Master's Thesis (YAMK)

Master's Degree Programme in Technological Competence Management

2023

Lasse Raakavuori

Software Development Project Overview from Management Perspective

– Case: Maintenance Management Software

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Opinnäytetyö (YAMK) | Tiivistelmä

Turun Ammattikorkeakoulu

Teknologiaosaamisen johtaminen

2023 | 100 sivua

Lasse Raakavuori

Yleiskatsaus Ohjelmistokehitysprojektista Projektipäällikön Näkökulmasta

- Case: Huollon hallinnan ohjelmisto

Tämän opinnäytetyön tarkoituksena on tarjota yleiskatsaus projektinhallinnallisiin asioihin ohjelmistokehityksen kentällä. Kohdelukijakuntaa ovat erityisesti henkilöt, joilla on kokemusta projektinhallinnasta perinteisen vesiputousmallin muodossa, mutta joille ohjelmistokehityksen erityispiirteet eivät ole vielä tuttuja. Aihe on rajattu yleisiin asioihin, metodologioihin, kehitystiimin rooleihin ja kehitysprosesseihin. Teoreettisen viitekehyksen tärkeimpänä lähdemateriaalina toimivat IEEE:n (Institute of Electrical and Electronics Engineers) valitut julkaisut. Teoriatietoa sovelletaan ja verrataan käynnissä olevan case-projektin toimintojen sekä kohdattujen ongelmien kuvauksilla. Näiden kahden materiaalin synteesi tuottaa myös käytännönläheisiä johtopäätöksiä sekä neuvoja ohjelmistokehitysprojektin hallinnan kehittämiseksi ja joidenkin haasteiden välttämiseksi.

Asiasanat:

Software development, software project management, software development life cycle (SDLC), case-study, Project Manager, agile methods, development team, scrum, development processes.

Master’s Thesis | Abstract

Turku University of Applied Sciences

Master’s Degree Programme in Technological Competence Managemet

2023 | 100 pages

Lasse Raakavuori

Software Development Project Overview from Management Perspective

- Case: Maintenance Management Software

This thesis is meant to provide a general overview of project management-related issues in the field of software development. The target audience are people who have project management experience in conventional waterfall projects, but are not yet familiar with the specifics of agile software development. The subject matter is limited to general aspects, methodologies, team roles and development processes. The main source material for the theory part is selected IEEE (Institute of Electrical and Electronics Engineers) publications. The theory is applied and compared with descriptions of the activities from an ongoing software development case project as well as the challenges it has faced. The synthesis of the materials leads to pragmatic conclusions and advise to help develop software development management as well as avoid some of it’s pitfalls.

Keywords:

Software development, software project management, software development life cycle (SDLC), case-study, Project Manager, agile methods, development team, scrum, development processes.

Contents

List of figures

Glossary

Asset	An item of value/use in the pursuit of value creation. Asset produces information, has a function, and has recorded data.
Backlog	A list of broken-down tasks needed to be done in order to create the product. (Cobb. 2015, 40).
Classification Society	An independent party, owning a set of standards, which need to be adhered to in order to receive certificates of compliance and classifications.
Feedback Loop	A tool for refining a process, using the analyzed output of a previous cycle as the input for the next.
KPI	Key Performance Indicator
OpEx	Operating expenses a company incurs to keep its business running (Ross 2022).
Scope creep	The unintended growth of the project’s scope during software development projects by either adding items or finding out that certain items are more complex than expected.
SDLC	Software Development Life Cycle.
Technical debt	A term meant to describe the costs of later reiteration of a solution, which was done in a simplistic way earlier.

Test-Driven Development	An approach with a main idea that testing aspects of the units in development are considered and executed already during production, not after it and that constant testing in various forms is a part of the team's toolbox. TDD tends to find defects earlier, and can refine code from the beginning, as it is written to be able to handle the test cases (IEEE 2014, 76).
Velocity	The combined amount of quantifiable work a development team can implement in a single sprint.
Waterfall	A conventional model of project management, which views project phases as linear and emphasizes planning and minimizing deviations.

1 Introduction

The subject of this thesis was chosen because of the need to increase knowledge in the field of software development is emergent in the author's company. The need is dual. Firstly, there is a need for rapid adaptation to a new business environment. Adaptation is meant to be achieved by gathering information from the field. This thesis aims to collect information from established publications that present industry standards from the IEEE (Institute of Electrical and Electronics Engineers), such as SWEBOK and ISO/IEC 29119, which can be considered as compilations of general practices in software development. The information is supplemented with literature to provide an additional layer of depth. Secondly, a basic process guideline is needed for similar future endeavours. The direction chosen here is to combine pragmatic data from an ongoing case project with the theoretical body of knowledge. The aim is to show how theoretical guidelines have been translated into practical work. The project in question is a software development project taking place in a company that has previously focused on design & engineering projects. As the project proceeded, some pitfalls were found. These pitfalls express the dissonance between theory and practice and their presentation in the form of descriptions and tips is essential for the usefulness of this thesis.

Along with strategic digitalization efforts, projects similar to the one described in this paper will undoubtedly emerge. It is therefore essential that the company's knowledge base is enriched by studies such as this. The focus of this thesis is on the Project Manager's perspective. The content is intended to provide a curated summary of the most important issues concerning the Project Manager's skills during a software development project and to present real-life examples of how the presented issues have been handled in an actual project. The content of this thesis also provides top management with high-level information on the organisational and resource requirements of software development projects.

Most companies have project execution guidelines, but they often arguably consist of several versions due to different business areas and requirements. It is not possible to create a comprehensive, all-encompassing project guide, except at a high level. Similarly, it is not possible or feasible to have the same ambition towards the software development process. The aim of this thesis is therefore to maintain a level of generality that is broadly applicable to software development projects. Questions through which the above mentioned goals are meant to be achieved are: What is involved in a typical B2B software development project? What issues are relevant from a Project Manager's point of view? How is the theory implemented in a real-life project? What can go wrong? After reading this paper, the Project Manager should have a framework in place for understanding the characteristics of software development, deepening his/her knowledge and avoiding some probable beginner's pitfalls.

1.1 Case-example's company

The author's employing company (referred to as "company") is a design & engineering office, operating in multiple locations and industries. company's main project types are:

- Design projects (Engineering)
- EPC/EPCM/Turn Key -projects (Engineering, Procurement, Construction, Management)
- R&D projects (Research and Development)

The company has previously been involved in software development, but overall the business is marginal and the market share is small. The internal staff involved in software development are scattered and no guidance documentation has been produced from previous projects. On this basis, it can be argued that the company's competence in software development is low.

1.2 Case-project background

The company has considerable expertise in ship design and works almost exclusively for shipyards that have a shipbuilding contract with a shipowner. Apart from small consultancy projects, the company is not in very close direct contact with shipowners, but acts as a third link in the chain. The strategic direction is to broaden the range of services in order to develop closer relationships with shipowners. This requires the company to develop solutions for servicing ships after they have been built and are in service.

Irrespective of the shipowner's customer profile (cruise ships, cargo ships, government and military vessels, etc.), ship maintenance constitutes a portion of the total operating expenses (OpEx). In addition, if a vital piece of equipment on a ship fails mid-voyage due to lack of maintenance, the cost of such a failure can be disproportionately high. It is imperative for shipowners to use some form of maintenance management software (see Chapter 2) to meet regulatory requirements and to eliminate the risk of unexpected failures. On the other hand, proper maintenance management will improve the performance of the vessel and extend its life. Ship maintenance software is a relatively large market with an estimated value of USD 1.71 billion in 2022 and an estimated annual growth rate of 11.3% between 2022 and 2030 (Infinity Business Insights 2023, 19).

It was decided that the software development project would be carried out in partnership with an experienced software development company (referred to as the partner). The rough division of roles was that the company was responsible for requirements management and overall project management, while the partner was responsible for developing the actual product according to the given scope and requirements. It can be thus argued that the company might have been more strongly involved in the development than is regular.

1.3 Assumptions and limitations

This thesis builds upon conventional design & engineering project management, so the basics of a project manager's role and responsibilities in a waterfall-type project are not elaborated.

Software development project attributes in this thesis are compared with a waterfall-type project. Typically, a waterfall approach emphasizes pre-planning and a negative stance towards changes along the way. In waterfall, it is also common that the work is done in large, concurrent parts. Often the output of the previous part is needed as the input of the next. In a waterfall model, the customer gets to give feedback at a very late stage.

A significant amount of decisions regarding the issues highlighted in this thesis are based on IEEE publications. IEEE is the world's largest technical professional organisation with a portfolio of over 1000 industry standards. (ieee.org 2022.) The main focus is SWEBOK (a guide to Software Engineering Body of Knowledge). It is a comprehensive collection of peer-reviewed descriptions of software development. Although the publication is starting to age, the topics and practices described in it can arguably still be considered a baseline in 2023. Another valuable and more recent IEEE publication cited in this thesis is the standard 12207:2020 Systems and Software Engineering: Software Life Cycle Processes.

This thesis only examines the processes involved in the software development life cycle (SDLC) as defined by IEEE. The processes of SDLC are: requirements, design, construction and testing. The entire software life cycle, called Software Product Life Cycle, or SPLC, is longer and more multi-sectional, and also includes post-development activities, i.e.: deployment, maintenance, support, and retirement processes (IEEE 2014, 151).

2 Maintenance management software overview

This chapter provides an overview of the nature of the software being developed in the case project. The information presented here is intended to clarify the scope and complexity of the project, which will help to provide perspective, find similarities and compare this project with others. It should be noted, however, that in a situation where software is developed entirely in-house or with a partner, there may be significant differences in the aspects mentioned below.

The aim of the case project is to develop maintenance management software for ships. The software contains functions that are generally considered to be CMMS functions. CMMS stands for Computerized Maintenance Management System and is the most rudimentary of the maintenance systems, or at least the scope it aims to fulfil is the narrowest. At its core, CMMS is a maintenance log that manages the maintenance of assets (machine, plant, ship, etc.) by setting maintenance intervals for them. The maintenance philosophy of CMMS is usually preventive maintenance, where the intervals are based on elapsed time or hours of operation of the asset. In the case of a machine or part, for example, the intervals are often specified by the manufacturer. The simplified question of this philosophy is, "When is the next maintenance due on the asset?" This type of philosophy is the easiest to implement because it requires minimal interaction between the asset and the software. It does not take into account failures, unexpected operating modes or the operating environment. (Mobley 2004, 4.)

The case-example software's properties can be separated into five individual units:

- PLATFORM. The basic overall functionalities of the system, e.g. user management, data import, search function, authentication, alarms, interfaces etc.
- EQUIPMENT INFORMATION MANAGEMENT. Basic information of assets, systems and sub-systems.
- WORK MANAGEMENT. Maintenance plans, workflows for tasks.
- MATERIALS MANAGEMENT. Features relating to checking and editing the inventory for spare parts and other (maintenance-related) consumables.
- DOCUMENT MANAGEMENT. Storing documents and document attributes, possibility to explore and comment documents.

