Tampere University

Jukka Murtonen

DESIGN AND IMPLEMENTATION OF HIGH-LEVEL AGILE TEST PLANNING FOR AN INDUSTRIAL AUTOMATION SYSTEM

Master of Science Thesis

Faculty of Engineering and Natural Sciences

Examiners: Professor Jose Martinez Lastra

University Instructor Luis Gonzalez Moctezuma

November 2021

ABSTRACT

Jukka Murtonen: Design and implementation of high-level Agile test planning for an industrial automation system
Master of Science Thesis
Tampere University
Master's Degree Programme in Automation Engineering
November 2021

Software is used in industrial automation systems in multiple aspects and scenarios. This software needs to be tested to prevent expensive and potentially dangerous failures and the testing needs to be planned for it to be successful. Studies point to a test plan being beneficial in consistently finding the issues in software testing and that risk analysis and prioritization of testing further emphasizes the benefits. However, in Agile development the use of a test plan is not usual and often the large picture of the project can be lost due to it, making it harder to tell when a project is truly finished and thoroughly tested. Developers have noted that often the documentation and specifications of projects are scattered in many different databanks within the company, making it challenging to find necessary information.

The thesis aims to develop a comprehensive framework for testing an industrial automation project, provide a way of defining the most critical features that need to be prioritized in testing, gather all the testing related information into one location and to develop a way to make certain the system is ready and thoroughly tested before dispatching it to the customer. The thesis consists of theoretical background, design and implementation chapters. The theoretical background explores levels of testing, Agile testing, Test Driven Methods, test automation, test plan, test planning and prioritizing tests. The design proposes a framework for forming a lightweight test plan, including risk analysis and test prioritization and the implementation utilizes the framework in practise.

The thesis presents a lightweight test plan template for Agile development projects developing industrial automation systems. The template is based on earlier studies found in literature and standards for software testing. The plan gathers all the necessary documentation in one place, provides a way of defining the most critical features that need to be prioritized in testing by way of risk analysis and different indicators used in literature, and makes it easier to make certain the industrial automation system is complete and thoroughly tested before delivery to the customer.

Keywords: agile, software testing, test plan, test prioritization

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Jukka Murtonen: Ketterän testaamisen korkean tason suunnittelu ja toteutus teolliselle automaatiojärjestelmälle
Diplomityö
Tampereen yliopisto
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Marraskuu 2021

Tietokoneohjelmistoja käytetään teollisissa automaatiojärjestelmissä useilla tavoilla erilaisissa tilanteissa. Nämä ohjelmat tulee testata, jotta voidaan välttyä kalliilta ja potentiaalisesti vaarallisilta häiriöiltä, ja testaamista tulee suunnitella, jotta testaus onnistuisi. Tutkimuksien perusteella testaussuunnitelma on hyvä vaihtoehto ongelmien johdonmukaiseen löytämiseen ohjelmistotestauksessa ja riskien analysointi ja testauksen priorisointi edelleen korostaa hyviä vaikutuksia. Ketterässä kehityksessä testaussuunnitelman käyttö ei kuitenkaan ole tavallista ja siksi usein projektin laaja kuva voi unohtua, mikä vaikeuttaa varmistumista siitä, että projekti on todella valmis ja läpikotaisin testattu. Ohjelmistokehittäjät ovat huomanneet, että projekteissa dokumentit ja spesifikaatiot ovat usein hajallaan eri paikoissa yrityksen tietopankeissa, mikä vaikeuttaa oikean tiedon löytämistä.

Diplomityön tavoitteena on kehittää kattava viitekehys teollisen automaatioprojektin testaamiseen, tarjota tapa kriittisimpien ominaisuuksien määrittelemiseen testauksen priorisointia varten, kerätä kaikki testaukseen liittyvä tieto yhteen paikkaan ja tarjota tapa varmistaa, että järjestelmä on valmis ja hyvin testattu ennen sen luovuttamista asiakkaalle. Työ koostuu teoreettisesta taustaosuudesta, suunnittelu- ja toteutusosuudesta. Teoreettinen taustaosuus tutkii testauksen eri tasoja, ketterää testausta, testivetoisiamenetelmiä, testiautomaatiota, testisuunnitelmaa, testisuunnittelua ja testien priorisointia. Suunnitteluosuus ehdottaa viitekehystä kevyen testisuunnitelman luomiseen, mukaan lukien riskien analysointi ja testien priorisointi, ja implementaatio-osuus käyttää kehystä käytännössä.

Työ esittää kevyen testisuunnitelman viitekehyksen ketteränkehityksen projekteille. Viitekehys perustuu edellisissä tutkimuksissa kehitettyihin menetelmiin ja ohjelmistotestaukseen liittyviin standardeihin. Viitekehyksen avulla kaikki testauksen liittyvä dokumentaatio saadaan kerättyä yhteen paikkaan, kriittisimmät ominaisuudet saadaan määritettyä testauksen priorisointia varten riskianalyysin ja eri indikaattoreiden avulla sekä helpottaa varmistumista siitä, että teollinen automaatiojärjestelmä on valmis ja kattavasti testattu ennen asiakkaalle toimittamista.

Avainsanat: ketteräkehitys, ohjelmistotestaus, testisuunnitelma, testien priorisointi

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

PREFACE

Five years of studies in Tampere University and abroad have culminated in this thesis. There have been many bumps in the road to this point, but with a few patches to the tires along the way, here we are finally. The edge of studying and working.

I would like to thank Fastems for providing me the opportunity to work at their company on a thesis with an interesting topic. I want to thank Antti Mertsola for providing me with guidance on and support with the many challenges included in testing and quality assurance, the whole SW QA team and Kuha development team for insights into Agile development, Luis Gonzalez Moctezuma and Jose Martinez Lastra for precise mentoring in the form and necessities of a Master's Thesis and Anni Mattila for pushing me forward every day and encouraging me to keep at it.

I think I'll have some cognac now.

Pirkkala, 26th of November 2021

Jukka Murtonen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AAT	Agile Acceptance Testing
ANSI	American National Standards Institute
APFD	Average percentage of fault detection
ATDD	Acceptance Test-Driven Development
ATP	Acceptance Test Plan
BDD	Behavior Driven Development
CP	Customer Priority
DO	Delivery Owner
EDD	Example Driven Development
ESA	European Space Agency
FICA	Federal Insurance Contributions Act
FMEA	Failure Modes and Effects Analysis
FP	Fault Proneness
GTS	Gantry Tool Storage
GUI	Graphical User Interface
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
LTP	Level Test Plan
MLS	Multi-Level System
MMS	Manufacturing Management Software
MPM	Manufacturing Project Management
MTP	Master Test Plan
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technology
PLC	Power-line Communication
PLM	Product Lifecycle Management
PORT	Prioritization of Requirements for Test
RIBC	Risk Impact based on Business Criticality
ROI	Return-On-Investment
RPN	Risk Priority Number
SBE	Specification by Example
ST	Story Testing
SUT	System Under Test
SWQA	Software Quality Assurance
TSC	Tool Service Cell
USA	United States of America
USD	United States dollar
UTP	Unit Test Plan

1. INTRODUCTION

Test planning is important for the efficiency, quality, and profitability of industrial automation projects. Test Plans have an important role in making the testing efficient in detecting defects in important features of the software development projects. In this chapter the background of the research and its motivations are explained and the limitations, research questions and the outline of the research are stated.

1.1 Background

Software is of growing importance in every aspect of today's life. This holds true for industrial automation systems as well. These systems can plan the schedule of a product from raw material to a readymade and packaged product ready to be shipped. On this journey the product can be shifted with robots, cranes, and automated guided vehicles using software to design its path, where raw materials are stored and where "Work in progress" parts are stored in the system. Also using software, the machining and processing of a product is done in huge industrial lathes, washing machines, deburring robots and other machining tools by controlling them automatically by the ways of Computerized Numerical Control. In such huge systems with many moving parts, changing factors to consider, and expensive and dangerous machinery moving, the cost of failure can be huge as well, so testing to find those defects beforehand can be worthwhile.

A study by Sorqvist showcases the cost of failures. In the study 30 Swedish medium to large companies were followed for three years and the results indicated 9 to 16 percent losses from turnover because of poor quality. (Sorqvist 1998) Another study by the NIST, short for National Institute of Standards and Technology, was conducted to find out the effects of poor planning in software testing in the United States. The study tells of huge failures that happened because of poor software testing, such as the Titan III-Centaur rocket failure at launch that cost the US Air Force 1200 million dollars in 1999. But the study points out that the everyday, routine like losses, because of inadequate quality are much bigger than the example extremity. According to the study, the combined cost of poor software testing because of inadequate planning in the United States is about 60 billion dollars per year. What we can see here is that effect of everyday losses due to inadequate software testing are much bigger than the huge defects combined. (NIST 2002) A comprehensive framework for testing could help plan the testing and catch the causes of bad quality, whether they are big or small. To prevent the big, devastating, and dangerous defects from rising, a method for prioritizing the testing of potentially dangerous and system critical features would be beneficial as well, to make sure the risks are taken into consideration.

Testing is a collection of tasks that are planned and systematically undertaken before product is given to the end customer for its actual purpose. In the IEEE Standard 829–1998 for Software Test Documentation (IEEE 1998), the Test Plan is defined as “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.” (IEEE 1998) The importance of a test plan rises exponentially as the project gets more complex (Craig, Jaskiel 2002).

Test planning is a process which creates the Test Plan that allows all participants of the testing process to communicate with each other clearly and decide what are the most important issues and how to deal with them. Ultimately the goal of test planning is not to create a long document to sit on the self, but to deal with the aspects of testing, such as testing strategy, risks, priorities, responsibilities, and resources of the project. The process builds an understanding of what is going to be tested, why and how it is going to be tested and this understanding early in the development life cycle can make the testing run more smoothly and can save a lot of money and time. (Craig, Jaskiel 2002, Patton 2005)

Software quality consists of following the clearly stated performance and functional requirements, development standards that are clearly written down and unspoken expectations of characteristics that go with all professional software. In other words, requirements are the cornerstone for software quality and without following the requirements, the quality is lacking. Also, standards offer criteria to follow through the development process and offer guides for software engineering, and if the standards are not followed the quality will most likely be lacking. (Ahmed 2010) To help keep those points in mind during software projects, the requirements and standards could be gathered into one place, where the requirements can be easily updated and followed accordingly in the development of software projects. Often documentation can be scattered in various documents and places among the different libraries of the company, and the test plan could be used to gather the testing related information into one place, so that it would be easier to find the data about testing when it is needed.

In Agile development it is sometimes said that there is no planning. A big reason for developers to utilize Agile methods is that they have tried traditional planning and noticed that it does not work very well for them. In most businesses the situation can change rapidly and often, making the already developed big plans obsolete. In Agile development there are no big plans, but it is useful still to have some sort of idea about what the customers' needs are and how the development should get started. (Crispin, Gregory 2009) A lightweight planning framework that also works in Agile would then be beneficial, as the benefits of planning would be good to integrate to the Agile method as well. An updating plan to monitor the development and testing could help give an overview of the development situation with the whole project.

According to a Growth Statistics Report done by Global Market Insights, Software Testing Market size was 40 billion USD, United States dollar, in 2019 and its Compound annual growth rate was forecasted to be 6 percent from 2020 to 2026. Value projection for 2026 was 60 billion USD. Drivers for growth were listed as; growing adoption of Artificial Intelligence and Machine Learning in software testing; increasing agile testing; growing digitalization in developing economies; increasing adoption of DevOps; growing consumption of mobile based applications. (GMI 2019) So in the testing industry, ready-made methods for developing plans for testing, even in Agile developments, can pose a big opportunity for growth and differentiation in the large market.

In the United States of America, about 30 percent of the population in 2010 used computers, and there for software daily, for purposes of business or pleasure. That is about 93 million people. In 2017 from 85 to 88 percent of Europeans had mobile subscriptions and 70 percent of those had adopted a smartphone, so software through embedded solutions reaches quite a lot of people, not to mention other software solutions, such as digital cameras, digital watches, cars, home appliances, and entertainment devices. Further on, 65 percent of companies in USA in 2010 used computers and software from spreadsheets to ERP. (Jones, Bonsignour 2011, GSMA 2018) Today this number must be even higher in the ever-digitalizing world, and so the importance of proper testing and test planning has an increasingly crucial role in the industry to avoid losses that are quantified by the many uses of software.

There are countless possibilities for defects to surface through the whole process of software development. From when its initiated, planned, coded, tested and when the product is in the hands of the end user. Figure 1. illustrates how the cost of a bug can rise according to when it is discovered. (Patton 2005)

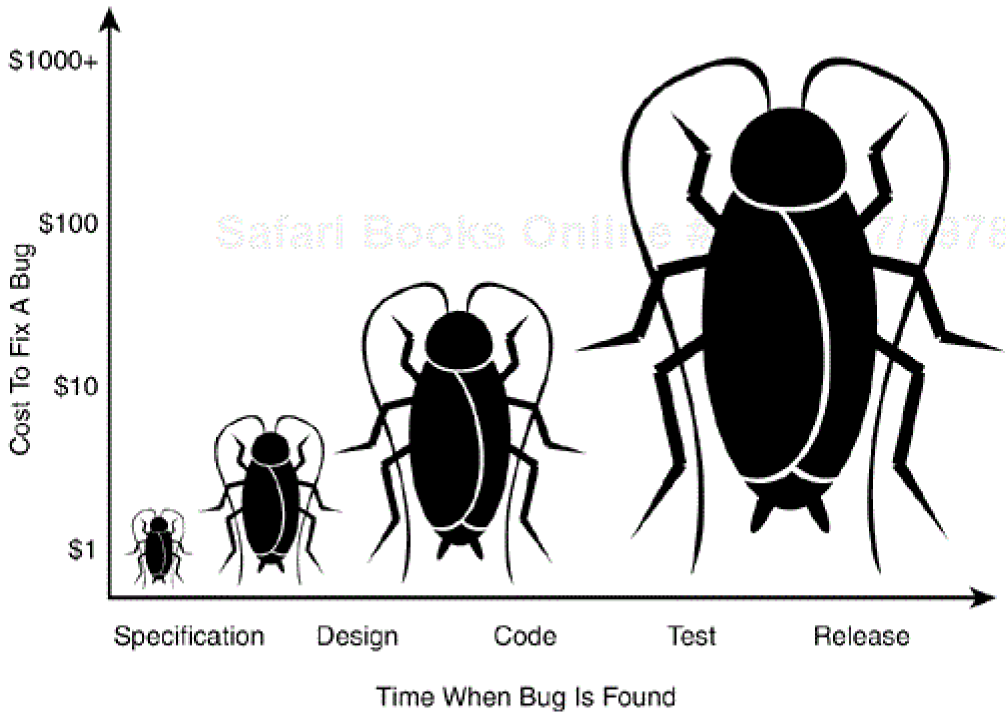


Figure 1. *Cost of a bug (Patton 2005).*

The bug cost crow logarithmically, meaning they increase with a factor of ten as time goes by. For example, a bug detected in the early stages, maybe when the specification is being written, can cost pennies. If the same defect is found during the testing of an already coded product, it can cost from tens to hundreds of dollars and if the customer finds it in a finished product, the bug can cost thousands or possibly millions. (Patton 2005) So it is important to be certain the system is completed and thoroughly tested before dispatching it to the customer and that the most critical features are tested, to avoid the larger costs of bugs detected later in the project.

1.2 Research questions

The goal of this thesis is to develop a comprehensive framework for testing an industrial automation project, provide a way of defining the most critical features that need to be prioritized in testing, gather all the testing related information into one location and to develop a way to make certain the system is ready and thoroughly tested before dispatching it to the customer. The research question formed from the previous are the following:

1. How a comprehensive framework for Agile testing can be designed and implemented for an industrial automation system?

2. How to define the most critical features that need to be prioritized in testing of an industrial automation system?
3. How to be certain the industrial automation system is complete and thoroughly tested for dispatching to the customer?

1.3 Limitations

The thesis is limited to a single project, that is developed in an Agile mentality and with an Acceptance Test-Driven Development method, to which the implementation was done, and to the higher-level planning of the testing, leaving the actual testing, test design and lower-level test plans out of the scope of this thesis. Decisions on test automation were also left out of the scope.

1.4 Outline

This thesis follows the following outlines; Chapter 2 discusses testing in general and establishes the current situation in software testing by discussing different levels of testing, Agile testing, the Test-Driven Methods, test automation, test plans, test planning, and test prioritization; In Chapter 3 a proposed framework for forming a simplified test plan is introduced; In Chapter 4 the implementation of the framework proposed in Chapter 3 is described; Chapter 5 discusses the results, concludes the thesis and ends in discussion on the results and possible future research projects on the subject.

2. STATE OF THE ART

Testing is a collection of tasks that are planned and systematically undertaken before the industrial automation system, or product, is given to the end consumer for its actual purpose and it is an important dimension of the software development life cycle. Programs are executed with specific inputs and the outputs are then measured and evaluated accordingly. If the output matches what is expected to come out, the program works as it should, and if the output is something else, something is wrong or broken somewhere in the program. (Singh 2012)

Testing is a task that is performed on a component or a system under determined circumstances, from which the outcomes are evaluated and documented and the workings of the component or system under test are evaluated (IEEE 2008). Testing should be undertaken with a specific goal in the sights, called the test objective, and the goal should be documented in specific, quantifiable expressions, such as test coverage or effectiveness (Syed, Kumar 2011).

Testing is meant to discover errors, defects, and failures. They may appear in any point of the software life cycle, as illustrated in Figure 2. Errors are defined as an erroneous result from a human action. Defects are problems or imperfections that can cause the software to give erroneous results, fail completely, or fail to complete the desired functions. Failures are the cessation of the software's ability to execute a specified function. (Laporte, April 2017)

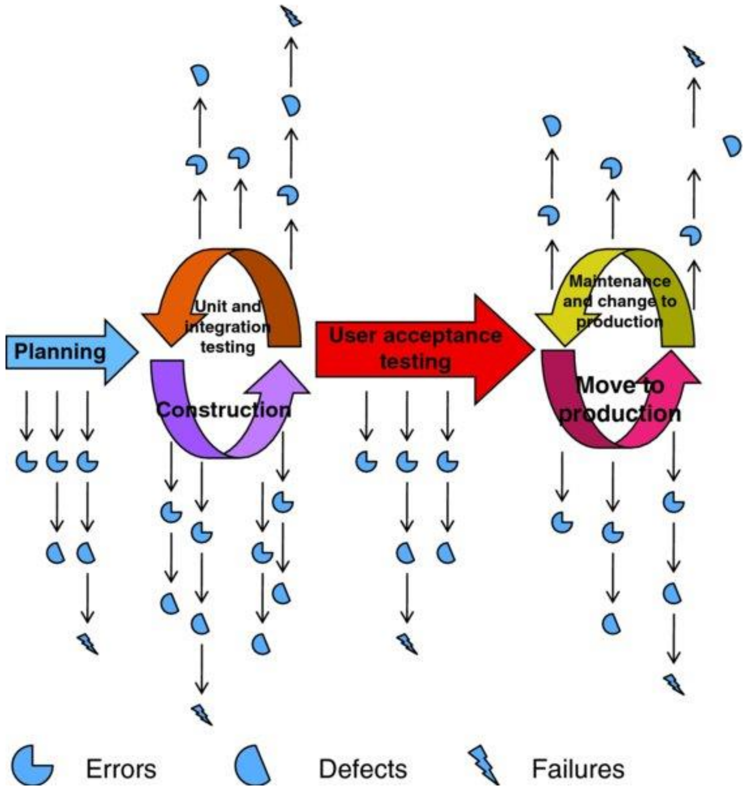


Figure 2. *Software life cycle with errors, defects, and failures (Laporte, April 2017, Galin 2018).*

There are numerous principles for testing software that need to be understood before planning effective and useful test cases. The following are a few examples adapted from The Art of Software Testing by Glenford Myers (Myers, Badgett et al. 2012) and Software Engineering and Testing by Agarwal, Tayal and Gupta (Agarwal, Gupta et al. 2009):

- All tests must be traceable to client's requirements.
- The test should be planned in advance, preferably as soon as the requirements are clear.
- The Pareto principle fits software testing in that 20 percent of software features are the source of 80 percent of the defects found. The probability of finding errors in a feature correlates with the number that have already been discovered in the feature.
- Testing the software should begin from small, individual components and end in the large, entirety of the industrial automation system.
- Testers should also realize that testing everything is not possible due to so many possible paths of execution and permutations.

