to-end tests (chapter 3.2.2).

As the tests for the Spectron testing framework can be written with the same language and ran with the same script runner as the existing unit tests (JavaScript/TypeScript

and Mocha), the standards and conventions for the Spectron tests were therefore already mostly agreed upon among the developers and testers. The Spectron end-to-end tests were possible to develop with the same format as the old unit tests (Figure 26).

```
it('Empty blockable group name', async () => {
    // Setup
    const state = testState();
    state.project.layout.blockableGroups.push(
        { Name: '', Segments: [3, 4], Stations: [1, 2], uuid: 'id1' },
    );

    // Act
    const result = await TestSuiteRunner.run(state, blockableGroupsSanityChecks);

    // Expect
    expect(result.failedTests).to.have.members([
        SanityCheckErrorCodes.BlockableGroupMustHaveName,
    ]);
});
```

```
it('Testing export with EasTest project', async function() {
    // Setup
    openTestProject()

    // Act
    const exportTab = await app.client.$('div=Export');
    await exportTab.click();
    const exportButton = await app.client.$('#export-project-button');
    await exportButton.click();
    const runFullCheckButton = await app.client.$('button=Run full sanity check');
    runFullCheckButton.click();

    // allow time for sanity check to run
    await new Promise( (resolve) => setTimeout(resolve, 20000) );
    const result = await app.client.$('#sanity-check-navigation > div.exported');
    const resultText = await result.getText();

    // Expect
    return expect(resultText).to.equal('Exported: Yes');
});
```

Figure 26. The similarity of the new end-to-end tests (below) and old unit tests (above).

The acceptance tests, which are developed with the Robot framework, are not compatible with any of the existing tests. But these tests are solely the responsibility of the testing team and therefore do not need to comply with the standards and conventions of the development team.

The reporting of the test results for the Spectron (and unit) tests is based on the script runners (Mocha) way of running the test. For a large set of tests (like unit tests), the

tests are divided under a header, which describes the category of the tests. For the actual test, what is being tested and possible input values are written as a sub-header. The result of each test is either pass (green checkmark) or fail (red highlight) and the summary of results are shown at the end of the test run (figure x).

Figure 27. Set of passing (right) and failing (left) tests.

The tests are written with a deterministic style by making specific assertions about the output of tests (Figure 26). This style also helps to interpret the results of tests: an assertion error is shown if the expected result did not match the actual result of the test (Figure 27).

The industrial system caveats were also considered, and the tests were able to be created using realistic input data. The project files used were real project files and all imported files were imported using the actual filesystem of the Configurator. The test cases for the safety-critical features were designed to cover as realistic use cases as possible with all of the dependencies. The dependencies were mainly due to shared data, so the test case was created to use the same data in different views.

The reusability aspect of test design is considered by creating e.g. setup functions, that create a similar startup scenario for each test. Besides the setup functions, the development of reused functions is still quite limited, but the technical similarity of the unit and Spectron end-to-end tests could offer some potential cohesion benefits.

4.5 Implementation of the tests

During the writing of this thesis, the development team and the testing team were able to implement the first set of end-to-end tests for the Configurator. The first set of tests was based on the features that were listed in table 2 (chapter 4.3). Some of the features that were initially planned to be tested on the first set were excluded (*feature switch* and *enable/disable maintenance order*) after later considerations. On the other hand, some features that were not included in the initial list were included in the test set. This was mainly due to the proof-of-concept work done before and during (chapter 4.3) the writing of this thesis. The results of the proof-of-concept work done by the testing team also proved that some of the features with 3rd party dependencies (login user, workflow edit) were possible to test automatically with the Robot framework. The long-term maintainability of these tests is still unclear and will partly be suspect to sudden changes from outside actors.

There was also some uncertainty with the test development roles between the developers and the new testing team. This caused some overlapping with the test development and some confusion about the testing level responsibilities. This is mainly due to the novelty of the new situation and most likely will balance out in the future.