The duration of the project is approximately 18 months. This thesis will be completed approximately 14 months after the start of development. The project is planned to continue through other phases, refining the product, adding features and providing services to other industries, but this thesis focuses only on the first phase because 1. it embodies the aspects of the SDLC. 2. it is an ongoing process with practical information available.

2.1 Marine sector additions

The software being developed is aimed at customers in the maritime environment, which is a complexity-adding factor. The users of the software are the personnel of ship operating companies. Some of the users are located onshore, usually in administrative roles, but a large portion of the user group consists of ship crew members. Internet coverage of the world's oceans ends usually a few kilometers from the shore, and satellite uplinks are rarely used for anything other than critical data transfer, because of the high costs and limited bandwidth. This means, for example, that the software must enable ships to manage maintenance activities onboard independently and offline. When there is a reliable connection to the internet and cloud server (e.g. when the vessel is near a port), all information is updated to the shore office and vice versa. This means that the system has to run in two environments: on the ship's internal network and in a cloud environment. For the crew, the software provides a mobile application connected to the ship's internal network. As the software is intended to serve a large customer base of shipowners, it must be able to cope with a wide range of fleet and ship technology levels and service expectations.

In order for the software to be valid in the maritime market sector, it also requires a certificate of compliance from a classification society. The certification proves that the software is capable of performing the necessary functions to be accepted as the maintenance management software of choice on any ship. The functions are relatively straightforward (such as filtering out all class-relevant items and generating their maintenance history), but it does affect the priority of certain features to be developed. It is also possible to obtain a cyber security notation from a classification society, which certifies that the software has been designed to combat cyber attacks. (Det Norske Veritas 2023.)

2.2 Other philosophies and types of maintenance management

As the field of maintenance management encompasses a number of terms and acronyms that are sometimes used interchangeably, it is worth highlighting a few that were not covered in the previous chapter. They are also important because, outside the scope of the current case project, the company's software aims to achieve a much higher level of service, more akin to the concepts presented below.

As a continuum to the preventive maintenance mentioned above, there are two other more complex types. The next level is condition-based maintenance, which also takes into account the mode of operation and operating conditions of an asset (Mobley 2004, 5). This is achieved by monitoring values (such as speed, temperature, pressure, etc.) and alerting when a pre-set threshold is exceeded. This allows maintenance activities to be more accurately timed and based on need. As the decision rules are static, condition-based maintenance is still a reactive method (Neurospace 2019). However, it should be noted that this level of monitoring requires a more complex level of instrumentation. The simplified basic question that defines condition-based maintenance could be "How do operating conditions change maintenance activities?

The highest level in terms of development is the predictive maintenance philosophy. This monitors the operation of the asset in a similar way, but in a more proactive way, using machine learning to analyse the data collected and, over time, make predictions about the appropriate maintenance intervals (Neurospace 2019). The ultimate goal of the predictive maintenance philosophy is to answer the question "How should maintenance activities be optimised?

3 General aspects of a software development project

3.1 Agile methods

According to IEEE (2020, 139), agile methods are widely used in software projects. Some of the reasons given are that agile methods are more flexible and reactive compared to traditional project management models, mainly waterfall. Flexibility is an important quality in software development, as the process is prone to change and emergent requirements. Agile methods are also considered to produce applicable content more quickly and are more affordable because they reduce overhead costs compared to large, intensely planned methods (IEEE 2014, 170).

There are several different agile methodologies, but they share some common denominators. The first is that they can be thought of as 'lightweight' models in which work is divided into short, incremental, successive iterations or cycles. Each cycle is intended to produce a small but functional part of the product, meaning that (re)planning, design, integration and testing take place concurrently during the cycles (IEEE 2014, 63). The workload and pace of the cycles is such, that they can be performed indefinitely (Measey et al. 2015, 8). The process also includes evaluation, which is carried out after each cycle. The evaluation is done with an emphasis on informality and aims to look for areas for improvement. This creates a consistent and precise feedback loop that works towards improvements after each cycle.

Another common factor is that the stance towards change is more accepting with agile methods than in waterfall. Agile emphasises reacting to change quickly and with minimized costs, rather than opposing differentiation from the created plan. (Kent et al. 2001.) In fact, because the content of the process can change dramatically during a project, according to IEEE (2020, 140), the dedication of agile projects is directed more towards the results than the planned activities. However, it is important to note that agile methods as a

whole have a requirements and design process that can generally be placed in the same timeframe as the planning process.

A third commonality of agile methods is that stakeholders and customers are heavily involved throughout the process (IEEE 2020, 139). The involvement of stakeholders, such as representatives of end users and potential customers, can ensure that the direction of the product is correct and that each iteration cycle produces a validated part of a whole. (Kent et al. 2001.)

A fourth factor is that agile teams are much more empowered and flatter than hierarchical models. Because progress is intense and change likely, the team must be able to make quick decisions and low-level improvisational pivots without consulting higher management or strict protocol (Holcombe 2008, 11).

The fifth factor common for agile is the critical attitude towards “comprehensive documentation”. This does not mean that documentation wouldn’t be important, but it should not take priority over delivering a working product. (Kent et al. 2001). Also, the created documentation should only fill the minimum critical requirements (term used for this is Just Barely Good Enough, or JBGE). Using more than the minimum effort on documentation can be considered a waste of resources in an agile mindset. (Cobb 2015 ,29).

The field of agile methods is dynamic. New methods emerge as some old ones are replaced as obsolete. Others may persist but change. Therefore, it is a challenge to provide a framework by which these methodologies can be selected for presentation. This thesis presents the agile methodologies described in SWEBOK, considering their mention in the publication as a reason to accept their generality and validity in the software development business. The main differences between the methodologies can be found in the life cycle models. However, since agile methods have several similarities, it is not uncommon to combine parts of them or to use only project-specific parts. This kind of combination is part of agile, and no single methodology is preferred over others (IEEE 2020, 139-140). The methodologies are intended to be taught as ways of thinking rather than as strict routines.

3.2 Scrum

Scrum is a well-known and widely used agile approach. Probable reasons for its popularity are the versatility of the methodology. Scrum can be used in conjunction with other methodologies (Holcombe 2008, 15). In addition, the methodology is considered to be "project management-friendly". The friendliness refers to the fact that in theory it is easier to monitor progress with Scrum compared to other agile methods. (IEEE 2014, 170.) In addition, Scrum is relatively easy to adopt (Kasurinen 2017, 19).

The development team in a scrum framework includes several different roles that work to ensure that production stays on track and that the product meets the customer's requirements. The roles are presented in chapter 4.1.

Sprints are an essential part of Scrum. Essentially, sprints are the division of work into separate, more manageable increments, all with the same pre-agreed duration, sometimes called a time-box (IEEE 2020, 139). The duration of a sprint varies from one to four weeks. To maintain focus, the duration of a sprint should not exceed 30 days (IEEE 2014, 170). In line with test-driven development, the scope of work in a sprint includes unit testing as well as UX design. The goal is to produce one or more functional and tested units of software as the end result of a sprint. One way of understanding sprints is that they are successive miniature lifecycles in themselves, affecting only certain parts of the overall product.

With this type of methodology, the intensity of the work should remain constant. Instead, the challenge is to break down the tasks into appropriately small increments. By default, there should be no single task that is too large to be completed in a single sprint (Cobb 2015, 40). The main idea of Scrum is to divide the workload of the development project into different units during the process. Initially, the division is high-level and becomes more specific as the project matures. At the most usable level, the units are user stories (see Section 3.7), ideally all of which are assigned importance values and estimated effort to produce. This kind of division allows the project team to always select the most

important items to be put into production, and also to include the right amount of them in order to maintain a moderate workload. The list formed from this activity is called the product backlog (Alt-Simmons 2016, 68). From the collective perspective of the stakeholders across the team, an obvious advantage over the waterfall is that large wholes do not have to be created completely and one after another, but instead the customer receives small pieces for validation at a steady pace. The highest value items can be created in any order, as shown in Figure 1. In practice, however, there may be several limitations due to conflicting interests (See Ch. 6.1.1).

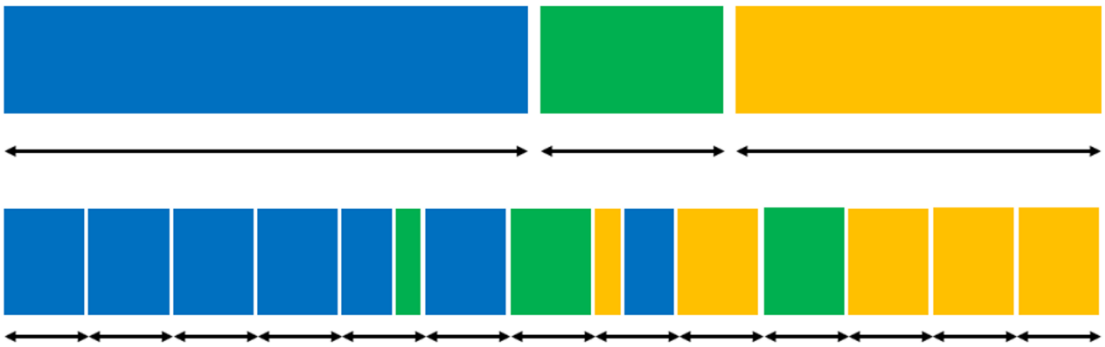

Figure 1. Difference in project implementation between waterfall and agile. In waterfall, the deliveries are large and happen in conjunction, whereas in agile, the deliveries happen more frequently and in a priority order.

For easy visual understanding, the team often maintains a Scrum Board, which is often a Kanban Board, on which all the items in the current sprint are placed in appropriate boxes representing their status (see Figure 2). With such a board, the status of each item selected for the sprint can be seen at a glance, providing an overview of the sprint situation. The use of Kanban boards improves the flow of work. (Gross & McInnis 2003, 1-3.) The number of boxes and their names vary from team to team. Typically, there is one box at the starting end of the linear formation for items not yet built and one box at the opposite end for completed items. The kanban board is a visual tool that is not inherently tied to any methodology, but is used in most. It should be noted that the states of the Kanban board do not necessarily represent all possible item statuses an item can have, but only those which are needed during the

construction process (See Ch. 8). For an example of the full status options, see Ch. 8.4.1.

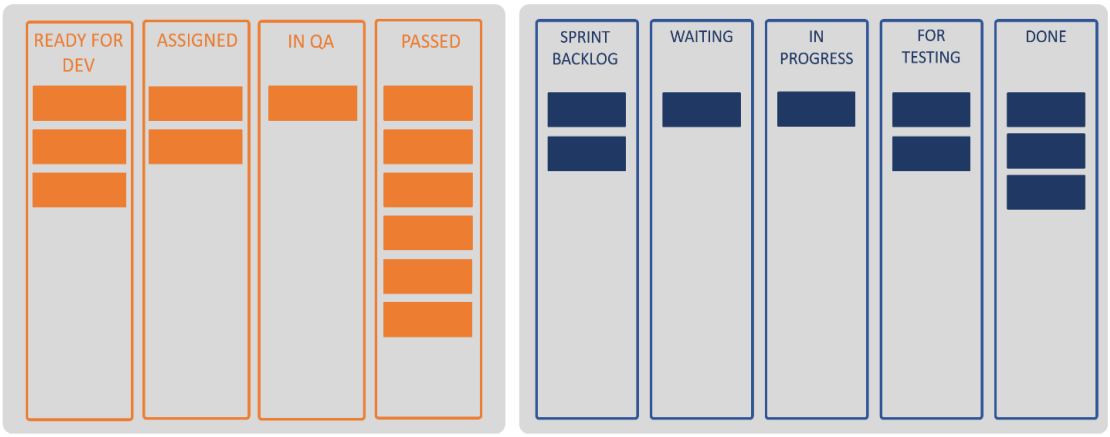

Figure 2. Two example box-sets on a Kanban-style scrum board.