- There should be testcases for valid and invalid inputs and outputs that are expected and unexpected.
- A tester should not only check if a program does what it is supposed to, but also check if it does what it is not supposed to.
- Lastly testing should not be performed by the developer of the software, but a third party. A programmer should not test their own program.

2.1 Levels of testing

Test planning should consist of several phases or levels, the highest level being the Master Test Plan (MTP), which can be a document or be a part of the project plan. The Master Test Plan is there to organize the other testing levels, as illustrated in Figure 3. The Master Test Plan should be general and offer rules and requirements for the whole project. The purpose of a Master Test Plan is to act as a managing document for schedules, resources, defining levels for testing, defining necessary tasks, provide objectives for the testing and for the underlying levels, but also to document risks and presumptions, identify how the testing is managed, to confirm the quality assurance goals and to define the documents to come out of the planning. (IEEE 2008)

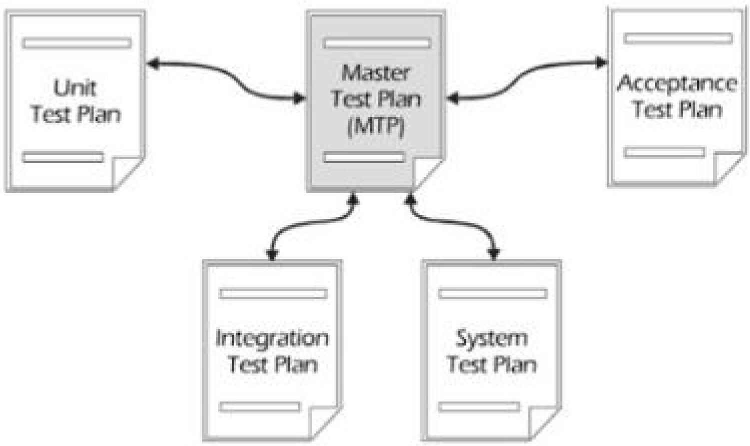


Figure 3. *Levels of Test Plans (Craig, Jaskiel 2002).*

According to the IEEE Std. 829-1998 standard, other levels can be Unit, Integration, System, and Acceptance (IEEE 1998), but more levels are used as well, such as beta, alpha, customer and user acceptance, string, build and development. These other levels should have detailed test plans of their own, like Acceptance Test Plan or System Test Plan. When designing these Level Test Plans, it is usually worthwhile to have them follow the same principles as the Master Test Plan to avoid confusion. (Craig, Jaskiel 2002)

In large scale projects it is often important to have many of these Level Test Plans, but in smaller scale undertakings, companies can suffice with just one test plan for the whole project. The decision of the scopes of test plans can be one of the first decisions in the test planning process. (Craig, Jaskiel 2002)

There are three main levels for testing: Unit, integration and system testing as illustrated in Figure 4.

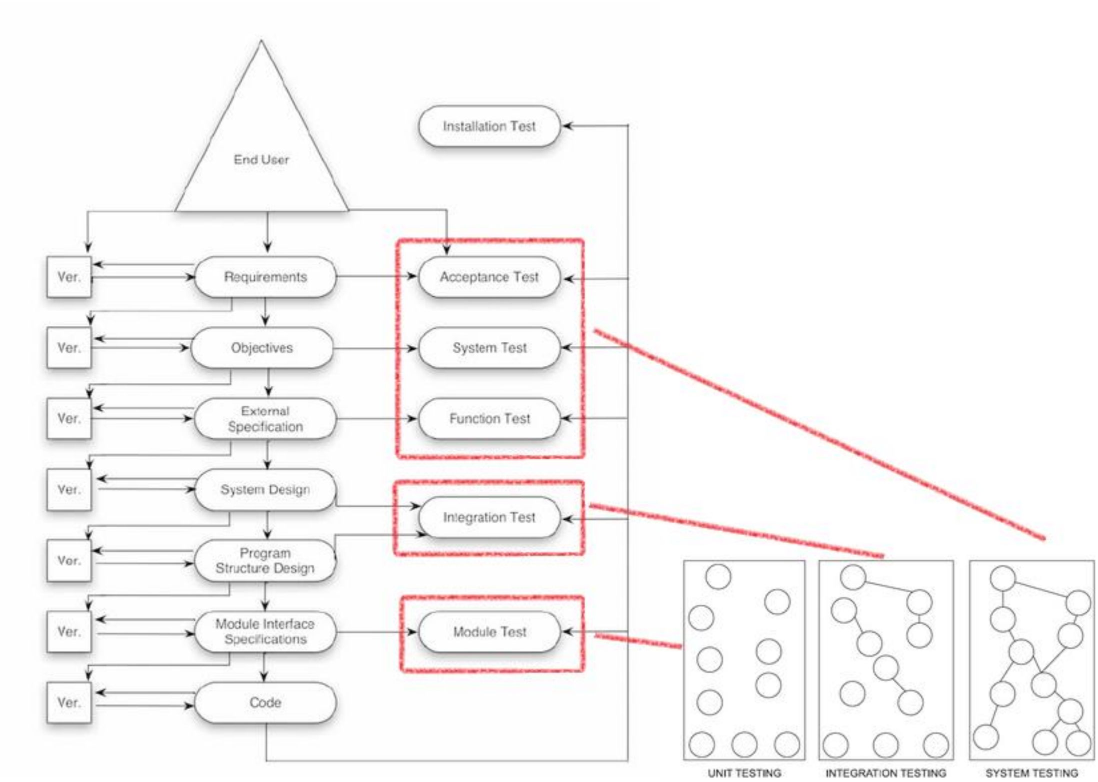


Figure 4. *Testing levels in development process. (Agarwal, Gupta et al. 2009, Myers, Badgett et al. 2012, Chopra 2018).*

First level of testing is unit testing. Unit testing is concerned with individual parts of the industrial automation system and its purpose is to make sure these parts function as they should. Each component is tested individually, without other parts influencing it. Testing components individually can help eliminate situations that produce multiple errors that could be confusing to diffuse. Unit testing can also be beneficial in that the code chunks can be small enough that the errors can be easily located, and the code blocks can be tested in a manner that is exhaustive of all possible errors. Common errors to be discovered during unit testing include data type comparisons of different types, erroneous logical operators, incorrect variable comparisons, failure to exit loops and erroneous loop variables. (Agarwal, Gupta et al. 2009, Chopra 2018)

Second testing level is integration testing. Integration testing is concerned with the construction of the program structure in a systematic way while simultaneously checking for errors in the interfacing. In integration testing different modules are integrated according to a plan. The main concern is to check the interfaces of the modules for defects that occur with modules invoking others with passing parameters. The integrated modules are planned in an integration plan which details the steps and the integration order of the modules. Some common approaches for integration testing are Regression Testing and Smoke Testing. (Agarwal, Gupta et al. 2009, Chopra 2018)

Regression Testing is used in assuring that already implemented functions do not break with new components being added or that the changes do not introduce new unwanted functionality to the system. Regression testing has three different types of test cases; Supplementary test for software functions; Cases that test all software functionality; Cases that focus on changes to the components. (Agarwal, Gupta et al. 2009)

Smoke Testing is a rolling integration approach because each time the software is rebuild with new functions new test are included. Smoke Testing contains testing software components that have been integrated into a new build. A build encompasses all required components, such as data files, modules, libraries, and coded components, that are needed to execute the functions that are under test. Smoke Testing can be a series of tests planned to find defects that keep the build from carrying out its proper functions. Smoke Testing can also mean testing the integration of two different builds. When the two builds are integrated the product is tested in its current form daily. Other builds can then be integrated to the testing build, and so the integration progresses. Smoke Testing offers benefits, such as minimizing the risks posed by integration, improvements to product quality, ease of diagnosing and fixing errors, and the ease of following the progress of the development. (Agarwal, Gupta et al. 2009)

After Integration Testing comes the System Testing phase, where the integrated builds have formed the whole industrial automation system. The goal of System Testing is to weigh up the SUT, meaning System Under Test, against the specified objectives. System Testing is not restricted to systems but can be done to programs as well, trying to show how the program does not meet its objective criteria. System Testing is not possible if there are no written documentation of the SUT and quantifiable goals for it. Much like Integration Testing, System Testing tries to expose errors resulting from the interacting, integrated system components. In addition to testing the components, System Testing also validates that the product meets its requirements, functional and non-functional. The three main types of System Testing are Alpha Testing, Beta Testing and Acceptance

Testing, but there is also Facility Testing, also known as Function Testing, that is worthwhile to mention. (Agarwal, Gupta et al. 2009, Myers, Badgett et al. 2012, Chopra 2018)

Alpha Testing points to the testing done by the testers in the company producing the software. The alpha tests are done in a controlled environment under the guidance of the developing organizations testers by the customer. The customers test the product in the development environment, simulating a real-life situation, and look for errors that need to be fixed. However due to the environment being controlled and simulated, it has a narrowed ability to detect errors for correction. (Agarwal, Gupta et al. 2009)

Beta Testing is System Testing done by a chosen team of friendly customers. Beta Testing is done at the customers site with multiple users simultaneously using the test environment in a testing mode, looking for mistakes, errors, and other considerations to be recorded and to be given to the developers for evaluation. The developers can be present for the testing, but they do not have to be, so the customers can test the product without supervisors and produce feedback. Then newer version with the changes applied will be in next version releases. (Agarwal, Gupta et al. 2009, Chopra 2018)

As seen in Figure 4, Function Testing is a type of System Testing that compares the industrial automation system to the specification and tries to find differences, or errors, in the system. If the specification does not match the system, there are defects in the product that must be fixed. The external specification is an exact representation of the systems desired functionalities according to the client. Function testing is usually done as Black-box testing, save when using it on tiny projects. Black-box testing meaning that the system is understood as a black box and only the inputs and outputs are monitored, without any of the functions inside being considered (Sawant, Abhijit et al. 2012). Function Testing is possible to be done in a Black-box-manner, as in the module, or unit, testing phase, the logics have been covered in the wanted White-box-manner and its criteria have been met. White-box meaning, as opposed to Black-box, that the inside workings and functions are looked upon and the way the system processes inputs and creates outputs is looked at and analysed in a logical way (Sawant, Abhijit et al. 2012). In Function Testing, test cases are made according to the specification. One important thing to keep in mind in Function Testing is The Pareto principle. The functions in which most errors were found are also most likely to contain the majority of the defects that are not found yet. Another key point to keep in mind is input forms and conditions. But the most important thing to keep in mind is that the point of Function Testing is to find defects and differences to the specification and not to show that the system corresponds to the specification. (Myers, Badgett et al. 2012, Baresi, Pezzè 2006, Chopra 2018)

The last type of System Testing is Acceptance Testing, in which the customers test that a software project meets its criteria for acceptance and to let the client see if they should or should not accept the product as it is (IEEE 2008, Craig, Jaskiel 2002). The tests are conducted so that the customer can be certain of the quality of the product, and that the product matches the requirements and specifications. An Acceptance Testing scenario can range from a test drive done by the customer to a full planned set of tests. Acceptance testing can take up to months of time to conduct. As seen in Figure 4, Acceptance Testing is comparing the system to the written requirements and comparing the system to the clients' desires. It is an interesting type of testing as it is most times performed by the customer and not the developer, and so is not deemed as the burden of the producer. Usually, the client compares the ready software to the terms agreed upon when the software was ordered. (Agarwal, Gupta et al. 2009, Myers, Badgett et al. 2012, Chopra 2018)

The best way of planning test cases for Acceptance Testing is to plan them to show that the software does not meet the requirements. If the test cases fail, the software is accepted by the client. Most times the smart clients perform Acceptance Tests to make certain the product fulfils its needs as well. Even though the Acceptance Testing is the clients' burden, the producer should also do User and Usability Testing to make certain the product is good before shipping to the client. User and Usability Testing mean testing from the point of view of the customer, for example, How the User Interface looks; How fast is the program; Any UI glitches; Are there clear guides; does the interface accept weird inputs? (Agarwal, Gupta et al. 2009, Myers, Badgett et al. 2012, Chopra 2018)

Normally Acceptance Testing at delivery is the last opportunity for the buyer to check the software and address the flaws to get them fixed by the developer and regularly it is the only period when the client can point out insufficiencies in the product. (Perry 2006)

Decisions about acceptance happen at times that are planned and scheduled for when tools, documentations, software components, processes and lastly the whole system needs to meet the acceptance criteria. The acceptance criteria can be divided in to four categories (Perry 2006):

- Functionality, meaning code and document consistency, functional traceability, logic verification, functional testing, and testing the functionality in the operational environment.
- Performance, meaning viability of performance requirements, simulation tool performance, and testing the performance in the operational environment.

- Quality of Interface, meaning documentation concerning interface, test plans for integration and interface, complexity and ergonomics of the interface, and testing the interface in the operational environment.
- Overall quality of the system, meaning measurability of quality measures, acceptance criteria, documentation and standards, and operational testing quality criteria.

2.2 Agile Testing

Agile testing is described in the Manifesto for Agile Software Development by four sentences: Individuals and interactions over processes and tools; Working software over comprehensive documentation; Customer collaboration over contract negotiation; and responding to change over following a plan.(Beck 2001)

From 2001 to 2018 the manifestos guidelines have become industry standard in software developments quickly developing field. “Agile” as a word is somewhat impaired in that it is used in so many different instances to mean everything from industry standards to methods adapted in practice in Agile software development and testing. Agile testing is software testing in an Agile environment, meaning test automation, testing manually, error reporting, and behaviour documentation, but it can also mean methods of development. Some popular development methods are Example Driven Development, or EDD, Behaviour Driven Development, or BDD, Acceptance Test-Driven Development, or ATDD, Agile Acceptance Testing, or AAT, Story Testing, or ST, and Specification by Example, or SBE. (Adzic 2011)

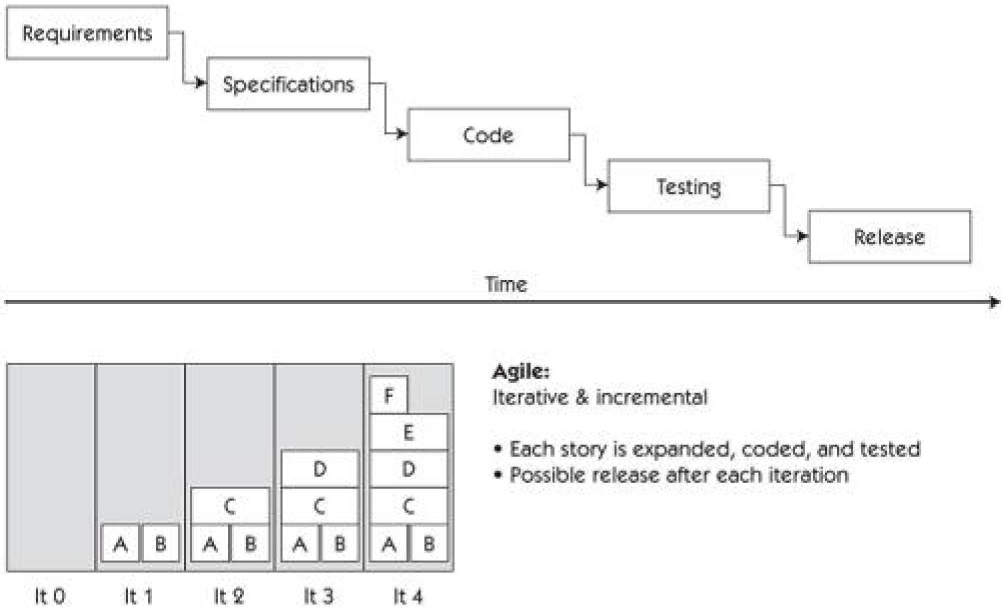


Figure 5. *Plan- Driven and Agile Development (Crispin, Gregory 2009).*

Lisa Crispin talks about Agile testing meaning pursuing the best product possible for delivery in her book Agile Testing. In the book Agile testing is divided into four quadrants, and the differences to the traditional waterfall model are brought to the foreground by the short iterative nature of Agile testing. As illustrated in Figure 5, In the waterfall model testing is done at the end of sequential development steps while in the Agile method the development and the testing are done in one-to-four-week cycles. In Agile Testing the testing is started when new features are introduced. Designing and planning the testing is done in cycles as well while the project moves forward but documenting is usually done very sparingly or not at all. In the cycles of Agile Development, features are developed, coded, and tested, after which the feature is deemed finished. (Crispin, Gregory 2009)

2.2.1 Agile Testing Quadrants

Lisa Crispin established the four Agile Testing Quadrants as guides on approaching Agile Software Testing from distinct directions. The Agile Testing Quadrants offer different views for showing from which directions the software is tested and how it is tested. The four quadrants of Agile testing are, from first to last, Automated, Automated and Manual, Manual, and Tools. The quadrants address aspects of software quality, namely Business, Team, Technology and Product. The quadrants and aspects of quality are shown in Figure 6.