The set of tests were automated to run as regression tests by attaching the tests to a script that runs in Jenkins every day. This also provides automatic and daily result reports from the tests.

5. RESULTS AND CONCLUSIONS

5.1 Results of applying the approach

Before the implementation of testing practices described in chapter 3, the Configurator project was seen to be lacking proper testing practices. Although the project did include some automated testing, it still mainly relied on manual testing methods, which were starting to be inefficient as the project has grown. Because of the maturity and the relatively large size of the project, there was also a need to find out the most important parts to be tested and get good test coverage efficiently.

RQ1. The approach helps to analyze the feasibility of testing in a mature software project: the objective is to identify whether testing is economically more sensible to automate or perform manually. The approach identifies management support as a key component of success for introducing testing to a project and proposes some practices to get the support needed for testing from the management. The maintainability of testing was identified to be highly dependent on: tool selection, test design, and reporting practices The Configurator project was analyzed with the feasibility analysis and was seen ready for extensive test automation. The management support towards testing was already good, and there was no need for justifying the investment towards testing, but the approximated ROI indicated that the investment would be profitable in 4 months. The success in regards to the maintainability of the tests cannot be evaluated in the time-scope of this thesis.

RQ2. The most critical parts of the software could be identified by analyzing the SUT using the two-phase analysis and the *model for forming a basis for LTP and assign priority*, which highlights the most critical features of the software and indicates how the features should be tested efficiently and to identify where to start testing in a mature project to archive the best possible coverage. This approach was used for the Configurator. Although there is a lack of reference, because of the uniqueness of each software project, the initial results indicate that the methods of testing and priorities identified for each feature by this model were relevant and also act as a good platform for opening a discussion about the testing priorities. The possible improvements to this model are discussed in chapter 5.3.

RQ3. The approach identified several factors that affect how the tests should be designed to be efficient and appropriate in an industrial context. The main factors affecting test design, in general, were related to standards and documentation, reusability, and reporting. There were also identified to be some caveats related to industrial systems, mainly caused by the nature of industrial software (a system of systems) and the safety-criticality of the software. This was applied to the implementation of tests for the Configurator. The results of the initial (only the first set of tests were implemented during the writing of this thesis) implementation indicate that this approach was able to identify relevant factors and guided the test design towards good design practices.

5.2 Effects on the development

One of the main issues was that the developers had to use a lot of time repetitively validating basic functionalities with every pull request and to keep the software ready for release. This was inefficient and still caused some lack of confidence in the validity of the software.

At the time of writing this thesis, the development of tests using the methods described in chapter 4 is still an ongoing process. The initial results indicate that the implementation of the first set of tests is promising: the most critical set of features (chapter 4.3) have been covered and the coverage will be improved incrementally.

With only the current limited set of tests, it is still too early to evaluate the effects on the time used for manual testing (chapter 4.2) and evaluate the ROI from the perspective of saved time, but based on the initial experiences among the developers, the confidence towards the validity of the software has increased.

5.3 Discussion

This approach presented a general approach to introduce testing to a mature project in industrial systems which covered the whole process from the beginning (economic feasibility and management) to the deployment (planning and design) and the maintenance of testing. In this chapter, the possible future work and improvements to the approach are analyzed.

The most beneficial possible improvements to this process could be improving the *model for forming a basis for LTP and assign priority* by adding more and a larger scale of parameters to decide the testing method and assigned priority. It could also be improved to provide some proposal about the testing type for a feature, but this would require a much more detailed analysis of the feature.

Another topic that was left out of the scope of this thesis, is the organizational level planning of testing and developing a general approach/model for creating MTP. This is a major challenge, because each organization is unique, and the needs and requirements for testing can be vastly different depending on the size, maturity, skills of the organization, and the software that is being tested.