3.3 Other methodologies in SWEBOK

3.3.1 Rapid application development (RAD)

Rapid application development arguably places the greatest emphasis on getting something on the table for comment as quickly as possible (outsystems n.d.). This is achieved through the use of prototyping tools. The RAD process starts with a requirement specification phase, which is notably short. Only the main features and the vision of the product are worked out in the team, no time is spent on the specifics, mostly on the assumption that they can be changed.

After a broad specification, prototypes are created using various specialised tools (IEEE 2014, 170). The created prototypes then serve as a medium for feedback, discussion and more specific requirements. Iterations are made as needed. The actual functional product is only constructed if the feedback from the prototype iteration is positive, as shown in Figure 3. In this way, the design is already validated. The prototypes are by nature focused on the front-end (see Section 4.1), so the construction phase then focuses on the backend. The

overall schedule is quite similar to other methodologies, mainly because the time saved in a short specification period is used in prototyping and correcting possible shortcuts made when prototypes were created.

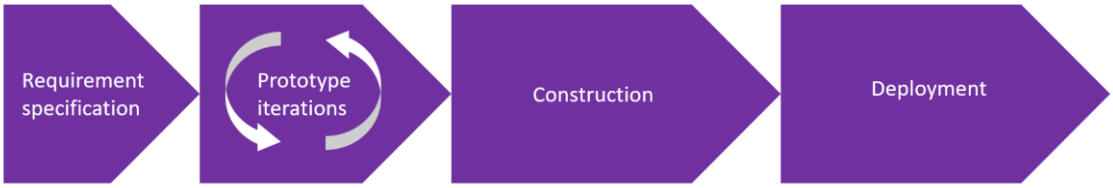

Figure 3. Phases of RAD (adapted from outsystems n.d.).

some criticism of RAD is that there are no pre-determined timeframes for activities, so there is no constant pace or easily measured speed. This makes the RAD methodology difficult to manage. Other negative aspects include the need for extensive prototyping skills and tools within the team, RAD does not work well with large teams and long projects, and also does not work when developing systems that cannot be modularised (Outsystems n.d.).

3.3.2 Extreme programming (XP)

As the name suggests, extreme programming stems from the need to increase productivity by emphasising the most useful practices of software engineering. The emphasis is on ease of use and product quality. The goal of extreme programming is to find the simplest functional solution to any problem. The written code is kept simple because of expected changes along the process. The work is intended to be carried out through feedback loops in various forms and timeframes, as shown in Figure 4. An example of this is pair programming, where each line of code is simultaneously 'tested' by the observer as it is written by the driver. There is a strong emphasis on testing, and in fact tests are developed before anything else (IEEE 2014). Even requirement specifications can be written in terms of test acceptance criteria, and test results are the main source of product validation. (Holcombe 2008, 25-26).

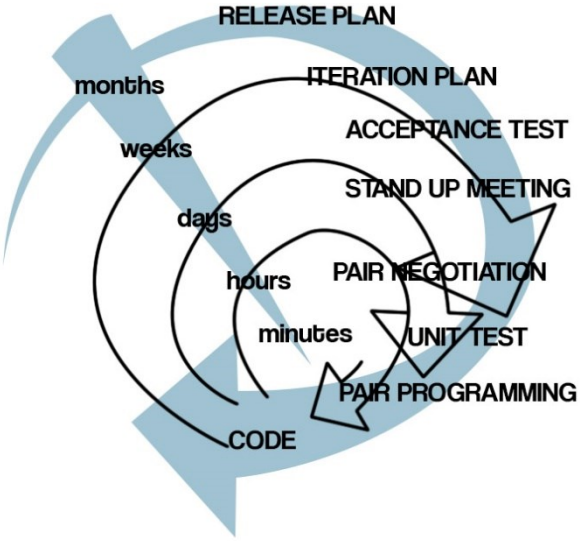

Figure 4. Feedback loop -testing system in extreme programming (Dennehy 2009, 6).

Unlike RAD, extreme programming uses development cycles of predetermined length, which makes forecasting easier. However, because the focus is on validating code, the long-term scope of the product is poorly understood. Extreme programming is very much focused on the present moment. (Holcombe 2008, 25-27.)

XP is unique in that one of its cornerstones is that the customer should always be available. This means that a representative of the customer is included in the project team, takes part in the definition and is available to answer questions, preferably on-site. (Holcombe 2008, 28.) This aspect has the potential to have a negative impact on the project in the form of, for example, micro-management by the customer in an otherwise flat management system (extremeprogramming.org n.d).

With the increased emphasis on creating functionality, the overall design of the system and the UI/UX aspects are left to accumulate technical debt. Also, because the requirements are essentially an acceptance test and new requirements are written with each failure, both the design of the system and it's

requirements are not well known in the beginning and their expansion is incremental. From a project manager's point of view, especially from an engineering project background, this can be challenging.

3.3.3 Feature-driven development (FDD)

Although it contains the elements of cyclic evolution and stakeholder involvement, feature-driven development takes steps towards a traditional waterfall model by having distinct phases (shown in Figure 5), a comparatively much larger amount of up-front planning, and a lower propensity to refactor (Holcombe 2008, 13-14).

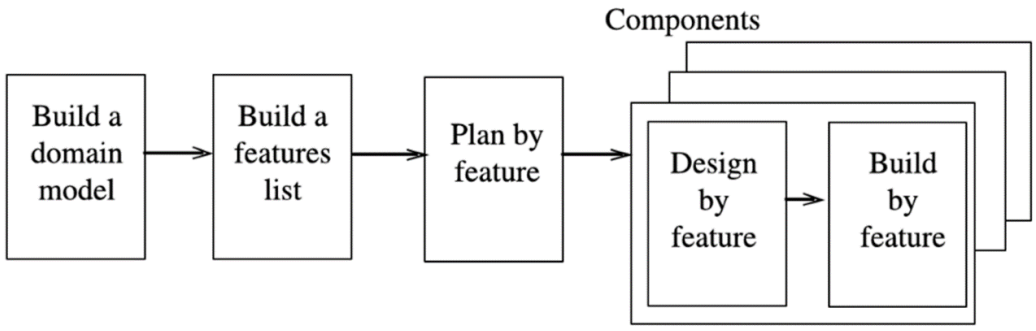

Figure 5. The five-phase process of FDD (Holcombe 2008, 14).

The first three phases of an FDD project are evolutionary planning, and the last two are the actual design, component by component. Unlike XP, code ownership is at the individual member level rather than the team level (IEEE 2014, 170). Features are often grouped into feature sets and assigned to appropriate developers. Pre-production planning facilitates the early introduction of written documentation and makes it easier to produce interim reports and performance updates (Holcombe 2008, 13-14).

The disadvantages of FDD are that it takes a lot of time to plan, which also results in delayed deliveries. FDD is considered to be more complex than other methods and it's success relies heavily on the coordination skills of the design

leads. As whole features or feature sets are designed and built sequentially, the possibility to create high-value items first diminishes.

3.4 Social aspects of agile

Agile methodologies differ from waterfall models also in the sense of what kind of social aspects and skills are valued. The Agile Manifesto identifies twelve defining principles, which shape the nature of agile methods (Kent et al. 2001). By analysing these principles, it is possible to deduce that some of them contain promoted social values. The value-loaded principles and their analysis are presented below (Kent et al. 2001):

"Business people and developers must work together daily throughout the project". "The most efficient and effective method of conveying information to and within a development team is face-to-face communication." These two principles clearly emphasize the importance of constant and frequent communication between various people. From a social aspect this emphasis can translate to good communication skills and a certain level of extrovertedness. Even though Measey et al. (2015, 8) points out that well implemented virtual collaboration spaces can also constitute as a communication enabler, the full benefit of communication is expected to be received face-to-face in an agile mindset.

"Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done". "The best architectures, requirements and designs emerge from self-organizing teams." The underlying social aspects sought after here could be the ability for each team member to take responsibility and initiative of their work as well as willingness to share burdens and maintain other's motivation (IEEE 2014, 200). Literature even seems to imply that agile team's motivation would be in direct correlation with the amount of self-empowerment it possesses (Measey et al. 2015, 9).

“At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly”. Open mind, willingness to accept critique and resilience to change are qualities at the core of an agile team’s social attributes. Also a critical but constructive mindset towards oneself and other team members is required in order improve.

All of the qualities mentioned above are concerning to each team member of the development team. From the Project Manager’s perspective, the set of social skills are additionally complimented with the ability to motivate the team and communicate in a supporting way. Management overall must be able to trust the team’s ability to deliver quality products on time, give the team all necessary tools for success and then take a step back. A derivative of these issues is that the Project Manager needs to additionally possess good team-building skills.

3.5 Agile methods challenges

When making the transition from managing an engineering project to managing a software development project, certain fundamental aspects of the difference need to be accepted and the management style adapted accordingly. An appropriate mindset might be that instead of seeing aspects as disadvantages, they are trade-offs that have a positive impact elsewhere. With this in mind, there are however some items listed below, which require familiarisation from a Project Manager accustomed to waterfall.

- The project manager's position is more vague. In the optimal situation, where everyone on a team can focus on one role, some responsibilities related to general day-to-day team activities, such as work-sharing, are more naturally handled by another role than by the PM. In fact, a Scrum team does not have a Project Manager role within it. It exists at a higher level because there needs to be one person responsible for the overall schedule, budget and scope of the product. This forces the PM to move up and possibly look at the project from a different level than they are used to.
- Flat organisational structure. Software engineering is less hierarchical than traditional project fields. Decision-making power is spread across more team members, and even developers are more empowered to steer the product. Decisions are also made much more quickly, and it is a challenge for the project manager to keep track of them and assess their relationship to the scope.
- Challenges in assessing far-reaching progress. The Scrum Master, with the help of the development team, should usually have the best knowledge of the total workload. However, the Scrum Master’s main focus is on filling the next few sprints with correct and defined User Stories, rather than providing accurate information about the total workload far in advance. In addition, features and large user stories are often broken down into smaller pieces on a just-in-time basis, meaning that the backlog is likely incomplete until the end of the project.