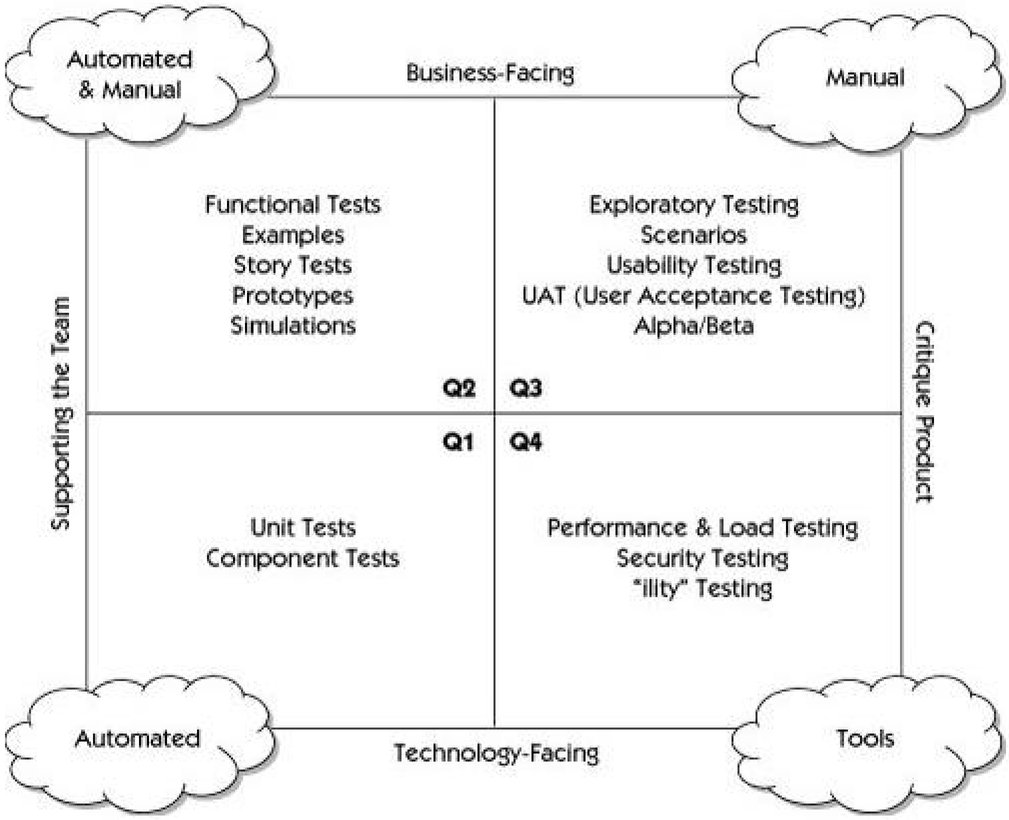


Figure 6. *Agile Testing Quadrants (Crispin, Gregory 2009).*

The first quadrant, Automated testing, holds Test Driven Development or TDD. The developers use a TDD method such as Behaviour Driven Development, BDD. Automated unit and component tests are developed simultaneously with the units and the components. The developed tests then check that the software works continuously while also working as automated regression tests for when new features and functions are introduced to the system. The tests for units and components are usually developed in the same language as the software, which then requires technical know-how to be able to understand the tests. The simultaneous development of automated unit tests makes the developers to consider the architecture of the code and to design the code so that it will be easy to test automatically. (Crispin, Gregory 2009)

The second quadrant, Automated and Manual testing, contains examples, functional testing and story testing, simulations, and prototypes, which function on a higher level than unit testing, while simultaneously also driving development. The testing in the second quarter is concerned with the business and customer aspects, defining the level of quality required and the needed features, and showing and making sure the system works as required. The tests work on a functional level and can be written and understood without technical knowledge. (Crispin, Gregory 2009)

The third quadrant, Manual Testing, contains exploratory, user acceptance, scenario, usability, Alpha and Beta testing. These tests are business facing and test the software to see if it meets the requirements set. Test automation can be useful in creating test data for the manual testing effort, but imitating a possible user at work requires a human. At the heart of this quadrant is Exploratory Testing, in which the testers design and conduct the testing with a critical approach to look at the outcomes. Exploratory testing is designed with a strategy and follows this defined strategy in the testing. The outcome of manual testing depends on the know-how of the tester, as the testing requires understanding of the software, intuition, and testing experience. Subjective measures for quality like usability or visual defects are considered in this quadrant. (Crispin, Gregory 2009)

The last and fourth quadrant, testing tools, contains testing that is connected to technology. Such attributes as robustness, performance and security are considered in this quadrant. Tests in this quadrant are dependent and influenced on the tools, technologies and the design used. These tests are there to make sure that technical attributes that cannot be tested through functional requirements end up being tested and that non-functional requirements are met. The tests aim for finding insufficiencies from a technical view. Many of the attributes tested in quadrant four are often deemed low of risk and excluded from the test plan. (Crispin, Gregory 2009)

2.3 Test Driven Methods

Test-Driven Development, TDD, Behaviour-Driven Development, BDD and Acceptance Test-Driven Development, ATDD, are all methods of Agile Development. The methods utilize the Shift Left principle, which aims to introduce testing as a part of development and move testing up from the end of the development life cycle. In these methods the actual tests at the beginning of a software development lifecycle form a basis for the development of the software. The feedback loop is shorter as a newly finished functionality gets tested right away against the formerly developed tests, (Moe 2019)

2.3.1 Test-Driven Development

Test Driven Development, which can also be called Unit Test Driven Development is founded on short cycles of developing the unit tests, and then the actual code based on and designed to pass the written tests. The functionalities being added to the software implementation will not be accepted until the functionality passes all the tests designed for it. And so, TDD is not a testing technique but a development and design methodology.

In TDD, the software architecture and higher-level designs are completed before development of functionalities for lower-level unit testing. This higher-level planning phase does not account for the implementation on a detailed level. Due to the incremental and iterative process of TDD the lower-level structures of the software are formed spontaneously during the software development. (Latorre 2014)

One cycle of the method is constructed from the seven steps shown in figure 7. (Latorre 2014):

1) Writing the unit test, but not yet coding.
2) Performing the written test to see that there is no implementation for the function yet.
3) The functionality is developed to pass the test.
4) The test and all previous tests concerning the functionality are performed and if one of the tests fails then step three is performed again.
5) The code if refactored so that it is less complex and it is simpler to understand and maintain.
6) All the test cases are performed again and if any one of them fails step 5 is performed again to fix the errors.
7) The functionality under development is deemed finished and development is sifted on to the next functionality.

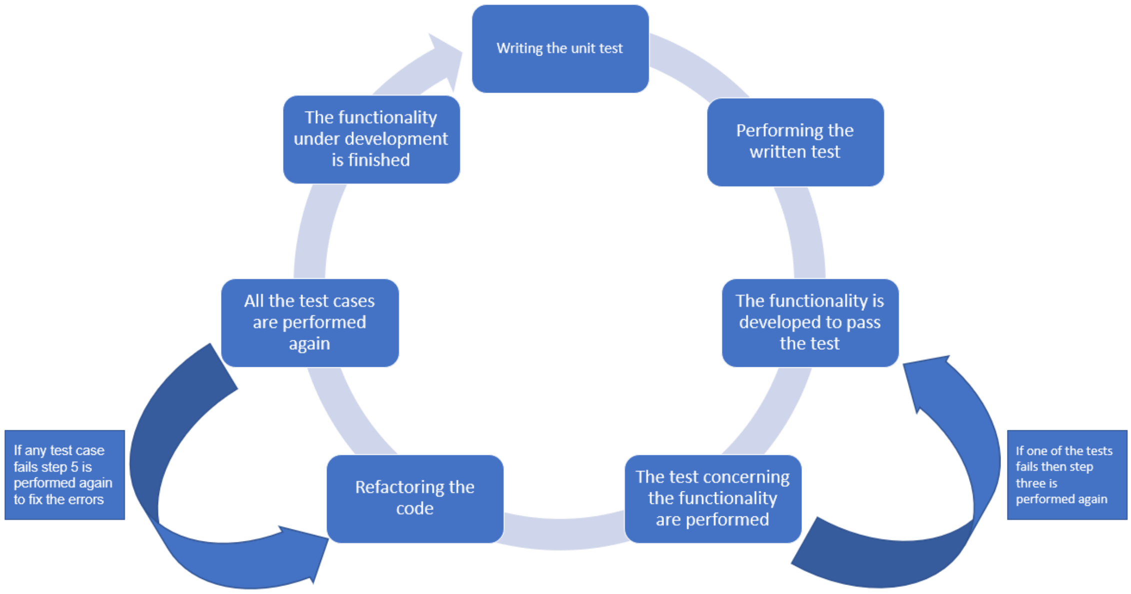


Figure 7. *TDD-cycle.*

It has been noted that TDD improves the quality of software but simultaneously decreases productivity. The decrease of productivity has been linked to the time demanded by the developing of automatized unit tests. But although the productivity is decreased, the time used to fix errors is also decreased due to better quality of code. (Latorre 2014)

2.3.2 Behaviour-Driven Development

BDD or Behaviour-Driven Development focuses on defining the wanted software behaviour and then developing the functionality so that it meets the wanted behaviour. The functionalities are developed in such a way that they meet the minimal requirements of marketability, but also so that they derive the greatest possible benefits. BDD creates a unified language between financing employees and the developers so that misunderstandings and exes code are decreased. (Ye 2013)

BDD is based on TDD, but as opposed to TDD, in BDD the functionalities are written into a user story, which describes how the functionality should work. The user stories should be easy to understand for the financial side of the company. Tests are written and run based on the user story. The tests are ensured to be proper before the functionality is developed, similarly to TDD. (Ye 2013)

2.3.3 Acceptance Test-Driven Development

ATDD or Acceptance Test-Driven Development consists of the testers, developers and the customers working together to generate requirement descriptions for producing software that is testable and faster to develop. In ATDD functionalities must go through acceptance tests before they are added to the main project. ATDD can be used with TDD so that the acceptance tests are the basis of the development. The point of ATDD is to create a simple description of the customers' requirements, and from the requirements develop tests for high-level-functionalities before any actual coding. Involved in creating these test cases are the triad, or the customer, the developer, and the tester. In the role of the customer can be anyone who can reliably convey the customers' requirements. The tests are written in a concise language that everyone in the triad can understand, to steer clear of miscommunications, save time on understanding, and to decrease time pressure. (Gärtner, Gèartner 2013, Latorre 2014) The easily understandable common language used between the triad can make upkeeping of the system easier, if the same concise language is used on the implementation level, the connection between the requirements and the code can be easily seen (Pugh 2011).

Acceptance tests differ from lower-level tests, like unit test, in that they do not change with the development of the software but change if the requirements are altered. The projects acceptance tests should cover all the functional requirements of the system and there should be new acceptance tests developed according to new requirements introduced. When the tests are created before the actual implementation of code, and the code must pass the tests it is based on, the tests cases can function as a description of the functionalities of the system. In other words, the tests are an executable specification which does not expire if the requirements do not change. (Pugh 2011)

If all the features of the system have test cases written of them, then the whole system gets tested, and the test coverage can be understood as being very comprehensive. Also, when all partners of the triad are involved in the writing of the tests, the cases are looked at from three differing perspectives. The questions asked during the writing from the points of view of different partners can bring forth defects that could have been missed otherwise or would have been noticed much later. (Pugh 2011)

The dark side of acceptance tests comes from automation of them. Automated tests require resources for their upkeep and running them, and changing requirements can break them. If the requirements change radically and there are many broken tests, the broken tests can begin to lag ant their fixing can be pushed to the next release of the software. Safe to say, if the tests are not fixed, bugs can slip through the cracks or broken tests can begin to be the new norm. (Pugh 2011)

2.4 Test automation

Test Automation is a vital part of Agile Testing, as automated tests for functionalities can serve as regression tests throughout the development process. Automated tests can be written with keywords, which makes it possible for anyone without coding knowledge to write the tests. Also, tests done using keywords are easier to maintain and create. (Liao, Wu et al. 2013)

Test Automation means using software that is unrelated to the SUT to perform, track and manage test cases on the SUT. This can mean automating functions of the delivery and testing pipelines, like reporting test results automatically, maintaining the Continuous Integration pipeline and creating input data for the tests. Test automation is a clear part of the software development of the present-day, sitting in between of quality assurance and continuous integration. Test automation serves an important role in assuring the quality of software during the development and maintenance of the product. Test automation is used in Agile to conduct continuous testing, for example in TDD. (Virtanen 2018)

Test automation is most useful in completing laborious, repetitive tasks. Compared to manual testing, test automation allows for shorter testing periods, offers higher levels of detected faults, and the manual regression testing can possibly be cut out completely. The quality of testing in automation testing stays the same throughout executions, as the steps are always the same as compared to manual testing, while also producing reports of the testing steps taken. Test automation also reduces the testers burden in manual testing, as it can be used to create data for the testing, create test situations and shut down testing situations, so that there is more time for the actual testing. However, test automation cannot be used to measure defects in unquantifiable, user experience-based metrics, such as usability or logic. That is where exploratory testing shines. (Adzic 2011)

In a study conducted by Dudekula Mohammad Rafi et al. (Rafi, Moses et al. 2012) the most beneficial aspects and restrictions of automated testing were uncovered. The most beneficial aspects and biggest limitations are listed in Table 1.

Table 1. *Test automation benefits and limitations (Rafi, Moses et al. 2012).*

	Benefits	Limitations
1	Reusable tests making automated testing more productive.	Higher costs than in manual testing, particularly at the initiation.
2	Test repeatability, making it possible to run more tests in a shorter timeframe.	Maintaining and designing require more work.
3	Improved quality due to increased test coverage.	Specific required skills for the test developers.
4	Saved time and resources as tests can be easily run again without extra investment.	Costs of investing in tools and training are high.
5	Increased confidence in produced product and deadlines met more easily.	No guarantee on finding complex errors.
6	Testing tools reduce the workload of developers.	Tools for testing can be conflicting and wanted functionalities can be missing.
7	Reduced overall costs.	Do not completely mitigate manual testing.
8	Reduced testing effort.	

Some common uses of test automation are regression test automation, automation of processes, automation of workflow, test automation for development and V&V, or verification and validation.

Automated regression testing means automated tests that are run continuously, especially in between software updates, to guarantee quality and that previous functionalities have not been broken due to newer functionalities. It is a major piece in Agile testing, used as a test automation suite that continuously tests the SUT on CI-servers, reporting the results every time. (Crispin, Gregory 2009)

Test automation is used in the development phase to report on the quality of the software continuously. The developers develop unit tests for the functionalities they code as well as maintain them as the SUT evolves. In TDD, these tests are used as the starting point for development. Test automation is used in automation of the testing workflow by automizing the reporting of bugs or creating data for testing automatically and in automation of processes by automizing the gathering of feedback from customers. (Virtanen 2018)

The aim of test automation is to achieve verification and validation, commonly known as V&V. In the IEEE standard dictionary of electrical and electronics terms verification is described as an activity of defining if the outcome of a software development phase matches its requirements and validation as the activity of assessing the produced software when its development is finished (IEEE 1984). Automatic tests can be set to verify that developed functionalities match their requirements and to validate the quality of the product. (Virtanen 2018)

2.5 Test Plan

A Test Plan is a document describing the scope of the tests, objectives of the tests, testing environment, strategy of testing, test deliverables, risks associated to the testing, testing timetable, testing levels, methods and techniques of testing and the tools used for testing. The test plan should be sufficient to support the needs of the client and the developing organization. As illustrated in Figure 8. below, the importance of a test plan rises exponentially as the project gets more complex. (Craig, Jaskiel 2002)

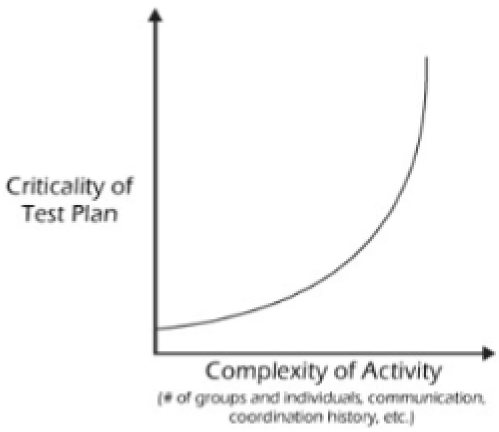


Figure 8. *Importance of a proper Test Plan (Craig, Jaskiel 2002).*

Testing is done by keeping an eye on the objectives or a specified goal for the testing. This goal should be documented clearly and in quantifiable means, like effectivity of tests or test coverage in code lines. Even though the point of testing is to uncover defects, good testing also looks at other properties, like maintainability, portability, usability. (Syed, Kumar 2011)

As stated in the IEEE Standard 829–1998 for Software Test Documentation (IEEE 1998), the purpose of a test plan is to:

- “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.”

It’s useful for an organization to have a template for their Test Plans. In the IEEE Standard 829–1998 for Software Test Documentation there is an example structure of a Test Plan, which can be used as a template and modified to the specific needs of an organization. For example, if the risks are needed to be approach from several directions, the one section in the standard can be divided in to several according to the needs. It is also possible that some sections are redundant to the testing organization and are left empty every time. In this case it is not useful to drag the irrelevant sections in the template, so they should be removed. (IEEE 1998, Craig, Jaskiel 2002)

The ANSI/IEEE 829-2008 standard offers guidelines for the Master Test Plan and the smaller Level Test Plans. Of course, the guidelines are only a framework, and the plans need to be adapted according to the specific project in question. The final document can be in written form but can take other forms as well depending on the necessities of the specific project. (IEEE 2008)

The standards test plan template includes the following 20 items (IEEE 1998, Craig, Jaskiel 2002):

1) Test plan identifier.
2) Table of content.
3) References, like Process plan and Quality Assurance Plan.
4) Glossary, meaning terms and acronyms explained.
5) Introduction, meaning scope such as Integration, System and Acceptance Testing
6) Test items, such as specification, user, operation, and installation manuals, or in lower-level plans, features to be tested.

7) Software risk issues, such as system interfaces, features with huge amounts of money or that affect a lot of stakeholders, features with a history of failures or a lot of changes.

8) Test features in more detail, meaning what is going to be tested from the end-user's point of view. Example from a Manufacturing Process Management system could be "Part location management".

9) Features that are not going to be tested, meaning features associated with a very low risk. Possibly unchanged features, unavailable to end-user, or no failures in the history.

10) Testing approach, meaning how testing will be done, techniques, tools and testing activities. Sometimes labelled as Testing Strategy, which can be a very wide section. Figure 19. shows usual strategic influencers. This section can contain the following:

    a. Methodology decisions. When testing begins? What testing techniques will be used? How many testers are needed for planning, designing, and executing the test? What are the testing levels?

    b. Resource decisions. For example, questions about time, money, and people.

    c. Decisions about test coverage. Code coverage, requirement coverage or design coverage?

    d. Decisions about reviewing. Inspections and walkthroughs are the two main reviewing ways. Walkthrough is a peer review which is done by going through the code line by line and an inspection is a formal technique for evaluating for requirements, code or design done by some other entity than the author (ANSI, IEEE 1983).

    e. Managing configurations, which includes managing changes and decisions on bug reviews, prioritization, bug-fixes, and re-tests.

    f. Gathering and validating metrics. How the metrics will be gathered, what they will be used for and how they are validated.

    g. Testing tools used.

    h. Handling of changes to the test plan itself.

i. Communication, meetings, and reports. For example, monthly status meetings, presentations to clients or managers.

Figure 9. *Influences of strategy (Craig, Jaskiel 2002).*

11) Criteria for passing and failing the test items. Possibly percentage of passed cases, number of errors, coverage, performance indicators.

12) Criteria for suspending and resuming testing. For example, critical tasks undone, critical defects or a lot of defects.

13) Deliverables from testing, meaning the Test Plan, specifications, logs, reports, and test cases.

14) Tasks for testing, meaning needed actions to get ready for testing and for conducting the testing. Sometimes as a Test Matrix under point 16.

15) Needs for the test environment such as hardware, data, software, facilities, interfaces, security access, and publications.

16) Responsibilities, who takes care of what. Possibly a matrix with responsibilities on one side and personnel on other.