The future work on this topic could also be extended to cover more technical topics of test design. One of the main points of this thesis was to find out an efficient way to implement testing practices to a project. In this generalized approach, a lot of technical details were left out of scope. The work could be extended to e.g. research test design efficiency and more in-depth analysis of test design in industrial systems.

REFERENCES

A. Causevic, D. Sundmark, and S. Punnekkat. 2010. "An Industrial Survey on Contemporary Aspects of Software Testing.". doi:10.1109/ICST.2010.52.

Ammann, Paul and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511809163. https://www.cambridge.org/core/books/introduction-to-software-testing/EE74986D7F75690F6E08532775D64DDF.

Boehm, Barry. 2006. "A View of 20th and 21st Century Software Engineering." Shanghai, China, Association for Computing Machinery, . doi:10.1145/1134285.1134288. https://doi.org/10.1145/1134285.1134288.

C. Persson and N. Yilmazturk. 2004. *Establishment of Automated Regression Testing at ABB: Industrial Experience Report on 'Avoiding the Pitfalls'*. doi:10.1109/ASE.2004.1342729.

Carlshamre, P., K. Sandahl, M. Lindvall, B. Regnell, and J. Natt och Dag. 2001. "An Industrial Survey of Requirements Interdependencies in Software Product Release Planning."Aug. doi:10.1109/ISRE.2001.948547.

Crnkovic, Ivica. 2008. "Are Ultra-Large Systems Systems of Systems?" Leipzig, Germany, Association for Computing Machinery, . doi:10.1145/1370700.1370716. https://doi.org/10.1145/1370700.1370716.

Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, K. Petersen, and M. V. Mäntylä. 2012. *Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey*. doi:10.1109/IWAST.2012.6228988.

Garousi, Vahid, Michael Felderer, Cagri Karapicak, and Uğur Yılmaz. 2018. "Testing Embedded Software: A Survey of the Literature." *Information and Software Technology* 104. doi:10.1016/j.infsof.2018.06.016.

Garousi, Vahid and Mika V. Mäntylä. 2016. *When and what to Automate in Software Testing? A Multi-Vocal Literature Review*. Vol. 76. doi:https://doi.org/10.1016/j.infsof.2016.04.015. http://www.sciencedirect.com/science/article/pii/S0950584916300702.

Gleb Bahmutov. 2020. *Why End-to-End Test using Cypress with Gleb Bahmutov*. Vol. Dev.to.

Graham, Dorothy and Mark Fewster. 2012. *Experiences of Test Automation: Case Studies of Software Test Automation*. 1st ed. Addison-Wesley Professional.

Hollender, M. 2010. *Collaborative Process Automation Systems* ISA. https://books.google.fi/books?id=DamUz1oxs9QC.

IEEE. 2008. *IEEE Standard for Software and System Test Documentation*.

ISO/IEC/IEEE. 2013. *ISO/IEC/IEEE International Standard - Software and Systems Engineering -- Software Testing --Part 3: Test Documentation*.

J. Itkonen, M. V. Mantyla, and C. Lassenius. 2009. "How do Testers do it? an Exploratory Study on Manual Testing Practices.". doi:10.1109/ESEM.2009.5314240.

J. Lee, S. Kang, and D. Lee. 2012. *Survey on Software Testing Practices*. Vol. 6. doi:10.1049/iet-sen.2011.0066.

Kassab, Mohamad. 2018. "Testing Practices of Software in Safety Critical Systems: Industrial Survey.".

Kumar, Divya and K. K. Mishra. 2016. *The Impacts of Test Automation on Software's Cost, Quality and Time to Market*. Vol. 79 Elsevier.

Luo, Qingzhou, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. "An Empirical Analysis of Flaky Tests." Hong Kong, China, Association for Computing Machinery, . doi:10.1145/2635868.2635920. https://doi.org/10.1145/2635868.2635920.