- High uncertainty, low sense of control. Once the requirements for the product have been specified and the scope agreed, the project manager has only experience-based, high-level information about the schedule. The pace at which accuracy increases is slow, and there is a risk of emergent issues and scope creep. Agile methodology also seems to stem from the starting point that the team always knows what it’s doing, and can arrive to the correct solutions if left alone. With the combined effect of the above, the project manager is subject to a significantly lower sense of control, and the future is much fuzzier than in a general engineering projects.
- Documentation. In most agile methodologies, documentation is a relatively low priority. Software projects do produce documentation, but much of it is produced towards the end of the project. Instead of "this is what we plan to do", the theme of the documentation is "this is what we did". This deviation makes it very difficult for the project manager to accurately report, control and validate at the beginning of the project. The project backlog is an important initial document, but because it's often incomplete at the start of the project, its value is limited. The “Just Barely Good Enough” -mentality does not help in this respect.
- Stakeholder Resources. The agile development team produces new content frequently, which means that customers need to be involved in validating and commenting frequently. Customers also need to communicate their needs to the team with increasing specificity. This means that the customer's own resource commitment is likely larger than in waterfall.
- Social demands. Agile methods are fundamentally based on teamwork, self guidance and constant communication and collaboration. Individuals who do not possess teamwork skills, are not self-guided or would prefer to work independently from others are not suitable members of an agile team. It could be thus argued that agile team building can be more exclusive when compared to waterfall-type teams.

3.6 Work breakdown

The basic idea of breaking work down into smaller parts within a project is the same in software development as it is in other fields. Smaller pieces of work are easier to assign, easier to manage and less daunting. With agile methodologies, especially Scrum, the importance of this activity is key because of the time sensitivity. There are several ways to divide the work into parts. As a basic rule of thumb, there should be at least three different piece sizes: one that a full-stack developer can complete within a design cycle, another that takes 2-5 cycles and a third that combines the two smaller units and acts as the largest unit, with a completion time of 5+ cycles. This composition is illustrated in Figure 6. The tiers may have different names, depending, for example, on the corporate culture. There is a risk of confusion if a particular term is used for a different tier in different circumstances. The overarching theme of the tiers is user-centredness (IEEE 2020, 18, 45).

The breakdown and designations presented in this thesis follow the guideline used in the case project. It is found to be a useful framework because it is intuitively understood by both the company and the partner, it is well established, and it is supported in commonly used project management software.

Starting from the bottom, the lowest level unit is called a Task, sometimes a Subtask. Often, however, Tasks are only considered as framework level entities and are rarely documented. This is because Tasks are such short activities and there are so many of them. An example of a Task might be Create a button with the text "Add Document".

One level above tasks are User Stories, or simply Stories. They are the optimal level into which all work should be broken down. User Stories are documented and should contain all the information the team needs to complete them. The most important information is the non-technical, narrative description of what needs to be done, why, and by whom. Example of a User Story: "As the Chief Engineer creating a work order, I want to be able to select and add the required

documents to the work order so that the technicians know what documents they need to complete the work order”. Other important information attached to a User Story:

- Definition of Done (called also acceptance criteria), i.e. the metrics for evaluating when the development has achieved the functionality of the user story.
- Linked information, such as UX-designs.
- Link to relating higher tier.
- Priority indication.
- Workload, i.e. story/action point estimate (See Ch. 8.3.2)

Continuing upwards, the next level items are called Features, which contain the combination of multiple user stories associated with them. A Feature is a distinctive aspect of the whole product. An example following the theme of the User Story above might be User-Created Work Orders.

At the top level are Epics, which in turn collect related Features to form a substantial part of the product. Because of their size, epics usually take several development cycles to complete. Example: Work Management.

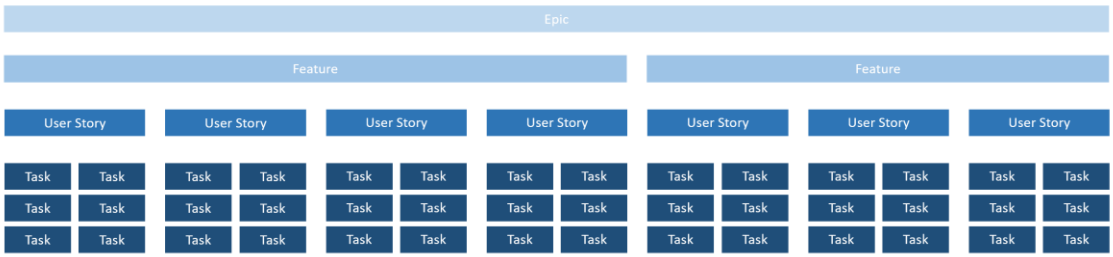

Figure 6. Tiers and designations of work breakdown in a software development project.

3.7 Unified modelling language

The documentation produced during the various stages of software development is often presented in a visual form, called models or diagrams. In order to fully understand the information presented in the documentation, it is necessary to have a basic understanding of the lingua franca of software development, the Unified Modelling Language, or UML. UML diagrams aim to show the structure of the software, the behaviour of its components, and how the components interact. Components are often represented as different shapes with descriptive text inside. Interactions are shown using different types of arrows. Examples are shown in Figure 7. The benefits of UML are improved communication between stakeholders and improved ability to specify the software system. (Otero 2012, 36.) Examples of the central diagrams are presented in chapters 6.1.1 and 7.

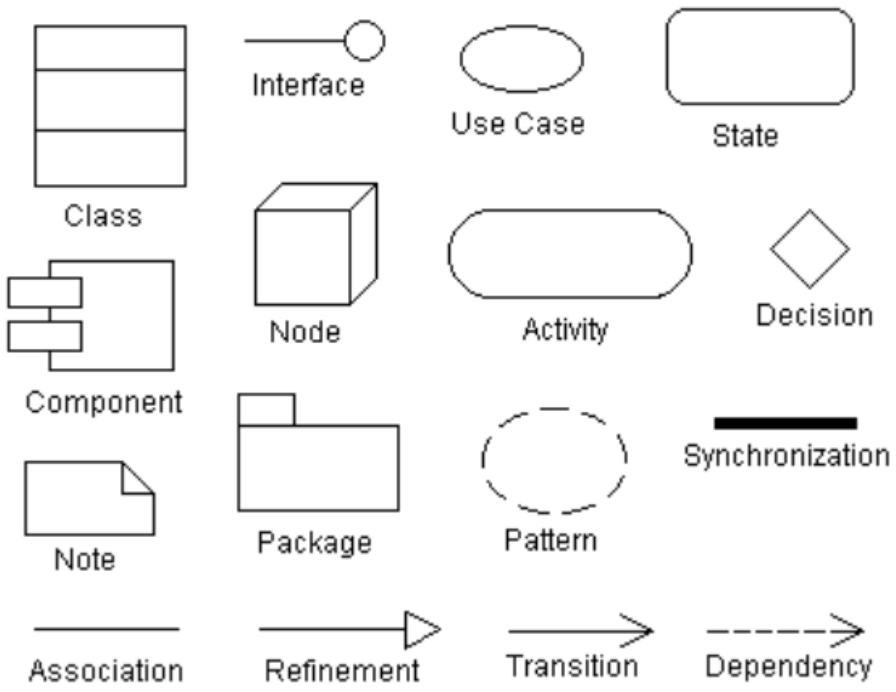

Figure 7. Examples of UML diagram elements (Loyola Marymount University 2023).

3.8 Case-project's insights for common practices

There are several factors indicating that the Case-project utilized an agile approach in its activities:

> The workload of the development team was divided into cycles throughout the project. Meeting types and the teams division of roles were those incorporated into the scrum approach (See Ch. 8.3 & 4.1). Work was categorized into epics, features and user stories and the workload was estimated using story points. Lastly, the tools used for managing the project are directed towards software development projects (See Ch. 3.9)

Simultaneously, there were also factors present suggesting towards a more conventional method of project management, or at least in contrast with agile methodology:

> Company emphasized the importance of planning and producing documentation before the design phase. This was partly due because it was urgent to provide company-wide information about the software in order to plan continuation. Company and partner were in practical terms two separate teams working together, instead of one common team with shared efforts. Some reasons for this were time difference, customer-service provider -relationship and need-to-know -basis of company information. Dual team structure caused some duplication of duties, resulting in a larger management structure than what might be present in an average agile development team.

The method of working in the case-project could be described as a combination of Scrum and FDD. Work was organized into sprints, which lasted 2 weeks each. The combined amount of development sprints for this project is 23. Sprints were started only at the moment when software construction began, before that the activities were not time-boxed. Additionally, task management in Jira was not executed before the start of construction. Tasks could be found from meeting memos, which were posted for all participants to see.

The partner was trusted with much liberties in planning the sprint contents in the beginning. This lead to a phenomenon, where the team decided to create complete Epics in the spirit of FDD feature sets. The first two Epics created were Platform and Company Management. From the perspective of PO and business side, these were the two most low-value Epics. Their content (e.g. creating new company profile to which to insert basic information, creating a method for logging in, flow for creating new password etc.) was so trivial that company personnel had nothing to show potential customers for an extended duration of time. Another factor which caused problems was that there was no visual look for the product designed for quite some time, leading the team utilize a very basic IBM Carbon -design system coloring, which had a plain appearance, an example screen can be seen in Figure 8.