17) Personnel and needs for training. How many people are required and what skills are required from them.

18) Testing schedule. Milestones from Project Plan combined with the Testing Milestones.

19) Risks to planning and their contingency plans. Some examples of risks to planning could be early delivery dates, too little staff, small budget and needs for training and tools. Contingencies could be increasing resources, like staff and money, and pushing the delivery dates back.

20) Approvals, who will sign and approve the plan. These persons should be those who can approve and say that the product is ready for going on to the next phase. The approvers should be a part of the planning process or at least the reviewing of the plan.

As mentioned in the list, the points 14 and 16 can be combined into a test matrix, or possibly a RACI-table, shown in Table 2. A RACI-table shows who does what and helps in remembering who should do what. In the table, the rows state the tasks to be accomplished, and the columns point to people or their roles. RACI consists of the following: R for Responsibility, A for Accountability, C for Consulted and I for Informed. The person who has been assigned the R performs the task or is among those who perform it. The A assignee supervises and makes sure the task is performed. Every task must have an R assigned and every task has only one person with an A. A person assigned a C can be consulted on the problem and a person assigned an I is informed on the completion of the task. There can be anywhere from zero to infinity of people assigned a C or an I. (Lehtimäki 2006)

Table 2. *RACI-table (Lehtimäki 2006).*

Tasks	MR. X	MS. X	MR. Y	MS. Y	MR. Z
Task 1	I	A/R	C		
Task 2	A/R	I	I	I	I
Task 3	I				A/R
Task 4	I	C	A	R	R

Although the test plan is a document that waits at the end of the planning process, the actual planning process is the more important of the two. The actual goal of the planning is communication. The planning is the delivery mechanism for the tester's intentions, goals, and other knowledge to the rest of the team and it makes sure the rest of the team knows about what the testers are going to do. (Patton 2005)

Sometimes it is necessary to do testing without a concrete plan. If the product is a newer version of a previous, thoroughly tested software, the testing team can have the knowledge to test without a test plan if they have worked along with the project for a long time and it will not be a big deal. The team could have members with specific knowledge about parts of the SUT and all team members are familiar with the industrial automation system, so the test plan is not a vital component. (Ahmed 2010)

If the SUT is new to the team, then testing without a plan can pose some obstacles, such as not testing and covering all or most functional features of the SUT; no risk management so risks are not known and are left unattained; even if some bugs are found, it is hard to reproduce them without knowledge of the system. (Ahmed 2010)

2.6 Test planning

Test planning is a process which creates a document that allows all participants of the testing process to communicate with each other clearly and decide what are the most important issues and how to deal with them. Ultimately the goal of test planning is not to create a long document to sit on the self, but to deal with the aspects of testing, such as testing strategy, risks, priorities, responsibilities, and resources of the project. The process should involve all testers and key actors of the production team. The process builds an understanding of what is going to be tested, why and how it is going to be tested. This understanding early in the development life cycle can make the testing run more smoothly and can save a lot of money and time. (Craig, Jaskiel 2002, Patton 2005)

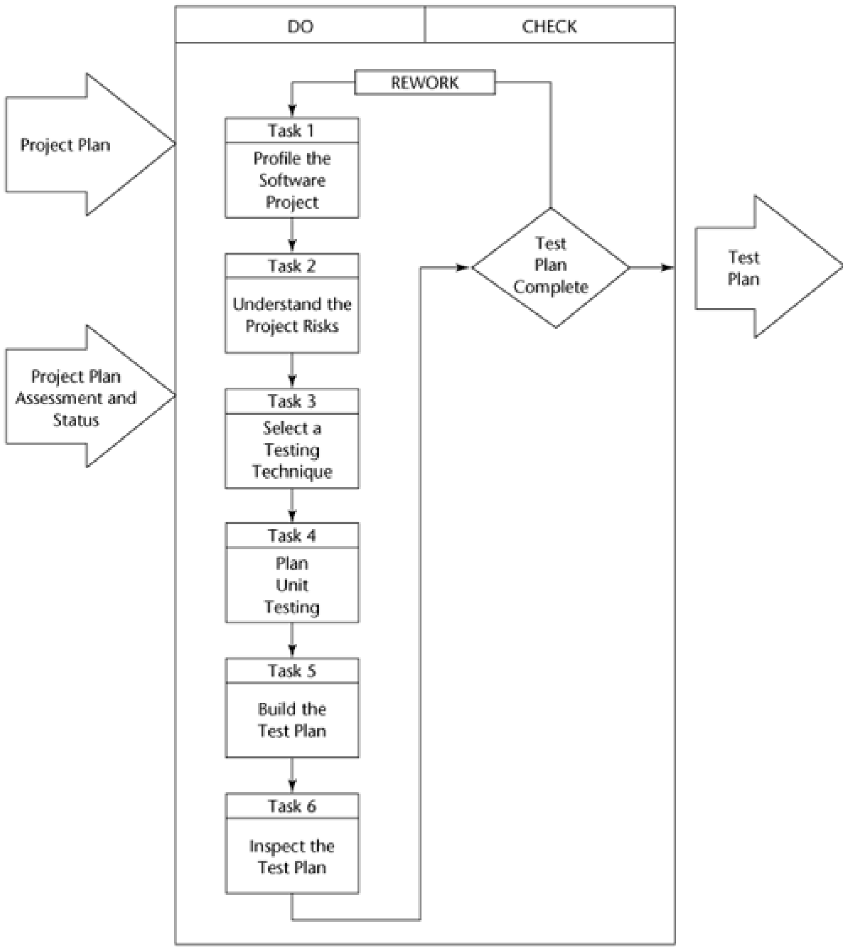


Figure 10. *Test planning procedure (Perry 2006).*

As seen in Figure 10, according to Perry, the Test Plan can be completed with the help of six steps: Profiling the software projects; Understanding the risks of the project; Selecting a testing technique; Planning the Unit Testing; Building the Test Plan; Inspecting the Test Plan. Of course, for these steps to be effective, they need correct and accurate inputs in the form of a Project Plan which documents all the actions necessary for completing the project and controlling its execution, and a report of the feasibility of the Project Plan. (Perry 2006)

The very first question when starting to do test planning is “Who is the test plan directed to?”. The answer can be different depending on the level, for example a Master Test Plan can be read by executives, in which case you should have a summary of the plan, because executives as well as other people may not be willing to read the plan if it is 60 pages long. Also, the use of technical terms and acronyms should come in to accord when thinking of the audience. If the plan seems like that it is going to be very long, it may be useful to plan smaller portions around different functions. (Craig, Jaskiel 2002)

The first task, profiling the software project, consist of two main steps: doing a walkthrough of the customer area and developing a profile of the project. The walkthrough will provide a bird's eye view of the users' activities that will be performed in the complete industrial automation system and offers an understanding of how the software will be used. The second task is to collect the information for the profile to help prepare for the planning. Information to be gathered contains the following: project objectives, development process, customers, deliverables, schedule, limitations, legal issues, talents of the developers, tools to be used, databases, interfaces and statistics. (Perry 2006)

Next task is understanding the risks related to the project. Test factors are the risks or concerns that the testers must comprehend to be certain the objectives of that factor have been reached. A matrix of the testing concerns can be constructed to determine the different characteristics of the project that can be looked at to see if the test factors have been handled. Some test factors can be, for example, Reliability, Ease of use, Maintainability and Performance. Each test factor then can have explanations about their requirements, necessary designs, programs implemented, testing phase, necessary operational tasks and maintenance tasks. (Perry 2006)

These test factors, or risks, should then be evaluated by the testing team. There are twelve steps to achieve this (Perry 2006):

1) Defining how to meet the objectives.
2) Understanding the core business.
3) Evaluating the potential harm of possible failures.
4) Recognizing the components of the systems.
5) Defining the test resource needs.
6) Develop testing plans: Unit Testing, Integration Testing, System Testing, Acceptance Testing.
7) Getting the necessary tools for testing.
8) Recognizing the testing environment needs.
9) Planning the schedule.
10) Looking at issues with interfaces.
11) Manufacturing contingency plans.
12) Recognizing at risk system parts and processes.

The third task is to select a technique for testing. The test factor also helps to identify the testing technique, the chosen testing technique should be identified by their ability to reach the testing objective of the test factor. The testing can be either functional or structural, the difference being structural testing makes sure the structures of the system and the system as a whole work as intended, and functional testing makes sure the functions operate as they should and that the specifications and requirements of the system are met. Some techniques for structural testing are Stress Testing, Recovery Testing and Security Testing and for Functional Testing some examples are Requirements Testing, Regression Testing and Parallel Testing. (Perry 2006)

The fourth task is to plan the Unit Testing. As discussed in Chapter 2.1. Levels of testing, the Unit Test focus on smaller entities in the system. Unit Testing can be structural or functional but also error-oriented, which means techniques that focus on the actual errors or on trying to show that there are no errors in the code. When selecting the technique to use, the selectors should look at the testing goals and the natures of the product and the environment. Different goals need different techniques as functional testing does not cover a lot of code and structural testing does not cover a lot of the specifications. (Perry 2006)

The fifth phase is building the test plan which consists of four tasks: Setting the testing objective; Developing a test matrix, Defining the administration of the tests; and writing the test plan. The objectives of testing should be in line with the objectives of the project plan and should determine if the project plans objectives have been reached. The objectives should be quantifiable as should their passing criteria. The simpler the criteria the better the tests and the easier they are to test. The test matrix is the most important feature of the test plan. It shows what is to be tested, how it is to be tested and shows that all features have a test for them. An example of a test matrix can be seen in Table 3, showing four functions with three tests for them. The administration of the tests defines the schedule, necessary resources, and the testing milestones. In other words what is to be tested, by whom, when is it going to be tested and when it is going to be finished, the resources and money needed for the testing. The last task is writing the test plan which basically consists of the steps and the produced document before the current step. The test plan can be informal or in a more formal standardised package, depending on the organizations culture. (Perry 2006)

Table 3. *Example test matrix (Perry 2006).*

SOFTWARE FUNCTION	TEST DECK TRANSACTION	PARALLEL TEST	CODE INSPECTION
FICA Calculation	X		X
Gross Pay	X		X
Tax Deduction	X		X
General Ledger Charges		X	X

The last phase is Inspection. In this phase ready products that have yet to be tested are assessed and the changes are evaluated. Inspectors compare the ready product to the product before the changes and to the definitions of the changes made, looking for defects from three categories: errors or changes that do not work right; missing implementations; and additional changes, in other words, changes that should not have been made. The targets of the inspections can be specified in the project plan, but should include requirements specifications, software maintenance documentation, changed technical documents, source code changes, the test plans, and the users' documentations. (Perry 2006)

2.6.1 Agile Test Planning and Iteration Planning

A big reason for developers to utilize Agile methods is that they have tried traditional planning and noticed that it does not work very well for them. In most businesses the situation can change rapidly and often, making the already developed big plans obsolete. In Agile development there are no big plans, but it is useful still to have some sort of idea about what the customers' needs are and how the development should get started. (Crispin, Gregory 2009)

In Agile development it is sometimes said that there is no planning. Contrary to that belief, release planning should be done at the beginning of the development of an epic or a bigger ensemble of features, on a higher level. Things that are considered in the Release planning are the customers' requirements; The objective of the release; The whole image; How does it benefit the customer; Who are involved in the project; When are Unit Acceptance Tests held; When are code releases scheduled; How is progress tracked? (Crispin, Gregory 2009)

The Release planning's are not meant to cover every detail of the cycles of development, but with an estimation of the standard speed of development the Release planning can cover the scope of the release, what is in the release and how long or how many cycles could it take to complete. The user stories, or functional descriptions, that the customer wants can be sorted corresponding to other stories, then the features derived from those stories can be prioritized accordingly. In the planning phase the customers and the developers can see causal connections of the planned features to the whole SUT and to other features, to better see what should be done first and better consider needed testing software and environments. (Crispin, Gregory 2009)

At the beginning of the project, there is no idea about what the iterations hold in detail. There can be some guiding ideas about the stories, what stories are more important than others, and so an estimation can be made of prioritizing them to specific iterations. During the planning it is important to look at testing from a higher perspective, to allocate enough time for testing. Detailed plans can be done when planning the specific iterations. Test planning helps avoid some pitfalls of Agile development, specifically forgetting the bigger picture of the project. (Crispin, Gregory 2009)

In lightweight planning, the project team focuses on understanding the stories of the highest priorities that have been ordered to be developed first, as the other stories or features need additional time to define the test cases for them. When the stories in the release are clearer, the scope of the testing can begin to take form. Question to consider include: Are there presumptions that could alter testing? Are there software's of a third-party utilized in the testing? Are there risks that should alter testing? Are there new technologies that need to be learned for testing? The environmental needs for testing, testing data needs and technology needs for test automation need to be considered as well. (Crispin, Gregory 2009)

The test plan can consider the agile testing quadrants and the different types of testing necessary in the project. A need for Unit Acceptance Testing or if it is covered in iteration demos for the customer needs to be considered. The different infrastructures need to be considered as well as the need for a CI-pipeline, some special testing environment, test data, or a server for automation testing, and of there are some special tools needed for the testing. One thing to consider is the reporting of testing results. It can be that the customer requires extensive reports on how the product has been tested, or that the producing organization has some standard for reporting. (Crispin, Gregory 2009)

The test plan can be in a document form but in Agile the form is very informal and can be whatever serves the needs of the planner. A lightweight test plan for example can include only the necessary parts and nothing that is mentioned in the project plan or other documents. A light test plan could include what functionalities are in and out of scope, what are the new functionalities, what is tested in other lower testing areas like performance testing and UAT and how they are tested, what are the environmental needs, are there any presumptions considering testing, and a risk assessment portion with identified risks and mitigations for them. The test plans' primary purpose should be to help identify risks to the testing of the project. (Crispin, Gregory 2009)

Iteration planning consist of handling one story at a time, writing task cards of the story, approximating, and documenting all the necessary tasks for that story's completion, giving a good estimation of what is involved in the story. The task cards hold in them the developing and testing of the task. There should be a delivery owner or someone with customer knowledge attending the planning, to address rising questions and explaining requirements for the functionalities. User examples about stories can make them easier to understand and help create test cases for the stories, for example for TDD. The stories should be in order of priority to give emphasis to more important stories that come first. A useful practise for iteration planning is to develop user acceptance tests for the stories, which are high-level and verify that the story is finished when they pass and give the developers a good direction to work at. The question of how many stories are in an iteration should be decided by how many can be developed and tested, and not just how much can be coded, as some stories can be easy to code but harder to test. Ultimately the test cases should be reviewed with the customer and the programmers, to make sure they satisfy everyone and that they convey the customer requirements. The high-level test cases also work as the project's documentation. (Crispin, Gregory 2009)

2.7 Prioritizing tests

Testing contains covering all functionality of the industrial automation system with tests, from the most insignificant feature to the most significant. Ergo, it is beneficial to prioritize the tests so that the more significant features are tested earlier than the less important. (Ahmed 2010)

A study conducted by Srikanth and Banerjee suggests an approach to prioritizing test cases on system-level titled Prioritization of Requirements for Test, or PORT(Srikanth, Banerjee 2012). There are four individual parts in the approach that can help in detecting failures and improve value of the business: customer-assigned priority, complexity of implementation, fault proneness, and requirements volatility. (Srikanth, Banerjee 2012)

Customer-assigned priority is a quantifiable value assigned by the customer to the requirements. The value ranges from 1 to 10, 1 being the least important and 10 being the most important. Prioritizing tests this way increases the customers satisfaction and perceived value, improving value of the business simultaneously. (Srikanth, Banerjee 2012)

Developer-perceived complexity of implementation is a measurement of how hard the feature is to implement in coding. Features are analysed to perceive the difficulty of implementation and given values from 1 to 10, 10 being the hardest to implement. (Srikanth, Banerjee 2012)

Fault proneness entails detecting the features that have had the most failures in the past and prioritizing testing based on them (Srikanth, Banerjee 2012). To further support this, a study by Ostrand, Weyuker, and Bell concludes that 20 percent of files account for 83 percent of faults (Ostrand, Weyuker et al. 2004).

Lastly, requirements volatility, which is a number from 0 to 10 indicating the number of times the feature has changed in development phase. (Srikanth, Banerjee 2012)

FMEA, or Failure Modes and Effects Analysis, is another technique to prioritize the testing, providing a methodology for identifying failure modes in the system and analysing the consecutive effects. FMEA utilizes three indexes which are defined with simple questions. The indexes are S, Severity; O, Occurrence; and D, Detectability, and each of them are measured with a simple, quantifiable scale from 1 to 5 or 1 to 10, 1 being the smallest effect, detectability, or occurrence likelihood, as seen in Table 4. (Meriem, Abdelaziz 2019)

Severity determines the effect of failure, how serious it is if the feature fails, how bad the consequences are. Occurrence determines the likelihood that the failure will happen, or how often the failure can happen. Detectability determines how likely the failure or a possibility for a failure will be noticed when it happens. (Meriem, Abdelaziz 2019)

Table 4. Scales for Occurrence, Severity, and Detectability (Meriem, Abdelaziz 2019).

FMEA scale for probability of occurrence (O)

Probability rating	Class	Meaning
1	None	Failure never occurred
2	Very minor	Failure unlikely occurred
3	Minor	Failure is Relatively low (Rarely occur)
4	Below average	Failure occasionally occurred
5	Average	Failure infrequently occurred
6	Above average	Failure little Occasional
7	High	Failure repeated frequently
8	Very high	Failure is almost inevitable
9-10	Extremely high (with warning)	Failure is inevitable (Always)

FMEA scale for severity (S)

Severity rating	Class	Meaning
1	None	Failure with no impact or effect
2	Very Minor	Failure with a very low impact
3	Minor	Failure with a low impact
4	Below average	Failure with less moderate impact on system operations (Tolerable impact)
5	Average	Failure with moderate impact on system operations (not tolerable in critical situations)
6	Above Average	Failure with very moderate impact on system operations
7	High	Failure with high impact on system operations
8	Very high	Failure with very high impact on system operations (Reduced performance)
9	Extremely High (with warning)	Failure with very critical impact on system operations but with warning in advance.
10	Extremely High (without warning)	Failure with very critical impact on system operations and without warning.

FMEA scale for Detectability (D)

Detectability rating	Class	Meaning
1	Certain	Failure mode very likely to be detected, it will be detected certainly
2	Very high	Very high chance to detect the potential and subsequent failure mode
3	High	High chance to detect the potential and subsequent failure mode
4	Above average	Above average chance to detect the potential and subsequent failure mode
5	Average	Moderate chance to detect the potential and subsequent failure mode
6	Below Average	Low chance to detect the potential and subsequent failure mode
7	Remote	Remote chance to detect the potential and subsequent failure mode
8	Very Remote	Very remote chance to detect the potential and subsequent failure mode
9	Almost Uncertainly	There is almost no chance to detect the potential and subsequent failure mode
10	Uncertainly	There is no chance to detect the potential and subsequent failure mode

The FMEA indexes can then be used to determine the RPN, or Risk Priority Number, which will then quantify the risk and its effects and allow prioritization with the number. The RPN can be calculated by multiplying each factor with each other, as seen in Formula 1. (Meriem, Abdelaziz 2019)

$$RPN = S \times O \times D \qquad \textbf{\textit{Formula 1.}}$$

Also noteworthy is software risk analysis as a way of prioritizing test using a risk matrix, as discussed in the next chapter.