Mäntylä, Mika V., Juha Itkonen, and Joonas Iivonen. 2012. "Who Tested My Software? Testing as an Organizationally Cross-Cutting Activity." *Software Quality Journal* 20 (1): 145-172. doi:10.1007/s11219-011-9157-4. https://doi.org/10.1007/s11219-011-9157-4.

Myers, Glenford J. and Corey Sandler. 2004. *The Art of Software Testing*. Hoboken, NJ, USA: John Wiley & Sons, Inc.

Ng, S. P., T. Murnane, Karl Reed, Doug Grant, and T. Y. Chen. 2004. "A Preliminary Survey on Software Testing Practices in Australia."02. doi:10.1109/ASWEC.2004.1290464.

Nidhra, Srinivas and Jagruthi Dondeti. 2012. "Black Box and White Box Testing Techniques-a Literature Review." *International Journal of Embedded Systems and Applications (IJESA)* 2 (2): 29-50.

Ostrand, Thomas J. and Elaine J. Weyuker. 2002. "The Distribution of Faults in a Large Industrial Software System." *SIGSOFT Softw.Eng.Notes* 27 (4): 55–64. doi:10.1145/566171.566181. https://doi.org/10.1145/566171.566181.

Ostrand, Thomas J., Elaine J. Weyuker, and Robert M. Bell. 2004. "Where the Bugs Are." *SIGSOFT Softw.Eng.Notes* 29 (4): 86–96. doi:10.1145/1013886.1007524. https://doi.org/10.1145/1013886.1007524.

Pierre Audoin Consultants GmbH. 2011. "Growth Market Software-Testing." . https://www.pac-online.com/download/7194/121155.

Ramler, Rudolf and Klaus Wolfmaier. 2006. "Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost." Shanghai, China, Association for Computing Machinery, . doi:10.1145/1138929.1138946. https://doi.org/10.1145/1138929.1138946.

S.M.K, Quadri and Sheikh Umar Farooq. 2010. "Software Testing – Goals, Principles, and Limitations." *International Journal of Computer Applications* 6. doi:10.5120/1343-1448.

Sahaf, Zahra, Vahid Garousi, Dietmar Pfahl, Rob Irving, and Yasaman Amannejad. 2014. "When to Automate Software Testing? Decision Support Based on System Dynamics: An Industrial Case Study." Nanjing, China, Association for Computing Machinery, . doi:10.1145/2600821.2600832. https://doi.org/10.1145/2600821.2600832.

Sawant, Abhijit, Pranit Bari, and Pramila Chawan. 2012. "Software Testing Techniques and Strategies." *International Journal of Engineering Research and Applications(IJERA)* 2: 980-986.

Srikanth, Hema and Sean Banerjee. 2012. "Improving Test Efficiency through System Test Prioritization." *Journal of Systems and Software* 85 (5): 1176-1187.

doi:https://doi.org/10.1016/j.jss.2012.01.007. http://www.sciencedirect.com/science/article/pii/S0164121212000027.

Taipale, Ossi, Jussi Kasurinen, Katja Karhu, and Kari Smolander. 2011. "Trade-Off between Automated and Manual Software Testing." *International Journal of System Assurance Engineering and Management* 2 (2): 114-125. doi:10.1007/s13198-011-0065-6. https://doi.org/10.1007/s13198-011-0065-6.

Torkar, Richard and Stefan Mankefors-Christiernin. 2003. "A Survey on Testing and Reuse."12. doi:10.1109/SWSTE.2003.1245437.

University of Cambridge. 2013. "Failure to Adopt Reverse Debugging Costs Global Economy $41 Billion Annually." . https://totalview.io/press-releases/university-cambridge-study-failure-adopt-reverse-debugging-costs-global-economy-41.

Varma, Tathagat. 2000. "Automated Software Testing: Introduction, Management and Performance." *ACM Sigsoft Software Engineering Notes* 25. doi:10.1145/505863.505890.

Y. Yu, H. Wang, G. Yin, and C. X. Ling. 2014. "Reviewer Recommender of Pull-Requests in GitHub.". doi:10.1109/ICSME.2014.107.