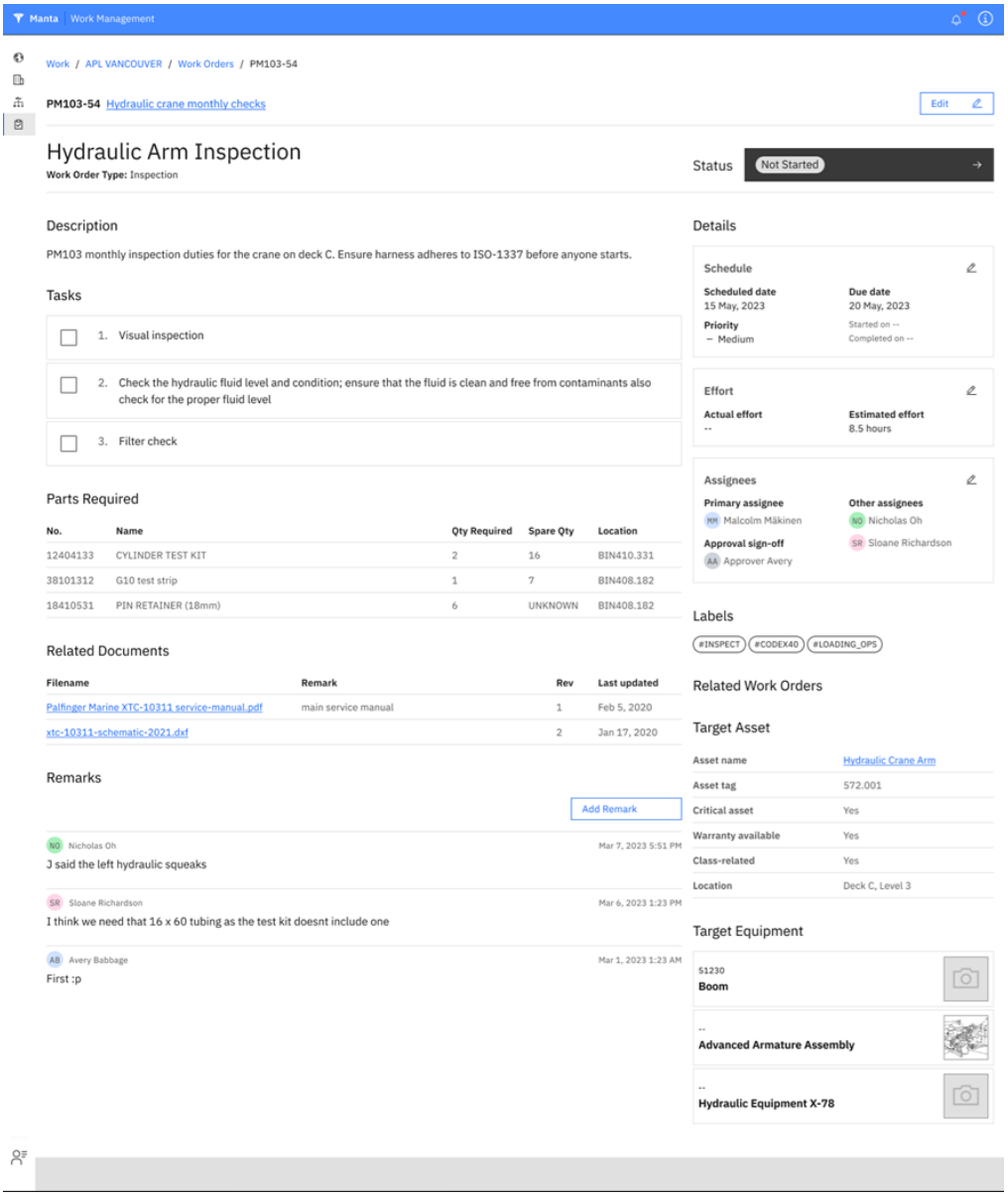

Figure 8. example view from the software before any design towards visual look.

The progress was presented well in biweekly sprint demos (See Ch. 8.3.3), but backend solutions, ever larger technological decisions were not presented well. The Partner did not provide options to be reviewed, but instead chose a certain path independently and and built the foundation on it. At a later stage it was discovered that this solution caused the program to be slow and unresponsive, with waiting times of up to seven seconds. At that stage of the project, pivoting to a different solutions would have set the timeline back several months, so the

only solution was to try all available methods to quicken the response time of the selected solution.

3.9 Case-project tools

The main project management -related tool used in this project is Jira (Atlassian corp.) The main tools for document management and storage are Confluence (Atlassian corp.) and M-Files (M-Files Corp.). In accordance with the theme of this thesis, tools mentioned above are at the centre of focus. It should be noted that Jira and Confluence were selected on purpose for this project, the usage of M-Files was according to Company policy. For communication purposes the team utilized Teams (Microsoft), Slack (Slack Technologies) and Outlook (Microsoft). These or similar tools can be considered standard practice in any type of modern project communication regardless of approach and as such are deemed unnecessary to be presented deeper by the author. Tools used for actual software engineering are outside the scope of this thesis.

Jira is a software especially designed to facilitate services for the needs of software development life cycle (Atlassian Jira. 2023). It utilizes many useful instruments, such as Kanban charts, roadmaps, charts, reports and customizable dashboards. Jira facilitates the possibility to create and assign various different kinds of items, and track their progress.

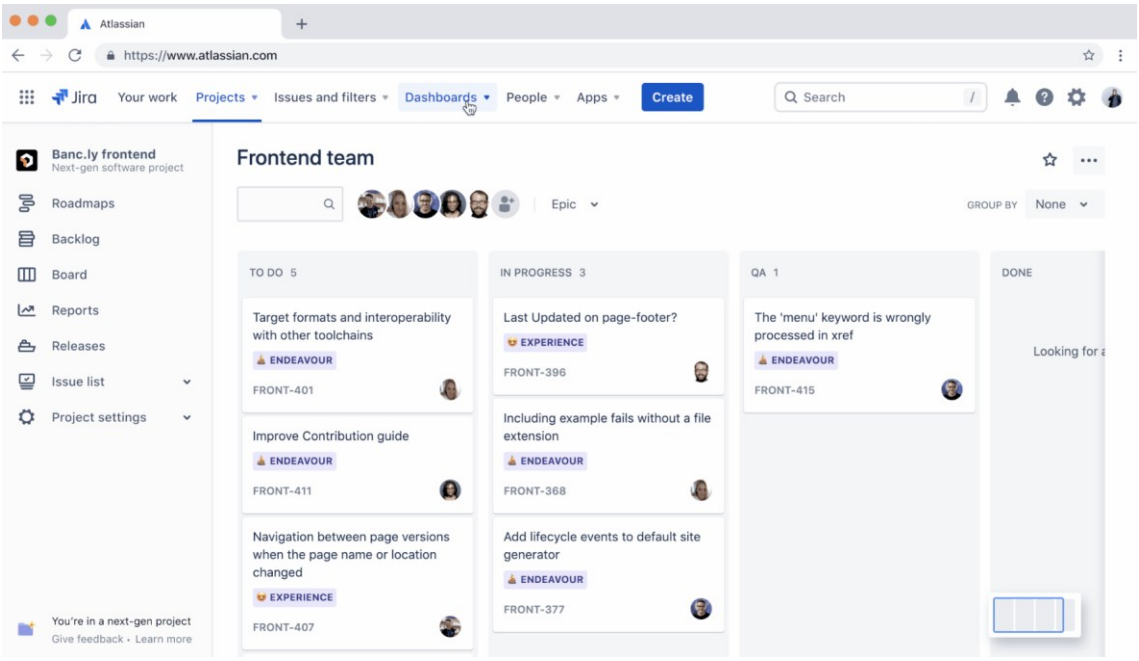

Figure 9. Example view from Jira.

The Company and Partner had separate instances of Jira. This was mainly due to the fact that it enabled a simpler way of managing both teams. At the beginning of the project, all participants had access to certain parts of the Company's instance, because the requirements for the software were stored there. However, because it was not necessary for the Partner to be able to see Company personnel's internal tasks, two separate instances were created. The whole team had full access to the Partner's instance. User stories were refined and sprints planned and executed there. An example view of Jira can be seen in Figure 9.

For document management, the team utilized Confluence, which is also an Atlassian product. Confluence is a workspace onto which team members are able to upload media and documentation. Additionally, documentation pages can be created and edited with the help of shortcut tools (Atlassian Confluence. 2023). For document management and archiving purposes, Confluence enables exporting of documentation in various forms.

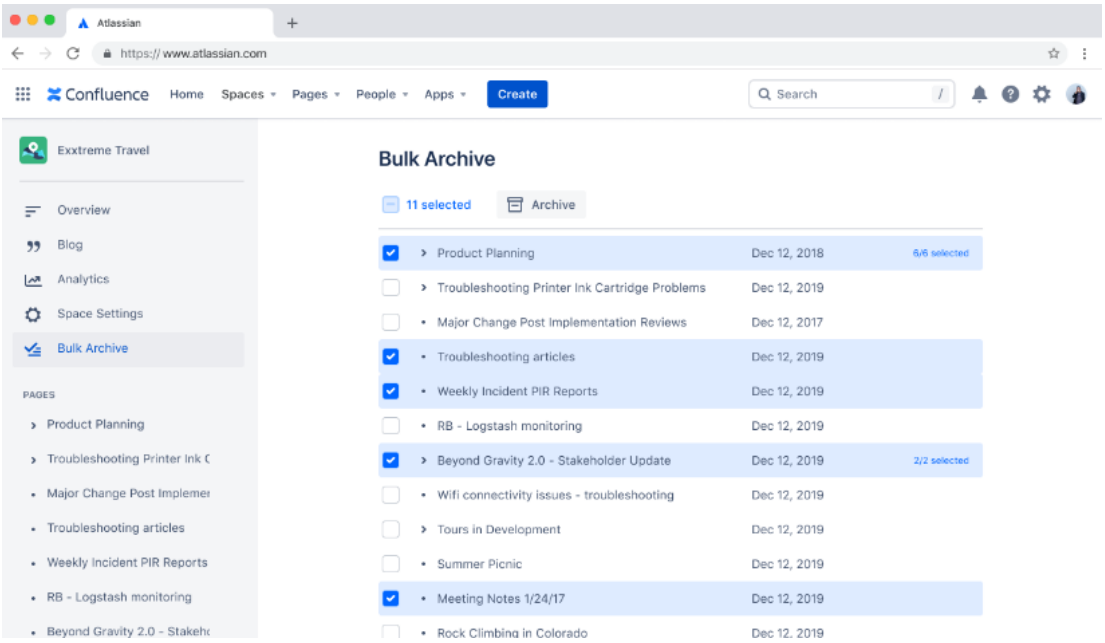

Figure 10. Example view of Confluence (Atlassian, Confluence 2023).

In the Case-project, there were also two instances of Confluence, but the Company's internal one was in lighter use. Instead. it was more practical to have all project-critical content in one place. Examples for the contents of Confluence for this project are initial information, process overviews, meeting notes, sprint demo recordings and test approach clarification as can be seen in Figure 10. During the process, it was apparent that Confluence was not in daily use of the team. The amount of content could stay unchanged for extended durations and there existed several pages of material acting as a space reservation for a certain topic, but never progressing further than the headline. For team collaboration purposes, especially discourse about topics, Confluence was underused in the Case-project.

Another document management system in use was M-Files which is based on sorting information by content instead of folder structure (M-Files. 2023). In the Project scope, it was used as the final place to store documentation. All important Confluence documentation ultimately needs to be found on the Company's M-Files. The Partner had no access. Example M-Files view can be seen in Figure 11.

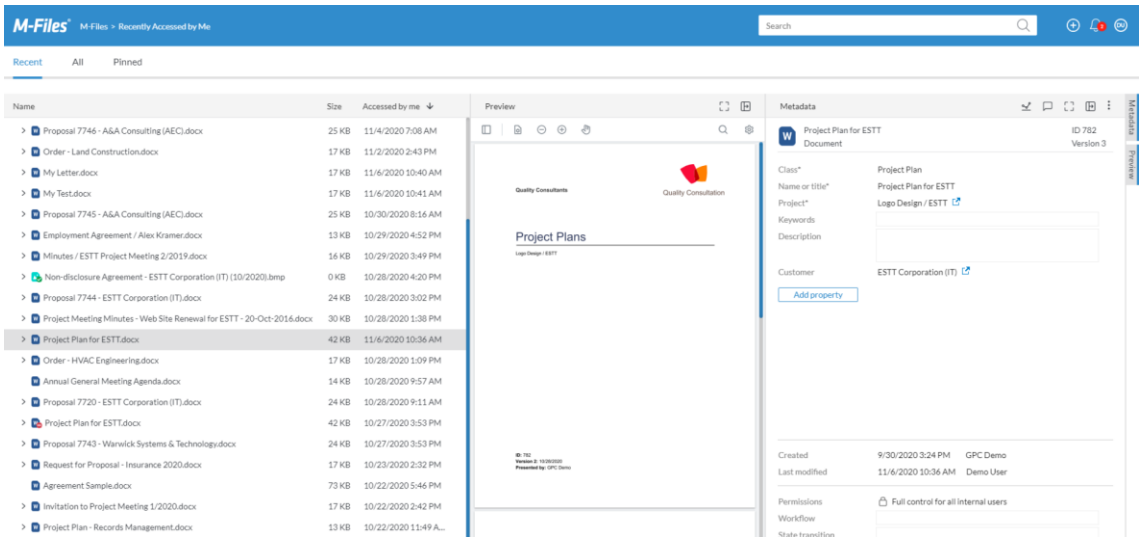

Figure 11. Example vie of M-Files (M-Files 2023).