2.7.1 Software Risk Analysis

Most projects have a predetermined date of delivery for the prescribed product and predetermined costs to go with them. If any of the before mentioned attributes are not met, then the project has failed. Risks are what can hinder projects from meeting these criteria, causing the project to be late, more expensive than planned, or the product can simply be lacking in quality. (Ahmed 2010) Risks are the problems of the morrow, meaning that risks are problems that can generate losses but have not yet occurred (Singh 2012).

The point of risk analysis in software projects is to find out what is going to be tested, how in depth the testing will be, the priority of tests, and sometimes what is not going to be tested. Risk analysis can assist in prioritizing features that need to be tested with a keener eye. The risk analysis results can be utilized in prioritizing features that need to be tested in the SUT. (Craig, Jaskiel 2002)

In Figure 11. there are ten steps illustrated that describe the process of analysing risks. The steps are opened and explained after the figure.

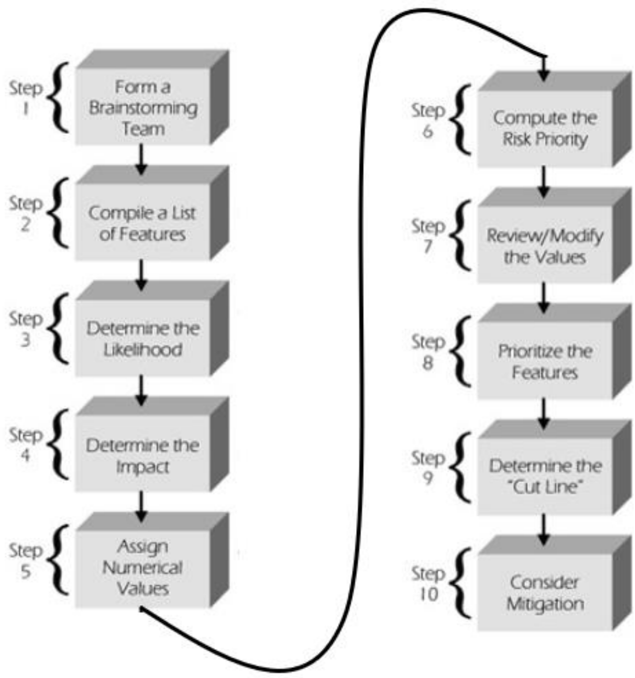


Figure 11. *Risk analysis process (Craig, Jaskiel 2002).*

Step 1: Form a team for brainstorming including developers, end-users, testers, marketers, and analysts.(Craig, Jaskiel 2002)

Step 2: Compile a list of features and attributes that is as wide as the industrial automation system. In this step the brainstormers need to gather information available of the project, such as specifications of requirements and functionality, desired changes, reports, and logs. Then from these documentations the brainstormers should identify features, at the beginning in a higher level and later in more detail as needs are clarified. Larger scale examples can be availability, usability, security, and maintainability. (Craig, Jaskiel 2002)

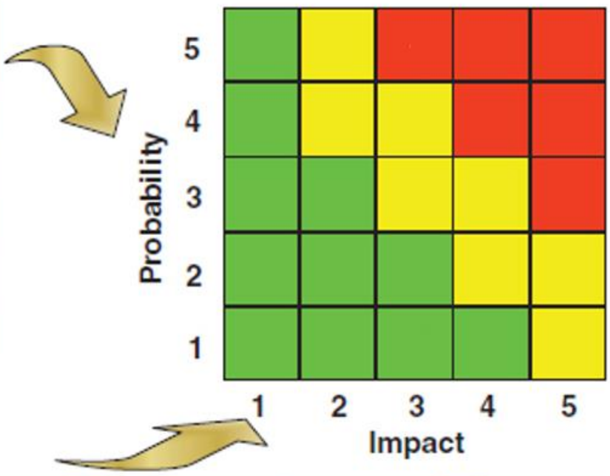
Step 3: Determine the likelihood, meaning assigning a value to the likelihood of the feature failing to work. It can be H like high, M like medium, or L for low, according to the likelihood of the failure. The values can be assigned by thinking what the likelihood is the feature will fail completely or does not operate correctly according to the current knowledge. The indicators are normally the cause of the systems properties, which can be more easily measured by members who are familiar with the technology. (Craig, Jaskiel 2002) Other possibilities are to assign the likelihood from 1 to 5, 1 being not likely and 5 being near certain, as illustrated in Figure 12 (Laporte, April 2017); a scale from 1 meaning low to 10 meaning high (Singh 2012); or to count the times the risk occurs in the project and use that as a measure of likelihood. (Ahmed 2010)

Step 4: Determine the impact, meaning assigning a value to the impact of the feature failing to work. The same values from high to low are used here, and the brainstormers should be thinking how this feature would impact the user if it were to fail. (Craig, Jaskiel 2002) Other possibilities are to assign the impact from 1 to 5, 1 being the lowest and 5 the highest impact, as illustrated in Figure 12 (Laporte, April 2017); a scale from 1 meaning low to 10 meaning high (Singh 2012); or to give a risk assessment questionnaire with close-ended questions, asking question from the group to determine the risk level from zero meaning not applicable to five meaning high risk (Ahmed 2010) .

Step 5: Assigning numerical values according to the determined likelihood and impact values. Here the brainstormers should, if not done already, give numeric values to the assigned letters. The values can be 1 for low, 2 for medium, and 3 for high, but can be any descending numbers. For example, impact can be 6 for high if its effect needs to be amplified, possibly because it is a safety-critical feature, in which case its testing is of the highest priority. (Craig, Jaskiel 2002)

Step 6: Computing the risk priority, meaning adding together the likelihood and impact values. The risk priority can also be calculated by multiplying the values together to emphasise the riskier features, which can work best if the assigned likelihood value comes from the number of times the risk occurs in the project (Ahmed 2010). (Craig, Jaskiel 2002)

Value	
1	Not likely
2	Low likelihood
3	Likely
4	Highly Likely
5	Near Certainty



Level	Technical	Schedule	Cost
1	Minimal or no impact	Minimal or no impact	Minimal or no impact
2	Minor performance shortfall, same approach retained	Additional activities required; able to meet key dates	Budget increase or unit production cost increase < 1%
3	Moderate performance shortfall, but work-arounds available	Minor schedule slippage; will miss need date	Budget increase or unit production cost increase < 5%
4	Unacceptable, but workarounds available	Critical path affected	Budget increase or unit production cost increase < 10%
5	Unacceptable, no alternatives exist	Cannot achieve key milestone	Budget increase or production cost increase > 10%

Figure 12. *Risk matrix (Laporte, April 2017).*

Step 7: Review and modify values according to for example the complexity, new or changed features, Pareto analysis, development methodology, usability, or historical data (Craig, Jaskiel 2002). Other ways to modify the values can be to calculate the Customer Priority, the Fault Proneness, or both together, which all have been shown to effectively prioritize test cases, and which will be discussed more in Chapter 2.4.2. Customer satisfaction and 2.4.3. Field Failures or Fault Proneness (Srikanth, Hettiarachchi et al. 2016).

Step 8: Prioritize the features according to risk priority. Here the brainstormers should arrange their list of features in order by the risk priority value calculated in the prior step. A downfall of this method is that it does not consider the effects of features on other features and their dependencies, but the features can be arranged according to those in a later phase when the dependencies are clearer. (Craig, Jaskiel 2002)

Step 9: Determine a "cut line" for separating test that are going to be tested from those which are not going to be tested or are going to be tested with a smaller effort. The "cut line" can be determined by looking at the time and resources designated for testing. (Craig, Jaskiel 2002)

Step 10: Consider Mitigation. Which risks can be lowered by more resources, or a different methodology? The mitigation can mean meticulous software inspections or assigning more time, testers, or other resources to the features testing. (Craig, Jaskiel 2002)

The product of the prior ten steps can looks something like Table 5. below, with variances in the priority values and likelihood and impact indicators.

Table 5. *List of priorities (Craig, Jaskiel 2002).*

ATM Software		Likelihood	Impact	Priority	Mitigation
Features	**Attributes**				
Withdraw cash		High	High	6	Code inspection
Deposit cash		Medium	High	5	Early prototype
Usability		Medium	High	5	Early user feedback
	Security	Medium	High	5	
Transfer funds		Medium	Medium	4	
Purchase stamps		High	Low	4	
Make a loan payment		Low	Medium	3	
Check account balance		Low	Medium	3	
	Performance	Low	Medium	3	

Some important aspects to keep in mind when undertaking the software risk assessment and filling the risk matrix are the risks caused by estimations of the schedule, the resources, or the budget; risks caused by communication; risks related to effectiveness, the culture, the size, the process, technology, or lacking skills. (Ahmed 2010)

While it is very convenient and widespread to use the risk matrices, according to Anthony Cox in his article, “What’s Wrong with Risk Matrices?” in the Risk Analysis journal points to caution in using them. According to Cox it is not certain that the risk matrices underwrite good, meaning better than random, risk management and successful management of resources and limited time. (Cox 2008) On the other hand results also point to prioritization based on risks being a good candidate for test case prioritization (Srikanth, Hettiarachchi et al. 2016). Of course, the effective use of these methods depends on the user’s diligence and the situation in which they are used.

2.7.2 Customer Priority and Fault Proneness

Customer priority (CP) is a value given to every feature based on their perceived impact on customer satisfaction and usage, assigned by a team of people who face the customer. In a study, CP had the largest improving effect on fault detection rates (Srikanth, Banerjee et al. 2014). CP tells about the importance of a particular feature or attribute to the customer. In a study conducted by Moisiadis, the findings indicated that about 45 percent of features are left unused, 19 percent are used occasionally, and 36 percent of the features are used regularly (Moisiadis 2000). Customer-perceived value and customer satisfaction have been observed to increase by focusing on customers priorities in development (Karlsson, Ryan 1997, Boehm, Huang 2003, Boehm 2003). Also, the successful execution of features that are of a high priority to the customer can improve opportunities for business. Often the higher customer priority targets are related to crucial features. (Srikanth, Hettiarachchi et al. 2016).

Prioritizing test cases with the CP value involves ordering the tests so that the higher CP value tests are performed first. The prioritization can also be done in another way, transforming the CP value into a business-criticality based risk impact value, or RIBC, which utilizes information about risks for prioritizing and utilizes the customer priority for the impact value of the risk. The RIBC can vary from 1 to 5. The derived RIBC can then be used in conjunction with risk likelihood by multiplying them together to get the risk priority number. In the study by Srikanth and Hettiarachchi, prioritization with the business-criticality based method resulted in an average percentage of fault detection, or APFD, of 71. (Srikanth, Hettiarachchi et al. 2016)

Fault Proneness, or FP, tells the story of previous faults in old requirements or features from a previous version of the product. The FP indicator utilizes field failures as well as testing failures. As discovered by Ostrand et al, focusing on features that have a higher likelihood of having errors can enhance the efficiency of testing (Ostrand, Weyuker et al. 2004). (Srikanth, Hettiarachchi et al. 2016)

The FP value is calculated with the reported number of field failures. The number of field failures for a given feature is divided with the number of all the field failures of the industrial system. For instance, if we had 60 field failures in the whole industrial system, and 12 of the field failures were located in feature X, then the ration would be 12/60 or 0.2. Then the given field failure ratio is normalized into a scale of 0 to 5, zero meaning there are no field failures in that feature. Legal reasons dictate that the precise number of field failures cannot be reported, and so the values need to be normalized. As expected, the test cases with the higher values get a higher priority and are executed first. The APFD for prioritizing on the Failure Proneness alone got the highest score of 80 in the study by Srikanth and Hettiarachchi. (Srikanth, Hettiarachchi et al. 2016, Srikanth, Cashman et al. 2016)

According to the results of Srikanth and Hettiarachchi, prioritizing with the FP indicator gives the best fault detection rate. When the FP and CP indicators were used simultaneously for prioritizing, the outcome was a higher fault detection rate than prioritizing with only the CP indicator. They also noted a strong relation with both indicators, meaning it is beneficial to use either of them to improve testing prioritization as compared to using a random order of tests. The risk-based prioritization techniques were also shown to be useful techniques in prioritizing tests. (Srikanth, Hettiarachchi et al. 2016)

3. PROPOSED FRAMEWORK

The previous research points to a comprehensive test plan being a great tool for finding errors and executing testing in a concise manner. Often it is a time-consuming process but has great potential in improving business value and quality of the software system, if done correctly. It can be difficult to describe the scope of the tests, objectives of the tests, testing environment, strategy of testing, test deliverables, risks associated with the testing, a schedule for testing, testing levels, methods, and techniques of testing. There is a lot of literature about what these necessary parts of the test plan are in a traditional waterfall approach, but there is a lack of guidance on the necessary parts in an Agile development project of an industrial automation system, that is trying to avoid unnecessary repetition, but still have the benefits of a test plan. But with the help of concise guidelines for developing a concise test plan the process can be made easier and more streamline.

In many sources it seems that in Agile, there is no practice for manufacturing a test plan. In Agile, making documents must serve some purpose and the traditional test plans are very cumbersome to do in a quick Agile development sprint. But the benefits of planning the testing approach beforehand is unmistakeable, and so, a lighter version of a test plan can be used in an Agile approach, only having the necessary parts, and not repeating lists of features that can be found in every other document and project plan.

Some pitfalls that testers need to watch out for are in prioritizing the test cases. There are many ways in literature for deciding what to test first. A big chunk of the decision on what to test first lies on the nature of the project. If the industrial automation system is new, there is no historical data to rely on and possibly not a lot of knowledge in the team on similar projects, or there is no data concerning the customers priorities. There can also be safety criteria that need to be assessed and possible dependencies between the different features of the industrial automation system, that can influence the way and order different features need to be tested. Deciding how to conduct the testing or in which order can also be a challenging task, but it can be analysed thoroughly with risk management and a diverse team of people to brainstorm.

Often companies that do and have done their own software development for a while can have ways of testing already in place, but these techniques can be scattered in various documents and places among the different libraries of the company. A Master Test Plan like plan could there for be used to gather the testing related information into one place, so that it would be easier to find the data about testing when it is needed.

The goal of the following framework is to make it easier for the development or the testing team to manufacture a plan for testing, gather the necessary testing related information into one place and provide a way for making sure the necessary tasks have been completed and the system is thoroughly tested before delivery. This proposed framework will be primarily for industrial automation systems, more precisely Manufacturing Project Management systems, or MPM's, and directed at an Agile approach of developing.

3.1 A framework for forming a simplified test plan

In this chapter a simplified framework for developing a test plan in an Agile software process is introduced, largely adapted from the standard test plan template. The headers in this chapter are formed like the headers of a test plan document.

3.1.1 Introduction

As in the standard test plan template referred to in Chapter 2.5 Test Plan, in this chapter of the test plan the identifier and the introduction are brought together, and so in this part the project should be identified, with a test plan identifier like ClientCompany-Name_123432, or something else that identifies the project. The introduction should also entail a short summary of the test plan, as stated in Chapter 2.6 Tests Planning, so it is easier to better outline the plan in a quick glance and not necessary to read through the whole plan.

3.1.2 References

The idea behind this section is to make it easier to find documents that are relevant to the project, without having to spend hours searching in company databanks for the info. Some example documents could be Process Plan, Quality Assurance Plan, Specifications, selling documents and checklists for testing, gathered as suggested in Chapters 2.5 Test Plan and 2.6 Test Planning. There should be links to the company's file management software or otherwise a location for the document.

IEEE Std. 829-1998 Standard for Test Documentation recommends:

- Project Plan,
- QA Plan,

- Configuration Management Plan,
- Relevant Policies
- Relevant Standards
- Requirement Specifications

3.1.3 Scope

In this section it should be described what is to be tested in the project. For example, testing may include all new features, tailored features, high risk features, User Acceptance Testing, Load, Performance and Stress Testing, and Manual Testing done as Exploratory Testing.

What parts of the SUT are included in this project should also be included here if they are not mentioned elsewhere. For example, “This project will include blocks 1 through 10 excluding block 8”. Also worth to mention is if there are some parts of an original source project that are out of scope. For example, translations are done as subcontracting and they are tested by the subcontractor or lift functionality is tested already in the original project and there are no changes to that area.

3.1.4 Test items

In this section it should be detailed what new features and tailored features are in this project. At the beginning of the software project, the tailored features and functionalities can be in larger entities called epics, which entail smaller more specific functionalities. An example list of features is shown in Table 6, but these epics could also be User Stories. The test items can be gathered from a Project Plan, a specification of sold project parts or any document that describes the functionalities of the SUT.

Table 6. *Feature list.*

Epic	Features
Pallet handling	Transport pallet to and from storage
Pallet handling	Load parts on to pallet
Order management	Report order progres
Order management	Monitor orders
Machine tool	Automatically grind part
Pallet handling	Move part from pallet to machine
Order management	Schedule orders
Reporting	Monitor efficiency
Washing machine	Wash parts
Order management	Add and remove order

In Agile the features are in User Stories, which work similarly to the Epics, entailing smaller functionalities. The User Stories describe how the SUT should work from the point of view of the user. Done in a table format, it can be easier to prioritize the features for development in the next part of the plan.

3.1.5 Risk Analysis

As in Chapter 2.7 Prioritizing tests and in its subchapters 2.7.1 Software Risk Analysis and 2.7.2 Customer Priority and Fault Proneness, in this section a list of features, or in the beginning of the project Epics or User Stories, is modified with values that describe the impact of failure, likelihood of failure, customer priority, fault proneness, and risk priority number.

As discussed in the literature, prioritizing with the FP indicator gives the best fault detection rate and it was deemed beneficial to use either the FP or CP values for prioritizing as compared to using a random order of tests. The risk-based prioritization techniques were also shown to be useful techniques in prioritizing tests. It would be beneficial to prioritize the testing and development of features in an Agile approach as well, as beginning with the more important features leaves more time for developing those.

The most efficient way to prioritize testing is with the FP value. The FP value or Failure Proneness is calculated with the reported number of field failures and errors found in testing. The number of failures for a given feature is divided with the number of all the detected failures in the industrial system, and then normalized to a range of 0-5, zero being no failures.

If there is no history data available, the second most efficient method of prioritizing is Customer Priority, or CP, multiplied by the determined likelihood of the failure to derive the CP weighed Risk Priority Number, RPN_{CP}. The CP score should be assigned by the customer, or someone from the team who is in contact with the customer.

If no FP or CP scores are available, the Risk Priority Number, RPN, should be used for prioritizing the testing and development. The RPN is derived from multiplying the determined likelihood of failure with the determined impact of failure.

Safety-critical features or features that can cause safety-critical failures in other features, meaning failures that can result in the loss of life or serious injuries, should be prioritized with a high priority even if their likelihood is deemed low. These features are marked by an X in the example Table 8, to allow sorting by them later.

Possible descriptions on how to identify the level of the likelihood and the impact of a failure can be found in Table 7. below, adapted from the Figure 12. Risk matrix. The table considers the impact of failure from the directions of technical impact, dependencies, cost, and safety implications, ranging from no impact to intolerable impact, while the likelihoods range from nearly impossible occurrence to nearly certain.