APPENDIX A: FEATURE LEVEL TEST PLAN

FEATURE	Scope	Ma-ture	Ope-rat C	De-pendc	Sa-fety C	His-tory	Rel Freq	Upd Freq	3rd part dep	Custo-mized	TESTING MTD	Reg-res-sion	Depen-den-cies?	Histori-cal er-rors?	Is cri-tical?	Pirority value (0-3)
Start page	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Creating a new pro-ject	Functi-onality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Opening a project	Functi-onality	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Data im-port	Functi-onality	1	1	1	1	1	0	1	0	0	AUTO	Yes	1	1	1	3
Data ex-port	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Login user	Functi-onality	1	1	1	0	0	0	1	1	0	MAN	No	1	0	1	2
Feature switch	Functi-onality	1	1	1	1	1	0	1	0	0	AUTO	Yes	1	1	1	3
Logout user	Functi-onality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Model page	View	1	1	1	0	0		1	0	0	AUTO	Yes	1	0	1	2
Editing a model	Functi-onality	1	1	1	1	1	0	1	0	0	AUTO	Yes	1	1	1	3
Exporting a model	Functi-onality	1	1	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Selecting a model	Functi-onality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Adding agv operation	Functi-onality	1	0	1	1	0		1	0	0	AUTO	Yes	1	0	1	2
Editing agv operation	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2

Adding error	Functionality	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Editing error	Functionality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Editing energy setting	Functionality	1	0	0	1	1	0	1	0	0	AUTO	Yes	0	1	1	2
Editing settings	Functionality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Battery management	View	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Battery settings	Functionality	1	0	0	1	0	0	1	0	0	AUTO	Yes	0	0	1	1
Group select	Functionality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Battery table	Functionality	0	0	0	1	0	0	1	0	0	MAN	No	0	0	1	1
Blocking gropups	View	0	1	0	1	0	0	1	0	0	MAN	No	0	0	1	1
Data import	Functionality	0	1	1	1	0	0	1	0	0	MAN	No	1	0	1	2
Virtual io	View	0	1	0	0	0	0	1	0	0	MAN	No	0	0	1	1
Adding io	Functionality	0	0	1	1	0	0	1	0	0	MAN	No	1	0	1	2
Editing io	Functionality	0	0	1	1	0	0	1	0	0	MAN	No	1	0	1	2
Data import	Functionality	0	1	1	1	0	0	1	0	0	MAN	No	1	0	1	2
Data export	Functionality	0	1	0	1	0	0	1	0	0	MAN	No	0	0	1	1
Escape stations	View	0	1	1	1	0	0	1	0	0	MAN	No	1	0	1	2
Communication page	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1

Adding a host	Functionality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit host settings	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Add host operation	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Edit host operation	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Add host error	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Edit host error	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Add host transport unit type	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Edit host transport unit type	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Host addresses	View	1	1	0	0	1	0	1	0	0	AUTO	Yes	0	1	1	2
Data import	Functionality	1	1	1	1	1	0	1	0	0	AUTO	Yes	1	1	1	3
Data export	Functionality	1	1	0	0	1	0	1	0	0	AUTO	Yes	0	1	1	2
Data generate	Functionality	1	1	1	1	1	0	1	0	0	AUTO	Yes	1	1	1	3
Data delete	Functionality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
View rows	Functionality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Add row	Functionality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit row	Functionality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Apply filters	Functionality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0