Tips:

- The selection of methodologies and meeting routines should be negotiated and decided together among the most relevant stakeholders.
- The aim of the development team should be to present every internal product decision to the customer representative for feedback. The positive effects of agile development dissipate if e.g. an architectural decision has been made unilaterally and it is deemed a wrong decision by the customer upon discovery. This risk can be mitigated with short sprints and defined backlog, but in practical life, the risk still exists.
- On the other hand, it should be acknowledged that customer feedback has a risk of containing items that were out of the agreed scope. These need to be clearly identified and negotiated. Facilitating frequent feedback cycles does not guarantee quality feedback. It should not be taken as a given that customers and stakeholders would able to convey all their comments clearly and on time, although this is the goal. A helping factor would be that the team members possess basic domain knowledge for informed decision-making to prevent a situation where unreceived feedback would halt production. Domain knowledge can be trained e.g. through workshops.
- When deciding sprint length, all pros and cons should be addressed. In general, a longer sprint is probably more productive because the meetings and ceremonies take less time in relation. On the other hand, e.g. four-week sprints have only 50% of the feeback loops compared to two-week sprints.

- In a situation where the customer is not the end user, it is important for project management to ensure an effective method of attaining domain knowledge for the team, or involving presentatives of end users to view and comment the product along the whole process.
- In the spirit of agile, the made decisions should not be considered set in stone, and can be changed later if deemed necessary. In practicality however, this always has an effect on the project aspects of timeline, scope and budget. With some decisions (such as selection of Cloud host), the effect is so large that they do not allow pivoting.
- The brand and visual look of the product should ideally be defined as early on as possible. Otherwise, the look and possibly even layout of all of the software's screen views will need to be reiterated later. UX Designer's most important objective at the beginning is to create wireframes of the usage flow, in order to gain headway in relation to the developers. Therefore it is advisable to utilize a separate expert to design the look and feel. This activity does not affect the development team's routines, so the expert can also be an external.

4 Development team

The composition of a software development team is well established in literature. Rather than a specific methodology, the team organisation is based on the roles needed to produce functional software. These roles may have several different names and may vary slightly in content between different approaches. This thesis uses the Scrum team names as a basis because of their widespread recognisability. An overview of the Scrum team roles is shown in Figure 12.

Figure 12. General software development team composition.

Although the management structure of development teams is deliberately kept flat, there is a need for specialised roles and overall team management. In this respect, the same rules apply as for conventional management styles, e.g. the recommended number of subordinates to be managed by one person. A typical software development team is generally considered to have 5-10 members with cross-functional expertise (Schwaber & Sutherland 2020). Measey et al. (2015, 8) arrive at nearly similar figures, suggesting team size to be between 3 to 11 members. In a very small team, each individual single-handedly represents a specific role and therefore needs to have a high level of competence. One or two people in the team will be in a managerial position, preferably at least one. A large or complex software project may have several individual teams, sometimes called pods. The software features are divided between the teams, so that each one has it's own implementation track and backlog. In these

situations, coordination activities between the teams are very important in order to create a cohesive product and to match timelines. (Resnick et al. 2011.)

4.1 Scrum roles

Scrum Master

The Scrum Master's role is to keep the Scrum team's process on track, to facilitate the work and to remove any obstacles the team encounters. The Scrum Master's role is prominent at the beginning of the project, but as the development team evolves into a self-managing unit, the role changes form to support. (Pries & Quigley 2010, 51.) Throughout the project, the Scrum Master is focused inward, concentrating on the team. Some pragmatic examples of the Scrum Master's tasks are facilitating meetings, refining User Stories and selecting appropriate work quantities for sprint plans.

Product Owner

Pries & Quigley (2010, 52) refer to the product owner as the voice of the customer. The product owner is more externally focused on the business perspective. The role involves making sure that the product contains everything the customer wants it to contain, and actively validating that the team is doing the right things in the right order. Very often the Product Owner is the centre of communication between the development team and stakeholders. Practical examples of the role's tasks include writing User Stories, refining the backlog with the Scrum Master, and answering the team's product-related questions.

As both Scrum Master and Product Owner are management positions, it is possible, especially in small teams, for these roles to be held by the same person. The same applies to managers of separate teams/pods.

Developers

The term developer encompasses several job titles, but in essence developers are the part of the group that 'builds' the software (Evans 2004, 77). The two main paradigms for building software are front-end and back-end. Front-end elements are everything that the user of the software sees and interacts with in the program, such as the functionality of software that responds to the click of a button. Backend refers to elements that work in the background, inaccessible to the user, such as rules for resolving data conflicts. To produce viable units of software in each sprint, the team needs both front-end and back-end development. This is usually achieved by having people on the team who specialise in one or the other. A developer who can handle both front-end and back-end is called a full-stack developer. When considering a single pod, the natural number of developers is 2-4 people. A developer's role is to estimate the workload of each story assigned to them at the beginning of each sprint, and to complete the stories, report progress and issues during the sprint. Other roles associated into the developer term are e.g.

- Technical Writer, who is usually participating at a relatively late stage of development, responsible for documenting all relating features and their development as well as creating a user guide.
- AI expert, depending whether the software is required to utilize machine learning or other contemporary technological solutions which require targeted expertise.
- Lead Developer, who manages the design in, mostly in large organization structures.

QA Engineer

The quality assurance (QA) of software under development is usually verified by various types of testing (see Chapter 9). For this reason, the QA Engineer can also be called a Test Engineer. The role includes planning the tests for the stories in each sprint, executing the tests, and reporting any defects and bugs to the team. Since the goal is to create a working unit with each sprint, it is expected that all defects will be fixed before the end of the sprint. The QA Engineer's work is therefore intensive and requires planning. (Westfall 2016, 447-448.) Depending on the complexity of the features and the speed of production, there may be more than one test engineer in the team. In such cases, the other may be dedicated to creating test automation.

UX Designer

UX is derived from the words user experience. As the name suggests, a UX Designer has the perspective of the future users of the software. By creating a clear and informative UI (user interface) and logical, intuitive usage flows, the UX Designer ensures that using the product is a positive experience. (Rosenzweig 2015, 7.) The UX Designer optimally works a few sprints ahead of the rest of the team, providing tangible wireframes for the team to use in development. In many cases, the wireframes can also be compiled into no-code prototypes of the software's features. The overall look and feel of the software may also be the responsibility of the UX Designer.

Data Architect

The Data Architect creates the models for the flow and use of data between different entities in the software. This role is key to clarifying the overall principles and behaviour of the product. The Data Architect tends to work at a high level, and the content he/she produces is often in the form of diagrams. The task of actualising these plans may be assigned to another person, whose title is often Data Engineer. (Nath et al. 2017, 19.)

4.2 Development team stakeholders

The environment outside the development team, as in other business environments, has a variety of stakeholders (see Figure 13). The stakeholders can be categorised as internal and external, where internal means that the stakeholders are within the same organisation that employs the development team. The composition of the stakeholders and their titles will vary from situation to situation and no general definition can be given. Stakeholders often have no direct contact with the team, the Product Owner is primarily the link between them.

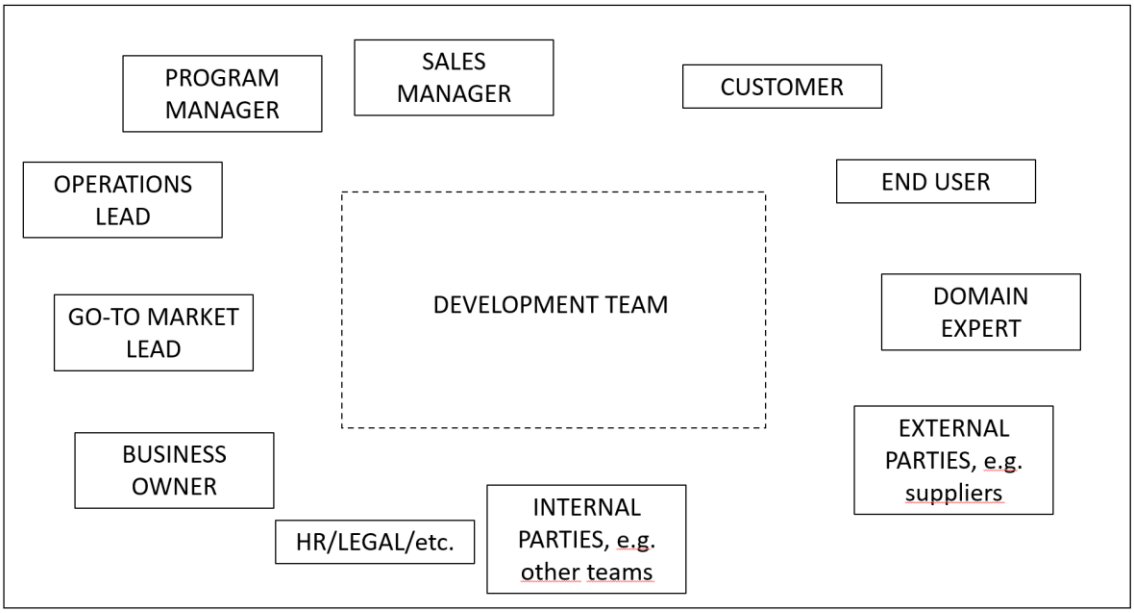

Figure 13. Examples of stakeholders.

4.3 Team scaling

Due to the nature of software development, the importance of the roles listed in Section 4.1 changes as the project matures. A simplification of this is shown in Figure 14. Not all team members are necessarily involved in the development from start to finish. If it is a single development team, the possible scaling tends to be downward (people become detached from the team rather than attached to it) and intensifies towards the end of the project.