Table 7. *Possible definitions for values.*

Value	Likelihood of failure
1	Nearly impossible
2	Low
3	Likely
4	Highly likely
5	Nearly certain

Value	Impact of failure on: Technical	Dependecy	Cost	Safety
1	No impact or minimal	No impact or minimal	No impact or minimal	No injuries
2	Small performance hindrance	Additional tasks, but will not miss scheduled dates	Feature production cost or budget increse <1%	No injuries
3	Medium performance hindrance	Medium setbacks, will miss some dates	Feature production cost or budget increse 1-5%	Minor injuries
4	Intolerable, but there are alternatives	Blocks level 4 features from testing	Feature production cost or budget increse 5-10%	Life impacting injuries
5	Intolerable, no alternative	Blocks critical (L5) features from testing	Feature production cost or budget increse >10%	Fatalities or life-threatening injuries

The gathered features are assigned values in a brainstorming session. In the example Table 8. below, the features are prioritized firstly according to safety-criticality and then on Failure Proneness. For example, number two, machine tool automatically machining a part is a safety-critical feature, which is therefore marked with an X in the safety-critical column and prioritized, but the AGV has been monitored in the past to be highly prone to failure as well as being safety-critical, and is there for prioritized higher.

If other features are heavily dependent on the development of a specific feature, that specific blocker feature could be given priority as well, possibly indicating that need by another column.

Table 8. *Example feature table with prioritization.*

Nr.	Epic	Features	Impact of failure (1-5) I	Likelihood of failure (1-5) L	CP (1-5)	FP (0-5)	RPN (I x L)	Safety-critical	RPN_{CP} (CPxL)
1	AGV	Transport pallet	3	3	5	5	9	X	15
2	Machine tool	Automatically machine a part	5	4	5	1	20	X	20
3	Robot cell	Move part from pallet to CNC-machine	5	5	5	1	25	X	25
4	Order management	Schedule orders	5	3	4	5	15		12
5	Reporting	Monitor efficiency	1	1	2	5	1		2
6	Pallet handling	Pallet transportation	3	4	3	4	12		12
7	Washing machine	Pressure wash	4	5	5	4	20		25
8	Order management	Report order progres	1	2	4	3	2		8
9	Order management	Print orders	1	1	1	2	1		1
10	Order management	Add and remove order	5	2	5	0	10		10

The Table 8. can be rearranged as needed, possibly if some features need to be working before specific critical features. Also, if some of the risks can be mitigated, for example by installing safety features around a robot cell, then its safety-criticality can be reduced.

A “Cutline” can be determined in this list of features if there is a need to leave something untested because of schedule changes. Maybe there is time to only test 80% of the features and the cutline is determined in between features 8 and 9, indicated by a red line in the Table 8.

3.1.6 Testing approach/ Agile Quadrants

This section is concerned with what kinds of testing will be done in the project, describing the necessary testing types for whoever is reading and interested in the testing of the project. The descriptions should answer how the testing is done and who is involved. In an Agile approach, the describing can be done with the Agile Testing Quadrants of automated, automated & manual, manual and tools. The Quadrants were discussed in Chapter 2.2.1 Agile Testing Quadrants and shown in Figure 6.

Quadrant 1 is concerned with technology facing tests that guide development, like Unit Tests and Component Tests. These tests are often linked to the CI-pipeline and are automated. In this quadrant the developers make sure the code works as it should and the software architects make sure the components work together. This quarters explanation should also address whether there are Unit Acceptance Tests or if those are covered in customer demos.

In Quadrant 2 the business-facing and development guiding Functional Testing, and Story Testing should be addressed. These tests are requirement focused, testing possible scenarios, example testing of the workflows and testing user experience with prototypes. This quadrant essentially makes sure all features function as they should together.

Quadrant 3 considers manual testing, such as exploratory, user acceptance, scenario, and usability testing. In this quadrant the tests are business-facing used to evaluate the SUT. The test cases in this quadrant can be written in a TDD-manner, especially in ATDD, as to making it easy to automate them in quadrants one and two. The exploratory testing in Agile can often be done in an iterative fashion in confluence with the development. In this quadrant the Alpha and Beta Testing, also called Factory Acceptance Testing and Site Acceptance Testing should be addressed. How and when are the FAT and SAT held and what is going to be tested in them? Also, User Acceptance Testing should be addressed, if there are customer demos held for the customer, or if there is a demo server in which it is possible for the customer to test things.

Quadrant 4 contains technology-facing tests, evaluating the products non-functional requirements like performance, security, and stability. Performance and stress tests should be explained here. How will they be undertaken, how they test the attributes and are the tests automated somehow.

Also worth to explain in this section is if the development process is a Test-Driven Method, as mentioned in the literature of Chapter 2.3 and its subchapters about TDD, BDD and ATDD. It should be explained here what it means in practise to develop the project with an ATDD approach, for those who are not familiar with it.

3.1.7 Infrastructure and technology needs

Adopted from the test plan template and from Chapter 2.4 Test Automation, in this section the needs for the test environment such as hardware, data, software, facilities, interfaces, security access, and publications are discussed. Is the testing environment as close to the real entity as possible? Used test automation tools should be listed here as well. Question to be answered here include what software's are used in testing, what type of test data is needed, is automated test data generation necessary, are the automated tests on a Continuous Integration Pipeline and what type of servers are used. Is there a virtual demo server for the client which needs to be upheld?

3.1.8 Deliverables from testing

In this chapter the testing deliverables are described, as suggested in the test plan template. Specifications, logs, reports, and test cases are some of the possible deliverables. There can be data about the projects testing status here, such as how many features have test cases and how many features have already been tested. Found issues and reported bugs can be reported here as well if it is convenient. A figure of the bugs found can be a nice addition to communicate the found problems, as illustrated in Figure 13.

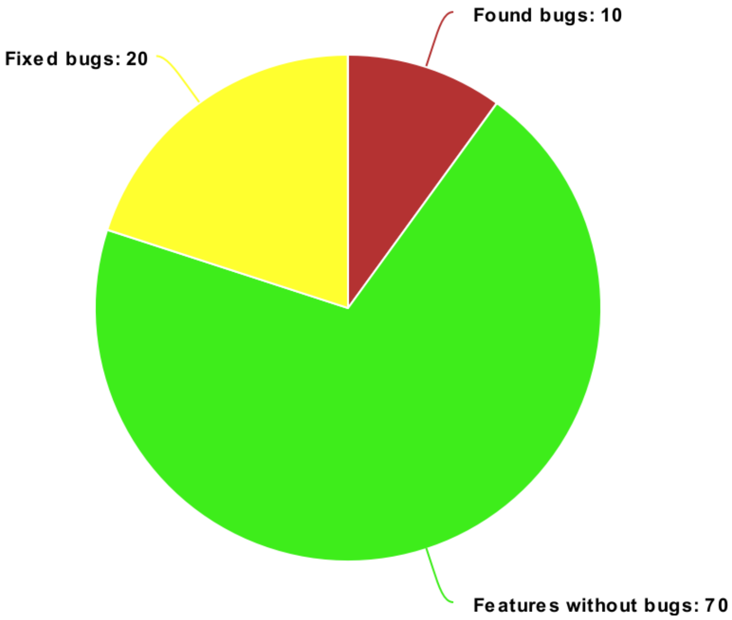


Figure 13. *Bug diagram.*

A list of issues should be provided here, and links to the reported issues would be a worthwhile addition as well. How serious the issues are should also be captured, as this data can be used in future projects in forming the FP-value for feature prioritization.

3.1.9 Personnel and their responsibilities

This section addresses who takes care of what. Possibly a matrix with responsibilities on one side and personnel on other, like illustrated in Table 9, or a RACI-table as described in Chapter 2.5 Test Plan. Also, should contain tasks for testing, meaning needed actions to get ready for testing and for conducting the testing.

Table 9. *Responsibilities.*

	Develop Unit Test Plan	Maintain Test Plan	Automate Tests	Create Test Environment
Test Manager (Jukka)		X		
Development Manager (Luis)	X			
Test Environment Coordinator (Antti)				X
Lead Tester (Jose)			X	

If there are dependencies between the tasks those would be worthwhile to address here. For example, you cannot automate testing without a working test environment.

3.1.10 Schedule

Milestones from Project Plan combined with the Testing Milestones, if those are not already listed in the Project Plan or a Release Plan. At the beginning it can be a schedule without calendar dates, to establish dependencies, identify the time the tasks will take. Can be a Gantt chart to better show the dependencies. An example Gantt chart can be seen in figure 14.

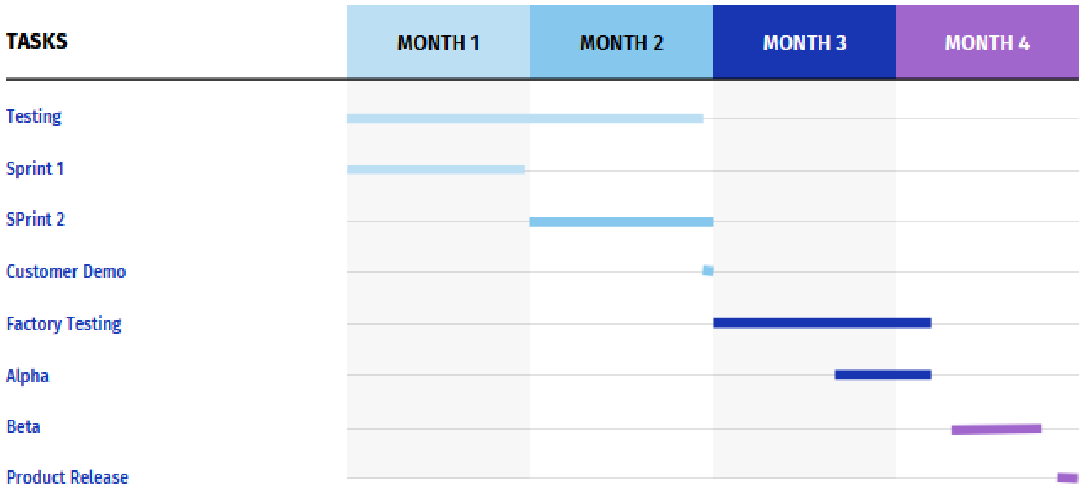


Figure 14. *Schedule.*

The schedule in Figure 14. shows a testing block, which is divided into two sprints, a demonstration for the client, a block for Factory Testing, which includes the Alpha Testing or FAT, followed by Beta or the SAT and the delivery of the product to the customer. Some milestones can be different releases, sprint reviews, test camps, client demonstrations, different factory acceptance testing dates for different parts of the system under test, commissioning, Site Acceptance Tests and Final release of the product to the customer.

4. IMPLEMENTATION

The implementation took the form of an intranet web page titled Software Status, which was under a higher order project page with all the necessary information already in one place. The project that the intranet page was developed for was an MMS project, meaning Manufacturing Management Software, which is a software for production planning and production execution, with automatic optimized production workflow calculations, optimized order-based resource calculations, resource management for automated manufacturing by communicating the part info, NC program, fixture, tooling and material information to the machine tools assigned to specific parts and orders. MMS brings together different data systems and machinery as well as production data and resources. MMS integration possibilities are illustrated in Figure 15.

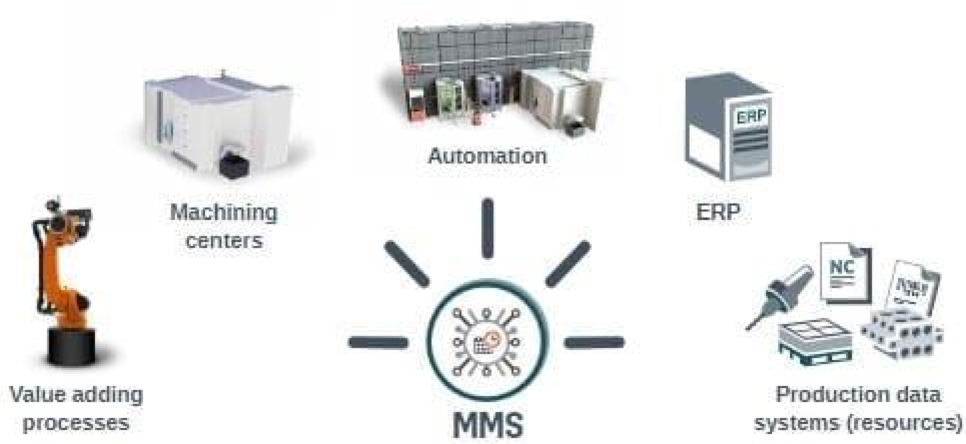


Figure 15. *Manufacturing Management Software (Fastems 2021).*

The page is accessible for all employees at Fastems, and so it was deemed necessary to explain the different testing types so that anyone could understand how the testing is done for the project. The intranet website was considered to be the best option for gathering the different links, schedules and checklist to one place for all to see, as the website can be made to update automatically as the project goes forward. The automatic updates was made possible with Atlassians Confluence, which is a web-based wiki for the company, and Jira, which is an issue tracking and product development tool for bug tracking and Agile management. The two softwares function together so that the intranet site could be inlaid with widgets that monitor the project on Jira according to given guidelines.

At Fastems, the risk analysis and prioritization of epics and features is already done in Jira in a prior stage of project development, so that part of the framework was left out from this intranet implementation. However, the risk analysis and prioritizations potential usefulness was noted, and it was pushed to wait for further development projects, as at the moment the prioritization is done case-by-case, according to customer interests, dependencies, laboriousness and importance.

As in the previous Chapter 3. Proposed Framework, in this chapter the headers are also formed like the headers on the intranet website that is functioning as the Test Plan.

4.1 Introduction

In this first part of the page, introduction part of the site, is a short summary of what is in the Software Status page, to make it easier to better outline the site in a quick glance.

The page consists of the personnel responsible of the testing in the project, a status section for how SW QA, short for Software Quality Assurance, PLC, short for power-line communication and Robo Jira tickets or tasks are progressing, what is the status of different checklists, descriptions about different testing levels, testing infrastructure descriptions, a list of main tailored features of the project, a list of features without test cases, what is the testing status of the project, a testing schedule, and references for further information on the project.

The introduction part was chosen to be the better alternative between it and a table of content, as the Software Status page was required to be a compact site, and so a table of content would take a lot more space compared to an extract of text. Of course, it can be tedious to look for things in text, but it takes less space and offers an outline to the site.

4.2 Personnel

In this section, the people responsible for testing and Quality Assurance, or QA-activities, are listed with their contact details, for easy contacting, and with their titles or areas of responsibility. Further responsibilities are listed in the Checklist Status where the checklists are pinned on people and in the RACI-tables in Testing description.

A list of people with their contact details, titles and responsibilities was deemed a better alternative to a matrix, as there are further areas of responsibility in the Checklist Status table where they can also be seen to be completed or not. Perhaps a matrix would have shown dependencies better and would be faster to find peoples responsibilities, but a list again takes less space, and the Jira tickets, which hold tasks to be done, are assigned to people in the Agile management tool, and so also show who is responsible for them. True to Agile, the site was required to be lightweight with no unnecessary repetition of information that can easily be found somewhere else, but it was concluded to be beneficial for the visitor of the site to find, for example the testers information upfront, if there happened to rise some questions.

4.3 SW QA, PLC and Robo status

In this status section, it is explained how SW QA, PLC, and Robotics Jira tickets are progressing regarding a specific point of view, such as PLC or Robot development kick-off, or status of reviewing PLC or Robot results and implementations.

The statuses can be inserted from Jira to the Confluence page by site macros which allow to sort the projects Jira tickets by different filters. The macro insertion tool and filter format can be seen in Figure 16. In the tool the user can choose which epics are shown by ticking the checkbox. The tool also shows the key corresponding to the epic and a summary to know what the epic is about.

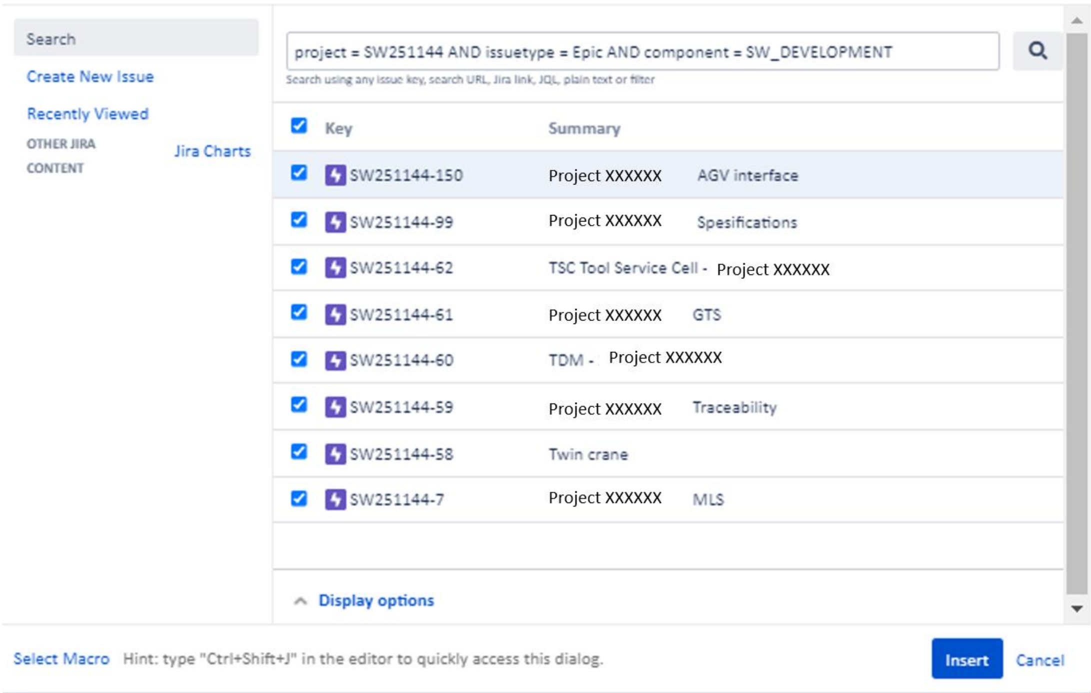


Figure 16. *Macro insertion.*

At the time of writing, the project had only one feature in the SW QA status table of Jira tickets, titled “Test plan for tailored features”, as shown in Figure 17. The distinction for the filter was made by establishing a new software team for the SW QA related tickets, called QA, which allowed the filter to find the ticket with QA as the selected software team. The table shows the key for the ticket, its title or summary, an icon depicting that it is a task, when it was created, updated, and when it is due, who it is assigned to, who reported it, its priority as a symbol consisting of arrows, its status and resolution or the outcome of it. Lists of Jira issues for PLC and Robot development Kick-offs and reviewing statuses were also planned for the Confluence page, but at the time there were no tickets for those.

Key	Summary	T	Created	Updated	Due	Assignee	Reporter	P	Status	Resolution
SW251144-56	Test plan for tailored features		Oct 13, 2021	Oct 14,2021		Antti Assignee	Administrator		REQUEST OPEN	Unresolved

1 issue Refresh

Figure 17. *Caption of Jira tickets for SW QA.*

A widget table for PLC development was however created to give some idea of the situation with PLC development. As seen in Table 10, the table shows an assignee and how many tickets that person has in different statuses, namely request open, resolved, closed, in progress, selected for development and in review. In the table the unassigned tickets are also shown along with the total amount of tickets in different states. A similar table was also created for the development of MMS, to show the software development situation as well.