Map inter-action	Functi-onality	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Table sca-ling	Functi-onality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Workflow manage-ment	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Open workflow	Functi-onality	1	1	0	0	0	0	1	1	0	MAN	No	0	0	1	1
Add work-flow	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Import workflow	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit work-flow	Functi-onality	1	0	1	1	1	0	1	1	0	MAN	No	1	1	1	3
Workflow rules	View	1	1	0	1	0	0	1	0	0	AUTO	Yes	0	0	1	1
Add host rule	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit host rule	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Add char-ging rule	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit char-ging rule	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Error rules	View	1	1	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Add ope-ration rule	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit opera-tion rule	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Add sys-tem rule	Functi-onality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit sys-tem rule	Functi-onality	1	0	1	1	1	0	1	0	0	AUTO	Yes	1	1	1	3
Internal addresses	View	1	1	0	1	0	0	1	0	0	AUTO	Yes	0	0	1	1

Data import	Functionality	1	1	1	1	1	0	1	0	0	AUTO	Yes	1	1	1	3
Data export	Functionality	1	1	0	0	1	0	1	0	0	AUTO	Yes	0	1	1	2
Data generate	Functionality	1	1	1	1	1	0	1	0	0	AUTO	Yes	1	1	1	3
Data delete	Functionality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
View rows	Functionality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Add row	Functionality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit row	Functionality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Apply filters	Functionality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Map interaction	Functionality	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Table scaling	Functionality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Bookable resources	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Add resource	Functionality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Edit resource	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Workflow data	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Add table	Functionality	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Delete table	Functionality	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Data import	Functionality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Data export	Functionality	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1

Dashboard settings	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Enable / disable	Functi-onality	1	1	1	0	1	0	1	0	0	AUTO	Yes	1	1	1	3
Edit set-tings	Functi-onality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Layout page	View	0	1	0	0	0	0	1	0	0	MAN	No	0	0	1	1
Data im-port	Functi-onality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Positio-ning cont-rols	Functi-onality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Import symbol	Functi-onality	1	0	1	0	1	0	1	0	0	AUTO	Yes	1	1	0	2
Add sym-bol	Functi-onality	0	0	0	0	0	0	1	0	0	MAN	No	0	0	0	0
Edit sym-bol	Functi-onality	0	0	1	0	0	0	1	0	0	MAN	No	1	0	0	1
Map inter-action	Functi-onality	1	1	0	0	1	0	1	0	0	AUTO	Yes	0	1	1	2
Mainte-nance or-ders	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Data im-port	Functi-onality	1	1	1	0	1	0	1	0	0	AUTO	Yes	1	1	1	3
Data ex-port	Functi-onality	1	1	0	0	1	0	1	0	0	AUTO	Yes	0	1	1	2
Data copy	Functi-onality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Data gene-rate	Functi-onality	1	1	1	0	1	0	1	0	0	AUTO	Yes	1	1	1	3
Data de-lete	Functi-onality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
View rows	Functi-onality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2

Add row	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Edit row	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Apply filters	Functionality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Map interaction	Functionality	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Add operation rule	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Edit operation rule	Functionality	1	0	1	0	0	0	1	0	0	AUTO	Yes	1	0	0	1
Add maintance order	Functionality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit maintance order	Functionality	1	0	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Events	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Add event	Functionality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit event	Functionality	1	1	1	0	0	0	1	0	0	AUTO	Yes	1	0	1	2
Edit event gropups	Functionality	1	0	0	0	0	0	1	0	0	AUTO	Yes	0	0	0	0
Export view	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Check configuration	Functionality	1	1	1	1	1	0	1	0	0	AUTO	Yes	1	1	1	3
Export configuration	Functionality	1	1	1	1	0	0	1	0	0	AUTO	Yes	1	0	1	2
Blackbox view	View	1	1	0	0	0	0	1	0	0	AUTO	Yes	0	0	1	1
Data import	Functionality	1	1	0	0	1	0	1	0	0	AUTO	Yes	0	1	1	2

Log view	Functi-onality	1	0	0	0	0	0	1	1	0	MAN	No	0	0	0	0
Video view	Functi-onality	1	0	0	0	0	0	1	1	0	MAN	No	0	0	0	0
Documen-tation view	View	1	1	0	1	0	0	1	0	0	AUTO	Yes	0	0	1	1