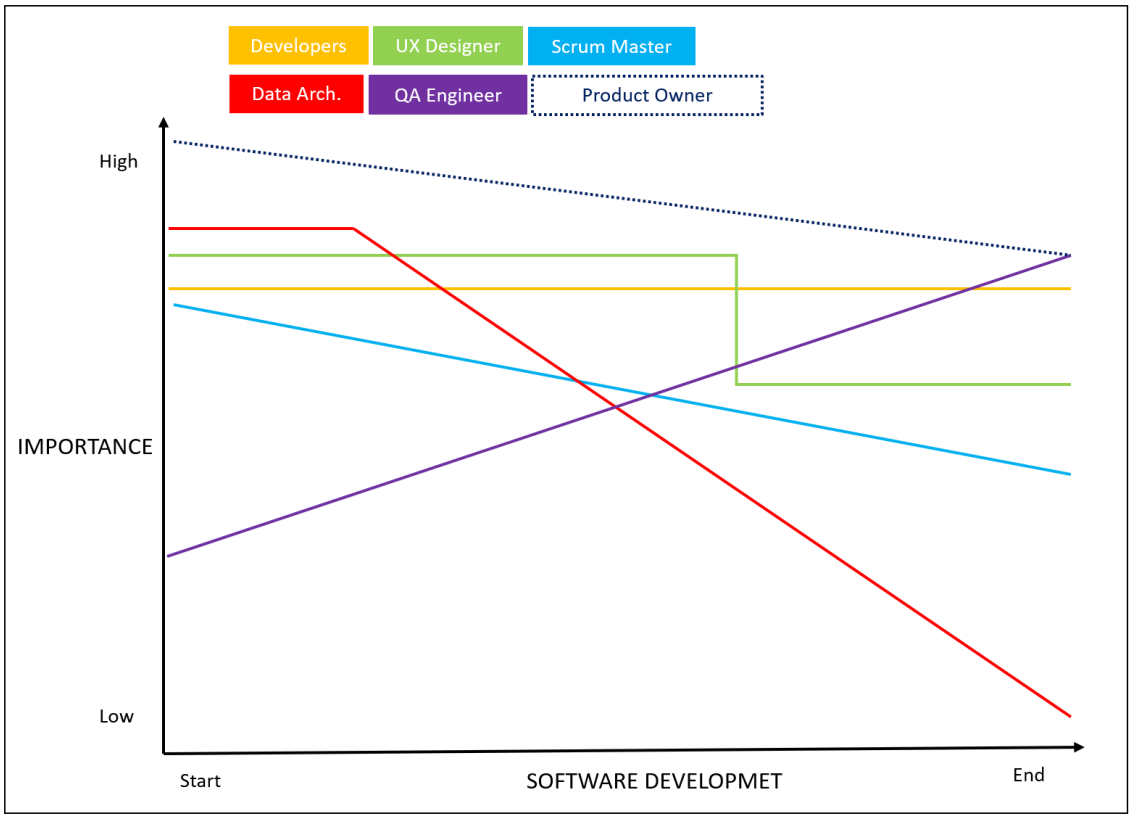

Figure 14. A simplification of the team roles' level of importance on a time-axis.

The only role where the need is constant and stable is the Developer. Coding can be expected to start from the first sprint of the design phase and continue until the last. Towards the end, the amount of Developers might however be dropped because of clarifired backlog and a moderate amount of remaining User Stories. The need for Data Architects and UX Designers can be seen as highest at the beginning of the project, as their input dramatically influences and

directs the work of the Developers. The Data Architect creates the framework from which the logic of the software is built, but after this effort the need for the role diminishes. The Data Architect is the most likely person not to continue until the end of the project. For the UX Designer, the need is constant until the full design round is completed. After that, the developers should have all the information they need to build the system. However, the need does not drop to zero, because in reality there is a lot of iterative work to be done.

The managerial positions of Scrum Master and Product Owner should become less important as the project progresses, signalling that the team understands the product and is empowered to continue. However, due to the volatile nature of software development, it is preferable not to eliminate these roles altogether.

The role of testing is initially low due to the limited number of items to be tested, but the work involves planning, so the need is by no means negligible. As the product expands, testing becomes more complex, and by the end of the build phase, the tester's role has become the most important, as the product must pass acceptance criteria to be considered complete.

4.4 Case-project’s insights for development team

As the case-project was carried out as a collaboration between two companies, it is obvious that there was some partial duplication of roles. Examples of this phenomenon are the roles of Project Manager and Product Owner. Although in theory a project should have only one manager and the product (of this size) should have only one owner, in practice in a joint global operation it is necessary for both participants to have their own internal organisational structure. This structure is visualised in Figure 15. An additional reason for the number of Product Owners was that during the project the product was split into two distinct but aligned pods (online & offline environments). Both of the partner’s PO’s also doubled as Scrum Masters, maintaining the production routine of the two teams. As the nature of the collaboration could be condensed into that of a customer and a service provider, the hierarchy could still be

formed and the effect of duplication on the practical work was limited. Each team member had a fairly clear and separate set of responsibilities.

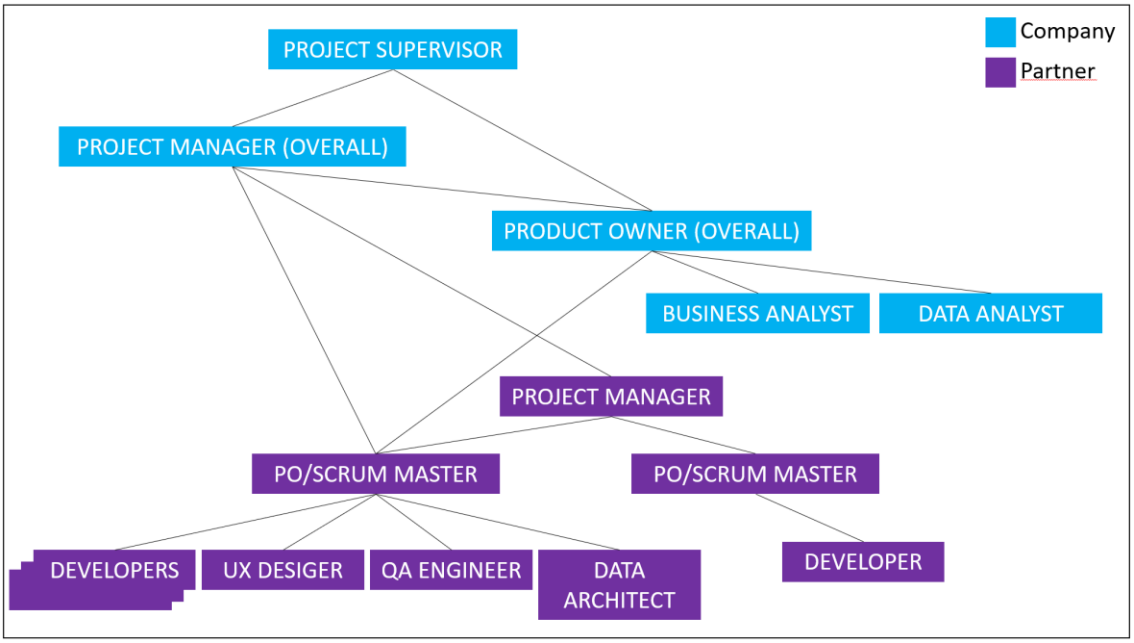

Figure 15. Case-project's organizational structure after one year from kick-off. Lines represent the main interactions.

The Company's team members were permanent, full-time employees. All had been with the Company before the case project began. The team was formed by the Project Supervisor's internal search for suitable people. At the beginning of the project, the Business Analyst was not involved.

As the partner created a new branch for this project, all team members are new hires. The author of this paper does not have access to the employment contracts of the Partner's team members, but it is assumed that they are fixed-term contracts for the duration of the project. The partner started with a Project Manager, UX designer, QA engineer and Developer. After a year, the Partner's team had expanded to include a Data Architect, two Scrum Masters / Product Owners and three Developers.

The foreseeable need for additional hiring from the Company's point of view is to fill the Business Analyst role, as the previous analyst has moved on to relieve the PO's workload. This is partly due to the fact that the Partner's PO/Scrum

Master has not yet fully embraced the business principles and scope of the product, leading to misunderstandings. In addition, as the time for extensive UAT is approaching, the need to hire a dedicated person to handle the activities associated with it is imminent.

The partner's team is otherwise in order, except that the progress of the product has reached a point where the QA engineer can no longer carry out all the necessary testing alone and needs additional help to share the load.

At the start of the project, the total number of team members was eight. With current and projected additional hires, the team size will be approximately 17 people by the end of the project.

Tips:

- It is advisable that team members are included into the project from a need-basis, and similarly downscaling should be exercised when the role’s importance diminishes. In the case of distributed teams the downscaling should always primarily mean subtracting the affected team member’s workload to e.g. 50% rather than cutting the role off altogether. This is because the person might commit very quickly to another project full-time, and is a lost resource in the case of re-emerging need.
- Product Owner is the main medium between the customer and the development team, which is good for consistency, but also leads to customer feedback being filtered. This phenomenon should be acknowledged.
- If the Product Owner role is doubled with the role of Scrum Master as the responsibility of a single person, the workload can likely grow to be so large that it will cause a bottleneck.
- When considering a partner in a distributed team -situation, the issue of time difference should not be underestimated. If the difference is large, responses are always delayed to the next working day with both parties. Additionally, certain meetings such as daily stand ups are not possible to

attend and Mondays and Fridays (40% of possible meeting days in a week) might not be suitable days for larger meetings. If the common suitable meeting times are limited, it is recommendable to facilitate workshops.

- If there is only one representative for each team role, mishandling or poor competence in any role is rapidly affecting production and require immediate intervention or personnel change.
- If the team is small and unilateral, workload estimation can be misguided, when no one contests anyone's initial estimates.
- If there are multiple teams, their management cooperation needs to be emphasized so that the architecture is common and sprints are in synchronization (see Scrum of Scrums, Ch. 8.4).
- Domain knowledge transfer from expert to team is a long process and might need constant checking for regression. From practical experience, a recommendable method of knowledge transfer is facilitating workshops. Non-recommendable approach is providing the team with large quantities of initial data for independent studying.
- If there is a need to scale the team size up, it is important to acknowledge that the velocity of team may drop for some time, because the current members need to onboard and advise the new ones. Therefore, team additions should not be applied as a remedy if the project is already late, because the result could be an even longer delay.

5 Software development process division

According to SWEBOK (IEEE 2014, 153), “A software development life cycle includes the software processes used to specify and transform software requirements into a deliverable software product”. The processes are (IEEE 2020, 23; IEEE 2014, 151-153): requirements, design, construction and testing. The following chapters present the main themes of each process. In an agile environment all of these processes are occurring concurrently with each evolutionary iteration, but when expanding the view to the whole development project, the same processes can be identified also as the major stages, as shown in Figure 16. In this view, the stages must happen more sequentially because their individual outputs are the next stages’ input. In real life however, all four processes are happening at the same time, but depending on the point in the software development life cycle, the activities of one process is emphasized over others. The duration of this lifecycle is fully dependant on the size of the effort, but as a basic rule of thumb, Resnick et al. (2010, 22) suggest that a major release with an agile approach takes between 6-12 months.