Table 10. *Example PLC development status.*

Assignee	Status						
	REQUEST OPEN	RESOLVED	CLOSED	IN PROGRESS	SELECTED FOR DEVELOPM...	IN REVIEW	T:
Antti Assignee	1	0	0	1	1	2	**5**
Amelia Assignee	0	1	1	0	0	0	**2**
Ahmed Assignee	11	0	0	0	0	0	**11**
Anni Assignee	26	0	0	0	0	0	**26**
Total Unique Issues:	**38**	**1**	**1**	**1**	**1**	**2**	**44**

Showing **4** of **4** statistics.
View in Jira

This implementation was found to be the easiest to undertake, as all the tickets and tasks are fed into Jira in different phases. With the help of the macros and integration between Confluence and Jira it is very simple to add desired tickets to the confluence site with proper filters. The lists of tickets and tasks also update automatically as they are checked as completed in Jira. A matrix of different responsibilities for SW QA, PLC and Robo implementers would have been extra work with no additional value added and would have needed extra work when updating the lists as tickets are completed, and so, would have not fit the Agile mentality. The status macros provide a quick overview of the project's different areas of development, easy for anyone interested to find and asses without having to do calculations in Jira. A danger hidden in this could be that tasks are forgotten completely or for a time if they do not have the proper tags to which they are filtered by, so the tickets need to be prepared carefully when they are established.

4.4 Checklist Status

In this section a checklist status table is provided, illustrated in Table 11. In the table, there are different phases of testing in the upmost row, namely Factory testing, FAT, installation and commissioning, and lastly SAT. The leftmost column indicates the specific machine or solution that is concerned in the checklists. In this table there are MLS, short for Multi Level System, GTSs, short for Gantry Tool Storages, and TSCs, short for Tool Service Cells. The inner cells of the table then has a link to the Teams documents, a link to Aton, which is a PLM or Product Lifecycle Management tool and holds the actual checklist, and the different checklists listed as to show who is responsible for creating the checklist template, going through the checklist and testing the Robot, MMS, PLC and after assembly and when the articles are due.

The progress of the checklist can then be indicated with the colours shown in the upper left side of the Table 11, indicating with red if the list is not ready, with yellow if it is ready for filling, and with green if the list is filled and stored to Aton. So, the Table 11. also indicates if the testing is done for that part of the project. Also indicated below the main table are the hardware assembly situation for cranes and loading stations in MLS.

Table 11. Checklist status table.

Status:
List not ready
List ready for filling
List filled and stored (aton)

Checklist	Factory testing			FAT			Installation / commissioning			SAT		
MLS	Teams: <LINK>			Teams: <LINK>			Teams: <LINK>			Teams: <LINK>		
	Aton: <LINK>			Aton: <LINK>			Aton: <LINK>			Aton: <LINK>		
	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>
	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>
	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>
	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>
	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>
GTS1	Teams: <LINK>			Teams: <LINK>			Teams: <LINK>			Teams: <LINK>		
	Aton: <LINK>			Aton: <LINK>			Aton: <LINK>			Aton: <LINK>		
	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>
	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>
	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>
	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>
	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>
TSC1	Teams: <LINK>			Teams: <LINK>			Teams: <LINK>			Teams: <LINK>		
	Aton: <LINK>			Aton: <LINK>			Aton: <LINK>			Aton: <LINK>		
	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>
	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>
	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>
	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>
	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>
GTS2	Teams: <LINK>			Teams: <LINK>			Teams: <LINK>			Teams: <LINK>		
	Aton: <LINK>			Aton: <LINK>			Aton: <LINK>			Aton: <LINK>		
	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>
	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>
	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>
	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>
	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>
TSC2	Teams: <LINK>			Teams: <LINK>			Teams: <LINK>			Teams: <LINK>		
	Aton: <LINK>			Aton: <LINK>			Aton: <LINK>			Aton: <LINK>		
	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>	Template	<WHO>	<DUE_DATE>
	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>	Robo	<WHO>	<DUE_DATE>
	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>	MMS	<WHO>	<DUE_DATE>
	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>	PLC	<WHO>	<DUE_DATE>
	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>	Assembly	<WHO>	<DUE_DATE>

HW / Assembly:				
MLS 1 Crane 1	Aton:	<Link>	Teams:	<Link>
MLS 1 Crane 2	Aton:	<Link>	Teams:	<Link>
MLS 1 Crane 3	Aton:	<Link>	Teams:	<Link>
MLS 1 Crane 4	Aton:	<Link>	Teams:	<Link>
MLS 1 Loading Station 1	Aton:	<Link>	Teams:	<Link>

The checklists are organized with letters, such as "A" in the example Table 12. After the identifying letter, there is a description of what the checklist is about, and examples of what to look for. The articles in the checklists are organized in a descending manner, with bigger parts of the whole system written on a blue background and smaller parts of the whole written on white under the main action. There are then instructions for the checks, on blue background there is a description of what the checks are for, and on white the actual checks, for example "visually inspect robot arm". When the checks are completed, they are checked as OK, and the date and initials of the tester are written down. There is also a section for notes, where the tester can write down observations and describe how the test failed if it did.

Table 12. *Example checklist.*

A	Visual Survey of the system	(device surfaces, colours, etc.)				
	Action	**Instructions**	**OK**	**N/A**	**Date/ Initials**	**Notes, observations and NOT OK related descriptions**
	ROBOT	**Devices attached to the robot**				
	Robot	Visually inspect the robot arm				
	Robot	Check robot mounting in case of stationary robot mounting				
	Gripper changer	Check pneumatics and hose markings				
	Dress-pack	Check robot dresspack mechanics				
	Linear Track	**Linear track related check points**				
	Track Mounting	Check track mounting				
	Robot Mounting	Check robot mounting to linear track				
	Track lubrication	Check auto-lubrication device for grease				
	Tooling System	**Inspections for robot-held tools (gripper/db-tool)**				
	Electric Spindle	Check tool mechanical assembly				
	Electric Spindle	Check cabling and hosing routes				

The checklist status table was designed so that all the checklists that are in many different locations could be found easily in one place, with their statuses visible there as well. Through the table it is easy to see which features are ready in PLC in the FAT, which are underway, and which have not yet been designed. This implementation saves the users time from having to look for the files one after the other in company Aton or Teams channels and provides the status at a quick glance.

In the beginning of the development process of this Software Status site, it was considered to scrap the checklist completely and design the FAT and SAT tests on the basis of the tests that are written to Jira. The testcases and features can be arranged in Jira into cycles of testing, so that the specific machines would be tested thoroughly according to the ATDD method of development. The test cycles in Jira would have allowed for easy test camps for thorough testing of the features, so that part was left on the table for future test camp planning. However, it was seen to take a lot more time and effort, and so was left out of this implementation, and the checklist were left as they were. The benefits of the checklists are that they are easy to understand and fill and required no development or planning for their usage, as they were already in place. Further on, Excel is a versatile tool for managing these kinds of checklists and allows fort them to be easily copied for future projects if necessary, which then saves more development time.

4.5 Testing description and responsibilities

In this section there are descriptions of how the project is tested on each level, intended for those that are interested in the details. Also in this section are bits of information on that specific level of testing in question, such as needed meetings, who are needed in those meetings, and action points needed to be taken care of before that specific level of testing. Specialities about the project are also listed in this section, such as what the system consists of, for example, two AGVs and three sets of double robot stations.

In the confluence site the descriptions are set in an info box to better distinguish the part as a separate section of the site. The different levels included in the project and described on the site are Functional Testing, Exploratory Testing, Stress, Performance and Load Testing, Factory Acceptance Testing, Commissioning, and Site Acceptance Testing and their descriptions on the site were written as follows:

1. Functional Testing
    - Functional testing makes sure the functions operate as they should and that the specifications and requirements of the system are met.
2. Exploratory Testing and Manual Testing
    - In Exploratory Testing the testers check system on the fly. Tester’s simultaneously design and conduct the testing with a critical approach to look at the outcomes. Functionalities are tested in an ad-hoc manner.
    - The development is done with ATDD-method and with pull requests to walkthrough and validate the code before manually testing the Acceptance Testcases and accepting the cases as passed, declining them as failed, or marking them as deprecated.
3. Stress, performance and load testing
    - Stress testing checks the stability and robustness of the SUT, going through scenarios with a simulation model.
    - Performance testing checks the speed of the SUT, making sure the performance of the different parts of the SUT is in order by giving different parameters in various load settings.
    - Load testing is the process testing the actual user load on the SUT, looking at how the applications manage during normal and high load settings.

4. Factory Acceptance Testing
    - The Factory Acceptance Test (FAT) evaluates the equipment during and after the assembly by verifying the SUT is built and works according to the design specifications, making sure that the controls and components, PLC and Robotics, are working as they should according to the functional description and the system passes the points in the FAT-checklist.
    - Overview of system devices to be tested on factory floor
    - In case of a robot project, link to layout and robot production follow up
5. Commissioning and Site Acceptance Testing
    - Site Acceptance Tests are carried out at the customers site to verify the systems specifications meet the customers’ requirements, as mimicking all site conditions at the factory floor is not possible. The SUT is taken through real production, following the process with all the actual steps and parameters, making sure the system passes the points in the SAT-checklist.

The different parts of the previous list also needed action points and tasks for planning, execution, and finalization, so that it can be easily seen who should do what. The RACI-table came to be the solution to that. Similarly to the responsibilities in Table 2. in Chapter 2.5, in the RACI-table of Table 13. the rows state the tasks to be accomplished, and the columns point to the roles, with additionally holding the phases for Planning, Execution and Finalizing. The table also explains the meanings of the letters in the table. The example Table 13. is somewhat compact, as to not reveal too much of the company’s activities, and for that same reason there is only the example for FAT, and not for the other four parts.

Table 13. *RACI-table.*

	Project Manager	Lead developer	QA Lead			
Activity		**Planning**				
Plan Factory Testing	A	C	R			
Verify FAT plan	A	I	R			
Activity		**Execution**			R	Responsible
FAT kick-off	A	I	I		A	Accountable
Perform FAT	I	R	A		C	Consulted
Activity		**Finalize**			I	Informed
Verify test results	A		R			

This implementation was chosen because there was a need for showing light on the testing practices in the company. There were no descriptions, however brief, anywhere to be found, so the descriptions were seen to be a good addition to the projects Software Status site. On the Confluence site anyone who is interested can read the brief descriptions of the testing that is done, to get an idea of the proceedings. A list was seen as a good format for this at it again shows the tests in chronological order. The descriptions may seem a little out of place here, and maybe in the future they will be sifted to some other location that makes more sense. However, the kick-off planning, action points, overview of system devices for factory floor and robot project layouts in the FAT-part would be good points to preserve in the case of relocation.

The Agile Testing Quadrant approach proposed in Chapter 3.1.6 Testing approach/ Agile Quadrants, was left out of this implementation, as not enough parts of the quadrants were in place, and so it would have served no purpose to have an empty quarter explained.

The RACI-table was chosen because it streamlines the processes by making sure all team members and stakeholders know their roles and tasks. Also, the RACI-table gets rid of confusion, keeps the project on the right course, makes sure every task is done and enhances communication between the participants, making the transition of tasks easier.

4.6 Infrastructures

In this section any infrastructure needs, tools for automatic and manual testing and software's utilized in testing are listed. What is in this section heavily depends on the project, as custom projects vary immensely, and at this point every project is not tested at all, as some are tailored but lightly from the main product. If this section is applicable to the project in question, the QA-team member who is filling the Software Status confluence page should seek to answer some questions in this section: How is the test data generated? Is there a client server, and where to find it? Is there test automation for the project?

At this point of the project in question, it was not certain if there would be any of the previous things listed, so this section was left as a place holder for the future and other projects, but still holds a list of the things that should be explained in the section. For example, it is listed here if there are virtual servers for the clients use, the testing environment that is used, a test data generator, test automation if there is any, CI-pipeline, and necessary servers. The CI-pipeline holds tests for the Vanilla project, from API to Smoke to Acceptance tests, but for the project the pipeline checks that the environment is built successfully.

It was discussed that test automation should be the future for testing all the projects, but at the moment it is not possible and requires more research and resources to achieve. The working hours, finances and resources required for test automation are not met at the moment, but as the Agile Development method is driven forward in the company, test automation is sure to expand from the Vanilla project to others, as is described by the Agile Quadrants in Chapter 2.2.1 Agile Testing Quadrants.

This section was seen to be a useful addition for those who do not know about the tools used in testing, and as testing and SW QA gains foothold in the company, it is good to show the tools used.

4.7 Project main tailored features/Epics and features without test coverage

In this section the new features, also known as tailored features, are listed. The features are those that are specifically developed for that specific project and differ from the Vanilla solutions. Similarly, to Figure 17. in Chapter 4.3 SW QA, PLC and Robo status, the table captioned in Figure 18. shows, from left to right, the key for the ticket, its title or summary, an icon depicting that it is an Epic, when it was created, updated, and when it is due, who it is assigned to, who reported it, its priority as a symbol consisting of arrows, and its status and resolution or the outcome of it. The dates, assignees, reporters, project, and customer names have been edited.

Key	Summary	T	Created	Updated	Due	Assignee	Reporter	P	Status	Resolution
SW251144-150	Project xxxxxx AGV interface		Oct 1, 2022	Oct 9, 2022		A. Assignee	A. Reporter		REQUEST OPEN	Unresolved
SW251144-99	Project xxxxxx Spesifications		Oct 2, 2022	Oct 10, 2022	Oct 28, 2022	A. Assignee	A. Reporter		REQUEST OPEN	Unresolved
SW251144-62	TSC Tool Service Cell - Project xxxxxx		Oct 3, 2022	Oct 11, 2022		Unassigned	A. Reporter		REQUEST OPEN	Unresolved
SW251144-61	Project xxxxxx GTS		Oct 4, 2022	Oct 12, 2022		A. Assignee	A. Reporter		REQUEST OPEN	Unresolved
SW251144-60	TDM - Project xxxxxx		Oct 5, 2022	Oct 13, 2022		B. Assignee	A. Reporter		REQUEST OPEN	Unresolved
SW251144-59	Project xxxxxx Traceability		Oct 6, 2022	Oct 14, 2022		A. Assignee	B. Reporter		REQUEST OPEN	Unresolved
SW251144-58	Twin crane		Oct 7, 2022	Oct 15, 2022		B. Assignee	B. Reporter		REQUEST OPEN	Unresolved
SW251144-7	Project xxxxxx MLS		Oct 8, 2022	Oct 16, 2022		B. Assignee	A. Reporter		REQUEST OPEN	Unresolved

8 issues Refresh

Figure 18. *Caption of the Main tailored features table.*

The list of tailored features shown in Figure 18. came from the Macro insertion tool shown in Figure 16. by giving the filter as follows: “project = “SWXXX” AND issuetype = Epic AND component = SW_DEVELOPMENT”. The string filters the specific project from other projects, selects the issuetype as Epic and component as software development asto filter out the robot and PLC epics.

A similar table was created for the features that do not yet have test case coverage by giving the Macro insertion tool a filter as follows: “project = "XXXX" AND issuetype in (Feature) AND issue not in hasTestCoverage() AND component = SW_DEVELOPMENT”. The resulting table had the same attributes as Figure 18, with the difference of holding the features that have no test cases designed for them yet.

The implementation was deemed to be the best solution as it shows what the new features are that are going to be developed and tested in the project, as well as their statuses and who they are assigned to. It is also possible to go to the Jira tickets straight from the Confluence site by clicking the “Summary” of the ticket, for further information. The features without test coverage was seen to be a good addition as well, because it provides an overview of the features still under work. A pie chart was also considered for the features that have no test coverage, but it was thought that beign able to access the tickets from the Confluence site had its own value, as well as seeing the names of the features to create an image of the test coverage situation per feature of the project.

4.8 Testing status

In this section of the Confluence site, a status of the testing is provided by different indicators, inspired by Chapter 3.1.8 Deliverables from testing. Status is shown by test execution results and by test cases by status as in Figure 19, and by bugs per priority and found issues in Figure 20. There is also a link to Jira for a "Test cases by coverage" -list, which would have been too long for the site, but it lists the features that have test cases and shows how many test cases the feature has in what status and how many cases in total. The test case status can be Draft if the test case is ready for approval, Deprecated if the test case has become obsolete, or Approved if the test case is approved and ready for testing. Further on, at the bottom the total number of test cases for Draft, Deprecated, and Approved can be seen as well as the total number of test cases for all the features.

The pie charts in Figure 19. illustrate the test execution results on the left and the test cases by status on the right. The test execution results pie chart shows the number of passed, failed, blocked and not executed test cases for the project and the test case by status pie chart shows the same data as in the Jira list, but in a format that compliments it and is easier to read in a quick glance at the Confluence site.

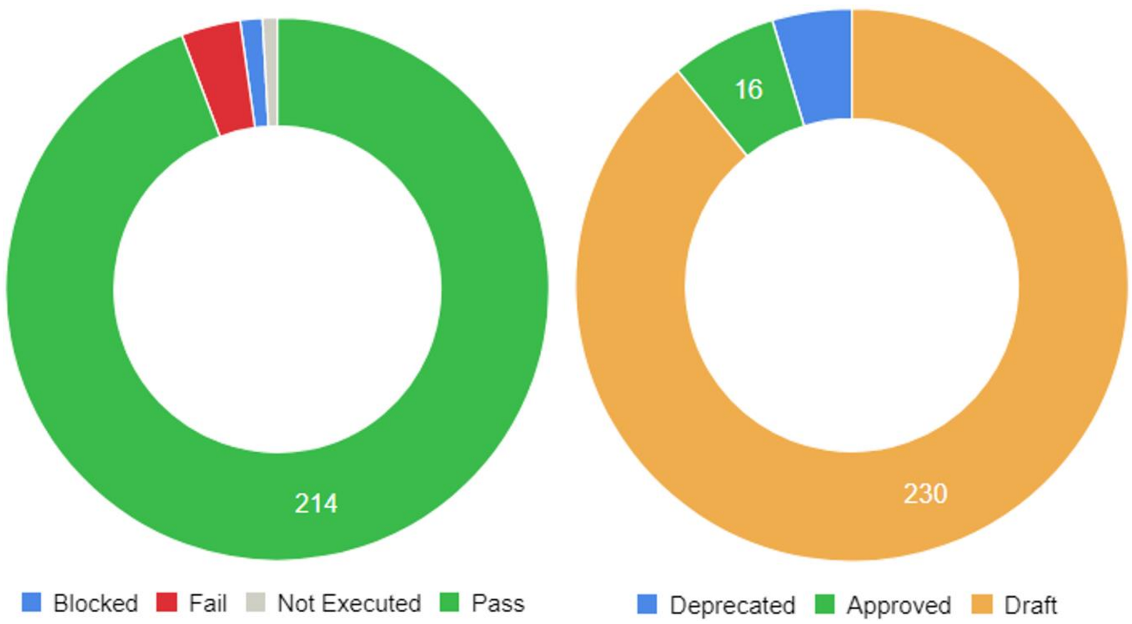
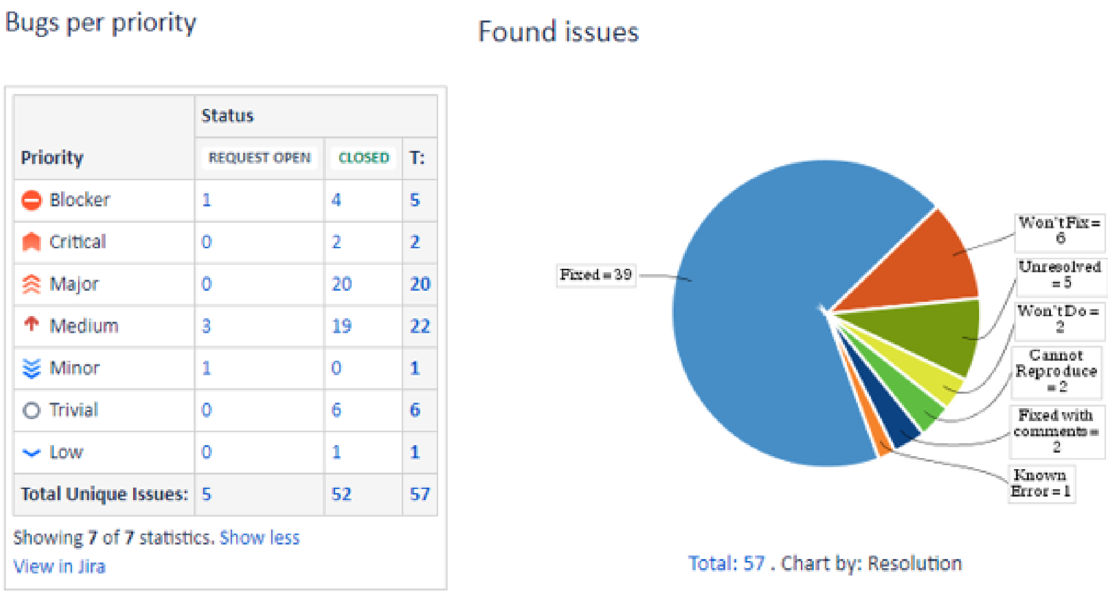


Figure 19. *Test execution results and Test cases by status.*

In this section, some bug and issue statistics were charted as well. Similarly to Figure 13. Bug diagram in Chapter 3.1.8, in Figure 20. there are the widgets for bugs per priority on the left and found issues and their resolutions on the right. The bugs per priority shows the priority of the bugs reported, ranging from minor and trivial to critical and blocker, their statuses of request open and closed, and the number of bugs per each status and priority class as well as a total bug count. The found issues chart shows how many issues are in what state, and what has been their resolution. Most of the bugs were fixed at this point while others were waiting for their resolutions and others have been left untouched, whether they will not be fixed, if the issue could not be reproduced or the error was already known.



Bugs per priority

Priority	Status		
	REQUEST OPEN	CLOSED	T:
Blocker	1	4	5
Critical	0	2	2
Major	0	20	20
Medium	3	19	22
Minor	1	0	1
Trivial	0	6	6
Low	0	1	1
Total Unique Issues:	5	52	57

Showing 7 of 7 statistics. Show less
View in Jira

Figure 20. *Bugs per priority and found issues.*

This implementation for the Software Status site was chosen, because with the list of Figure 19, it is easy to pinpoint if some and what test cases still need to be approved before testing can proceed. The pie charts of Figure 19. offer an overview of how testing is going and how the test case development has been going forward, and if there is still a lot of work to be done. The statistics offered by Figure 20. give valuable information about the bugs that have been reported for a project. As can be seen, five bugs were blocker issues that stop the testing of other features or production completely and so were valuable finds, as well as the other 39 issues that were fixed already.

4.9 Schedule

In this section the schedule related to testing is presented in the form of a gantt-chart. The chart, illustrated in Figure 21. features three lanes, or rows, titled Software design and testing, Factory testing and Site testing. The different activities and dates were looked up from the project plan, the different checklists for FAT and commissioning and release plan. Some testing related dates and activities were filled and scheduled into this schedule only, as test camps and client demonstrations were not scheduled into other plans yet.

The Software design and testing phase was divided into releases of smaller proportions, to make the job easier to approach. The small yellow boxes with the letter T resemble test camps, in which the features developed in the release are vigorously tested by multiple testers in the client demo server, which simultaneously prepares the client demo server for the client demonstration that follows.

The Factory testing phase overlaps with the third release of the Software design and testing phase, with the different FATs starting during the third release. The long Factory testing block that includes the smaller and reaches over the FATs to allow time for fixes and retesting for noticed defects.

The last lane, titled Site testing, holds the installation, commissioning and the SAT, with the whole schedule finishing at the DS, or delivery to service, meaning the development team hands the project to the service team, which will answer to further bugs and defects reported by the client.

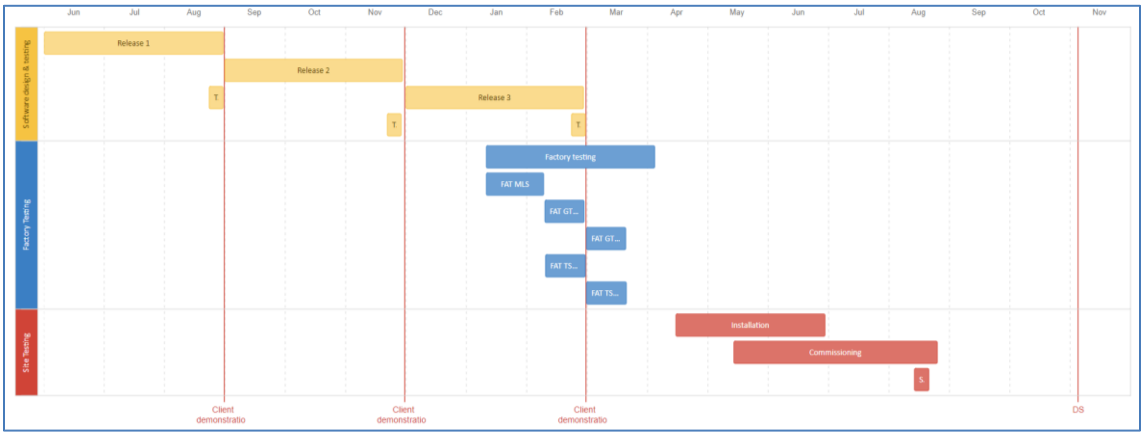


Figure 21. *Schedule Gantt-chart.*

A guiding text of “Fill a schedule, if DO has not planned testing related happenings in project release plan, test camps, etc.” was added to the top of the chart, to emphasize that there is no need for this if the DO has already addressed testing related events in the projects release plan. The schedule was seen as a good addition, if indeed the DO has not yet scheduled the test camps, because it is good to have a schedule even if the dates are not yet set in stone. Having a schedule makes the steps needed to achieve the goals clearer and easier to achieve in a timely manner and keeps every party on the same page, avoiding surprises of things that should have been done, but at the time of noticing there is no time for them to be completed. Deadlines regarding for example test camps and client demonstrations allows for prioritizing the right tasks, avoiding the situation that some feature should be demonstrated, but it has not been tested and it is not merged to the client server for the demonstration. As in case of the FATs in the example of Figure 21, the schedule adds contingency time for fixing problems that occur as well.

4.10 Related plans and schedules

In this section there are links to related plans and schedules. In this example they were chosen to be the Release Plan, the Factory Testing Plan and the Assembly Schedule.

As discussed in chapter 2.6.1. Agile Test Planning and Iteration Planning, the Release Plan is a document detailing the features for an upcoming release and tracking them. The document is mostly utilized by the development team and the Delivery Owner, dealing with resource allocation, development team members tasks, development phases and milestones of the project. The Factory Testing Plan holds a timetable for the factory testing, when specific checklists are going to be run through for specific devices. The Assembly schedule details when specific pieces of the delivery are going to be assembled on the factory floor, when they are going to be tested and when they will be dispatched for the customers facility.

Adopted from Chapter 3.1.2. References, this section was deemed a good addition because it makes it easier to find documents that are relevant to the project, as the relevant documents are listed and linked here to avoid having to spend hours searching in company databanks for the info.

5. CONCLUSIONS

In Chapter 5.1 Results the results of the thesis are examined regarding the research questions posed. The results chapter also works to sum up Chapters 3. Proposed Framework and 5. Implementation. In Chapter 5.2 Discussion there is further discussion on the results and what could be possible future topics of research regarding this thesis.

5.1 Results

The goal of this thesis was to develop a comprehensive framework for testing an industrial automation project, gather all the testing related information into one location and to develop a way to make certain the system was ready and thoroughly tested before dispatching it to the customer. The research question formed from the previous were the following:

4. How a comprehensive framework for Agile testing can be designed and implemented for an industrial automation system?
5. How to define the most critical features that need to be prioritized in testing of an industrial automation system?
6. How to be certain the industrial automation system is complete and thoroughly tested for dispatching to the customer?

The first research question can be answered with the proposed framework of Chapter 3 and the implementation of Chapter 4. The framework was designed to be lightweight, with only necessary parts adapted from the IEEE Standard 829–1998 for Software Test Documentation to avoid unnecessary repetition, and as nonspecific to allow its implementation in different projects by necessary modifications. The framework consists of and introduction, references, scope, test items, risk analysis and feature prioritization, testing approach and Agile Quadrants, infrastructure and technology needs, deliverables from testing, personnel and their responsibilities, and a schedule.

The implementation then morphed into something different, as major projects have different needs. The implementation took the form of an intranet site titled Software Status and had a focus on the testing and development status of the project, as in Agile ATDD the development and testing go hand in hand. The site functions as a place to monitor the progress of the project using the Checklist status, SW QA, PLC, Robo and Testing statuses, as well as an automatically updating Test Plan. The site also functions a databank, having all the testing related information in one place for a project. The differences of the proposed framework and the implementation can be seen in Table 14. By the table it can be seen that the framework fit the requirements of Fastems with slight adaptations, like leaving the risk analysis out. Testing description holds the scope, testing approach and personnel responsibilities for different areas of testing in the form RACI-tables, while the responsibilities can also be seen from the SW QA, PLC, Robo statuses and Checklist Status. The Test items also got divided into main tailored features and features without test coverage.

Table 14. *Comparison of Framework and Implementation.*

Framework		**Implementation**
Introduction	→	Introduction
References	→	References
Scope	→	Testing description
Testing approach/ Agile Quadrants	→	SW QA, PLC and Robo status
Personnel and their responsibilities	→	Personnel
	→	Checklist Status
Test items	→	Project main tailored features/Epics
	→	Jira features without test coverage
Risk Analysis	✗	
Infrastructure and technology needs	→	Infrastructures
Deliverables from testing	→	Testing status
Schedule	→	Schedule

The second research question was answered in Chapter 3.1.5 Risk Analysis. The most effective method for defining the most critical features for prioritization in an industrial automation system was found to be with the use of a historical data-based Failure Proneness value. The second most favourable was the use of a customer assigned priority value with a likelihood score and the last a Risk Priority Number assigned with an impact of failure score and the likelihood score.

The third research question got its answer from the framework's sections of Personnel and their responsibilities, the Deliverables from testing and the implementations sections of Checklist status, SW QA, PLC and Robo status and Testing status. With the widgets updating automatically from Jira it can be easy to see from the site if the software testing of the project is finished, if there are still some software features untested or still lacking test cases, if there are some bugs still unfixed and not yet retested and if some tickets and features are still in development by the PLC, Robo or MMS development teams. With the help of the Checklist status table, it is easy to see what the situation with the checklists is. When the whole Checklist status table is coloured green, all the features in the PLC, Robo and MMS teams Development status tables are either resolved or closed, the list of features without test coverage is empty and the Test execution results chart shows no tests that are not executed, the project is complete and thoroughly tested.

5.2 Discussion

As the implementation of the Software Status site is new, how it plays out in the future and what benefits or negative effects it brings were left out of the scope of this thesis. Future work on this could be researching how easy the framework is to adapt to other Agile software projects, surveying the usability, benefits, and negatives of the framework by questionnaires. As every organization and project is different and has different needs regarding testing due to differences in size, skills of the team, resources, and the System Under Test, the framework may need to be altered to fit the needs of the project in question. The framework can be expanded in future research projects to better cater to the vastly differing requirements.

Something worth looking into would be how defect data can be collected from projects for future use. Data that tells in what type of features or epics defects are found most often, and how serious the defects are, could be used in Failure Proneness prioritization of the testing and possibly prioritizing the development of features, as in Agile methodology, development and testing often go together. As the test case prioritization and software risk analysis of Chapter 3.1.5 Risk Analysis did not make it to the implementation of this thesis, it would be interesting to see how it would work in practice. Statistics such as how it affects the number of defects found, customer satisfaction and ROI would be interesting to see and so would be a good subject for future research.

Further research also needs to be put into test automation of the projects. Questions like what feature test should be automated, which features are worth the time to automate, and what would be the ROI of automating testing for features. An easy starting point for automating could be so called monkey testing in which the GUI is clicked randomly and given random inputs. When test automation is at a usable point in the projects, a widget from the CI-pipeline to the Software Status site could be added to show how the automated tests are going for that project.

REFERENCES

ADZIC, G., 2011. *Specification by Example: How Successful Teams Deliver the Right Software.* New York: Manning Publications Co. LLC.

AGARWAL, B., GUPTA, M. and TAYAL, S.P., 2009. *Software Engineering and Testing.* Jones & Bartlett Learning.

AHMED, A., 2010. *Software testing as a service.* 1 edn. Boca Raton, Fla: CRC Press.

ANSI and IEEE, 1983. *ANSI/ IEEE Std 729-1983: IEEE Standard Glossary of Software Engineering Terminology.* IEEE.

BARESI, L. and PEZZÈ, M., 2006. An Introduction to Software Testing. *Electronic notes in theoretical computer science,* **148**(1), pp. 89-111.

BECK, K., 2001. ***Manifesto for Agile Software Development***
.

CHOPRA, R., 2018. *Software Quality Assurance.* Mercury Learning & Information.

COX, J.,Louis Anthony, 2008. What's Wrong with Risk Matrices? *Risk analysis; Risk Anal,* **28**(2), pp. 497-512.

CRAIG, R.D. and JASKIEL, S.P., 2002. *Systematic software testing.* Boston: Artech House.

CRISPIN, L. and GREGORY, J., 2009. *Agile testing : a practical guide for testers and agile teams.* Upper Saddle River, NJ: Addison-Wesley.

FASTEMS, 2021-last update, MMS –
Manufacturing Management Software. Available: https://www.fastems.com/offering/mms/.

GALIN, D., 2018. *Software Quality: Concepts and Practice.* 1 edn. Piscataway: Wiley.

GMI, 2019. *Software Testing Market Size*
.

GSMA, 2018. *The Mobile Economy Europe 2018*
.

IEEE, 2013. *ISO/IEC/IEEE 29119-3:2013(E): Software and systems engineering Software testing Part 3:Test documentation.* IEEE.

IEEE, 2008. *IEEE Std 829-2008: IEEE Standard for Software and System Test Documentation.* IEEE.

IEEE, 1998. *IEEE Std 829-1998: IEEE Standard for Software Test Documentation.* IEEE.

IEEE, 1984. *IEEE standard dictionary of electrical and electronics terms.* 3 edn. New York, NY: Institute of Electrical and Electronics Engineers.

JONES, C. and BONSIGNOUR, O., 2011. *Economics of Software Quality.* 1 edn. S.I: Pearson Education.

LAPORTE, C.Y. and APRIL, A., 2017. *Software quality assurance.* 1 edn. Hoboken, New Jersey: Wiley-IEEE Computer Society, Inc.

LATORRE, R., 2014. A successful application of a Test-Driven Development strategy in the industrial environment. *Empirical software engineering : an international journal,* **19**(3), pp. 753-773.

LEHTIMÄKI, T., 2006. *Ohjelmistoprojektit käytännössä.* Helsinki: Readme.fi.

LIAO, Z.Z., WU, Z.Q. and LI, J.Z., 2013. Keyword Driven Automation Test. *Applied mechanics and materials,* **427-429**, pp. 652-655.

MERIEM, A. and ABDELAZIZ, M., 2019. Combining Model-Based Testing and Failure Modes and Effects Analysis for Test Case Prioritization: A Software Testing Approach. *Journal of computer science,* **15**(4), pp. 435-449.

MOE, M.M., 2019. Comparative Study of Test-Driven Development TDD, Behavior-Driven Development BDD and Acceptance Test–Driven Development ATDD. *International Journal of Trend in Scientific Research and Development,* **3**(-4), pp. 231-234.

MOISIADIS, F., 2000Prioritising use cases and scenarios, *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Pacific 2000* 2000, IEEE, pp. 108-119.

MYERS, G.J., BADGETT, T. and SANDLER, C., 2012. *The art of software testing.* 3. ed. edn. Hoboken, NJ: Wiley.

NIST, 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology,* .

OSTRAND, T.J., WEYUKER, E.J. and BELL, R.M., 2004. Where the bugs are. *ACM SIGSOFT software engineering notes,* **29**(4), pp. 86-96.

PATTON, R., 2005. *Software testing.* 2 edn. Indianapolis, IN: Sams Pub.

PERRY, W.E., 2006. *Effective methods for software testing.* 3 edn. Indianapolis, IN: Wiley.

RAFI, D., MOSES, K., PETERSEN, K. and MÄNTYLÄ, M., 2012. *Benefits and limitations of automated software testing: systematic literature review and practitioner survey.* IEEE Press.

SAWANT, ABHIJIT, PRANIT BARI and PRAMILA CHAWAN, 2012. Software Testing Techniques and
Strategies. *International Journal of Engineering Research and Applications,* **2**, pp. 980-986.

SINGH, Y., 2012. *Software testing.* Cambridge ;: Cambridge University Press.

SORQVIST, L., 1998. *On poor quality costing. Doctoral thesis, Royal Institute of Technology, Department of Production Engineering, Stockholm..*

SRIKANTH, H. and BANERJEE, S., 2012. Improving test efficiency through system test prioritization. *The Journal of systems and software,* **85**(5), pp. 1176-1187.

SRIKANTH, H., BANERJEE, S., WILLIAMS, L. and OSBORNE, J., 2014. Towards the prioritization of system test cases. *Software Testing, Verification and Reliability,* **24**(4), pp. 320-337.

SRIKANTH, H., CASHMAN, M. and COHEN, M.B., 2016. Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study. *The Journal of systems and software,* **119**, pp. 122-135.

SRIKANTH, H., HETTIARACHCHI, C. and DO, H., 2016. Requirements based test prioritization using risk factors: An industrial study. *Information and software technology,* **69**, pp. 71-83.

SYED, K. and KUMAR, P., 2011. Software Testing – Goals, Principles, and Limitations. *International Journal of Engineering Science & Advanced Technology,* **1**(1), pp. 52-56.

YE, W., 2013. *Instant cucumber BDD how-to: a short and quick guide to mastering behavior-driven software development with Cucumber.* 1 edn. PACKT Publishing.