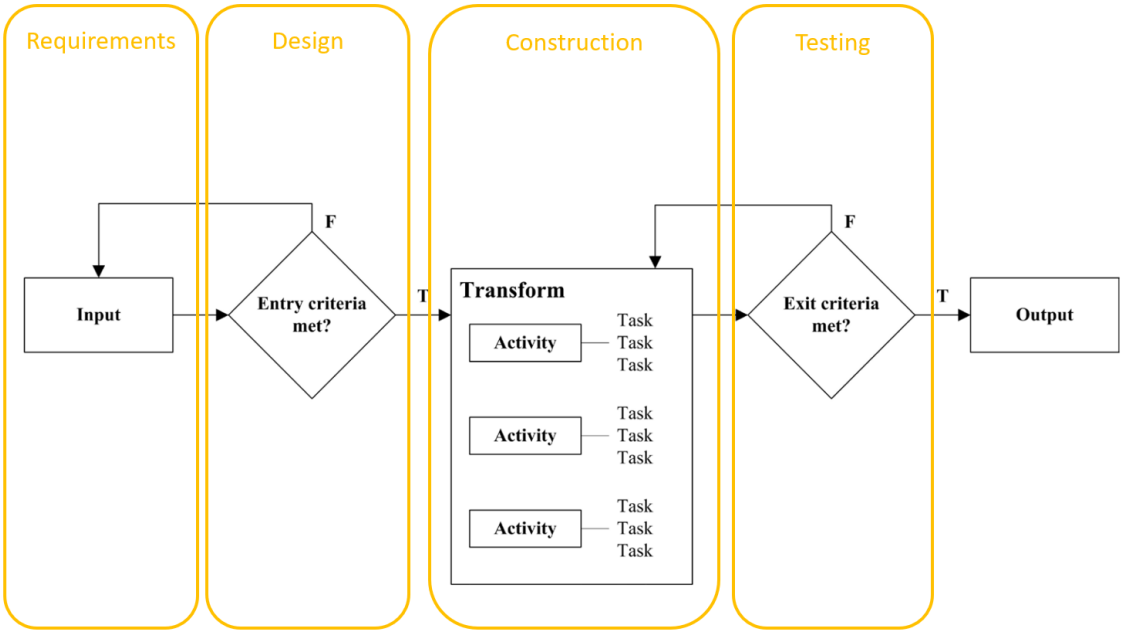

Figure 16. Elements of a software process (IEEE 2014, 150), expanded with orange boxes by the author to reflect the software development processes.

6 Requirements process

According to SWEBOK (IEEE 2014, 32), "Software requirements express the needs and constraints placed on a software product that contribute to the solution of a real-world problem". Software requirements can be divided into functional (representing needs) and non-functional (representing constraints). Functional requirements are a collection of functionalities that must be successfully executed by the software. Non-functional requirements concern the quality aspects of the software, such as performance and reliability. Non-functional requirements are sometimes referred to as quality requirements. What they have in common is that they must be in a form that can be validated. (IEEE 2014, 33-34.) Validating functional requirements is, on average, easier because they are either executable or not. Creating a non-functional requirement in a form that is quantifiable and verifiable is more challenging. For example, "The response time of the software for the search function shall be fast" is a subjective requirement and very difficult to measure. A better requirement would be "The response time of the software for the search function shall be less than X during any hour of operation".

Requirements are often gathered in an interdisciplinary process involving several stakeholders. This can lead to a collection of requirements that is overstretched in terms of time and budget. Therefore, the process involves negotiation. Software requirements are often prioritised to assist in these negotiations. For project management purposes, requirements also often have statuses so that their progress can be monitored. (IEEE 2014, 33.)

When talking about requirements, an important distinction is that software requirements are product requirements, but the project itself has it's own set of requirements, called process requirements, or in practical life, project requirements. These requirements define how the project will be run. An example might be "The project risk assessment -document will be updated monthly".

6.1 Activities during the requirements process

Requirements elicitation and refinement is a living process that continues throughout the development life cycle, with decreasing intensity. However, because the product requirements form the basis of the entire software and therefore shape the project scope, the intensive part of the work must take place before any other phases. IEEE (2014, 33) divides the requirements process into four activities (Figure 17).

Figure 17. Requirements process' activities (IEEE 2014, 33).

The first activity is called requirements elicitation or requirements capture. This activity initiates communication between internal and external stakeholders to understand what is expected of the software. The expectations are shaped by goals, business rules, operational and organisational environments. (IEEE 2014, 36-37.) The Product Owner should act as the main point of contact for communication, as part of his/her role is to have the best overall understanding of the product from all angles. A critical consideration is the equality of stakeholder requirements. Some ideas about the balance of power:

- If the technological angle (Represented by Scrum Master & Development Team) has more decisive power related to others, there is a risk that though the end product is “watertight” and works flawlessly, it does not meet all the business requirements set to it, i.e. does not fulfil all of the reasons for which it was developed. Second considerable risk is that the project timeline and budget are exceeded, when the development has concentrated on fine-tuning technical details instead of completing the whole product.
- If the Business feasibility angle (customer, management) has more decisive power related to others, the product’s list of requirements is excessive and not possible to achieve in the allotted time. Another risk is that these requirements are missing crucial technical elements of the software, which are not part of Business domain knowledge.
- If the Project Management angle (Project Manager, Program Manager) has more decisive power related to others, the project might be completed in time and within budget, but the product created as the end result might be considered lacking in overall quality, or unstable on account of corner-cutting.

Stakeholders rarely have a complete list of requirements, so they have to be elicited using various techniques, such as workshops, meetings, interviews and prototypes. There are also tools designed for requirements elicitation, an example of which is a use case diagram (Figure 18), a subclass of UML behavioural diagrams. A use case diagram focuses on the high-level external functions that occur in different use cases by different users. It

visually represents actors, associations, and system boundaries. (Aleryani 2016, 124-126.)

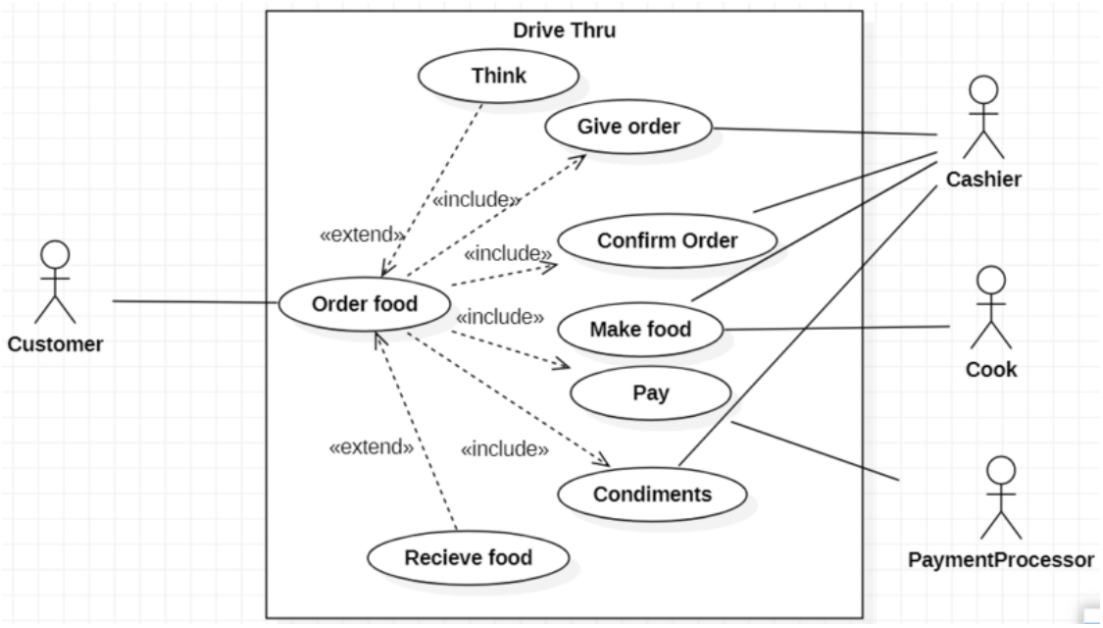

Figure 18. Example of a use case diagram (Gonzalez 2022).

The requirements analysis activity begins to organise and classify the collected requirements, as well as find new emergent requirements that enable the original ones. In the analysis phase, the negotiation of the main requirements begins. The importance of the presence of experienced software developers in this phase is emphasised because, at a technical level, some requirements may be contradictory or require an unreasonable amount of effort to create in relation to the benefits gained in the software. Experience is also needed to get an overall picture of the high-level interactions and initial architecture of the software. (IEEE 2014, 38-39.)

The requirements specification activity documents the results of the previous stages. The nature of the documentation is such that it can be systematically evaluated. Depending on the software, 1-3 documents are produced. Highly complex software, especially when interfacing with specialised hardware, will produce separate system definition and system requirements specification documents. However, regardless of the level of complexity, the one common document for all software is the software requirements specification. (IEEE 2014, 41-42.) The software requirements specification represents the requirements agreed between the stakeholders. In addition to natural language, the document may contain more formal descriptions and diagrams. The main goal of this approach is to produce the information in the most precise way possible. The software requirements specification is the first formal document that can be used to estimate project scope and risks. (IEEE 2014, 42.)

The final activity in the requirements process is validation, where the software requirements specification is evaluated and either approved or revised. This includes the validation of any models that have been created, such as use case diagrams. Validation usually takes the form of cross-cutting reviews. It is also possible that prototypes have been created to demonstrate the functionality of e.g. certain complex or ambiguous issues, and validation is done by assessing the prototype. An integral part of requirements validation is planning how to verify that the requirements are met in the final product. (IEEE 2014, 43.) This is achieved by creating acceptance test criteria, which will help to work more systematically in the user acceptance testing phase (see Chapter 9.2) and will be part of the quality plan for the project. The format of the document should be considered to allow for changes, scaling and history tracking. It is therefore advisable to use a dedicated tool for this purpose.

Once the scope is clearer, other project-related documentation can be started. Examples of these are the project plan and the risk assessment. As mentioned in Chapter 1.3, this thesis does not go through general project management procedures, including documents, but the risk assessment in a software development project has a special quality that is important to note. In addition to the traditional project risks, software development has a separate category of product risks. This means that both the process and the result are treated as separate risk factors. An example of a product risk might be that the software is too difficult for the average user of the target audience to use. Both project and product risks go through the same processes of identification and analysis, both groups can be placed on a risk matrix, and mitigation and contingency plans can be created for both groups.

6.2 Case-project insights for requirements process

The requirements for the software in the case project were initially compiled internally by the company. Although the partner was given access, comment on the requirements was sparse. Part of the reason for this was that the partner's team was initially quite small and didn't necessarily include enough experts to comment. There was also some confusion about the terms used, particularly when the aim was to produce a prototype or actual product during the scope period. The company's requirements were finalised just before the summer holidays and left (along with other additional material) for the partner to study while the company's staff were away. However, the result after the holidays was that not much progress had been made.

On the other hand, the requirements selected by the company were seen as mandatory to fulfil the company's view of a minimum viable product. Therefore, the requirements were "already negotiated", leaving little room for compromise. There was a strong emphasis on functional requirements, with non-functional requirements left to be discussed during the project.

All requirements were documented in Jira, divided into hierarchical linked structures. The hierarchy, which consists of three levels, was broken down during construction in a similar way to the work (Figures 19-21). The actual requirements represent the User Story level. They were generally written in the form of "System shall...etc". All items that were to be part of the Phase 1 scope were given the status "selected for development". They were also given priority ratings. The items did not include work estimates because the company staff did not have the domain knowledge to provide such estimates.

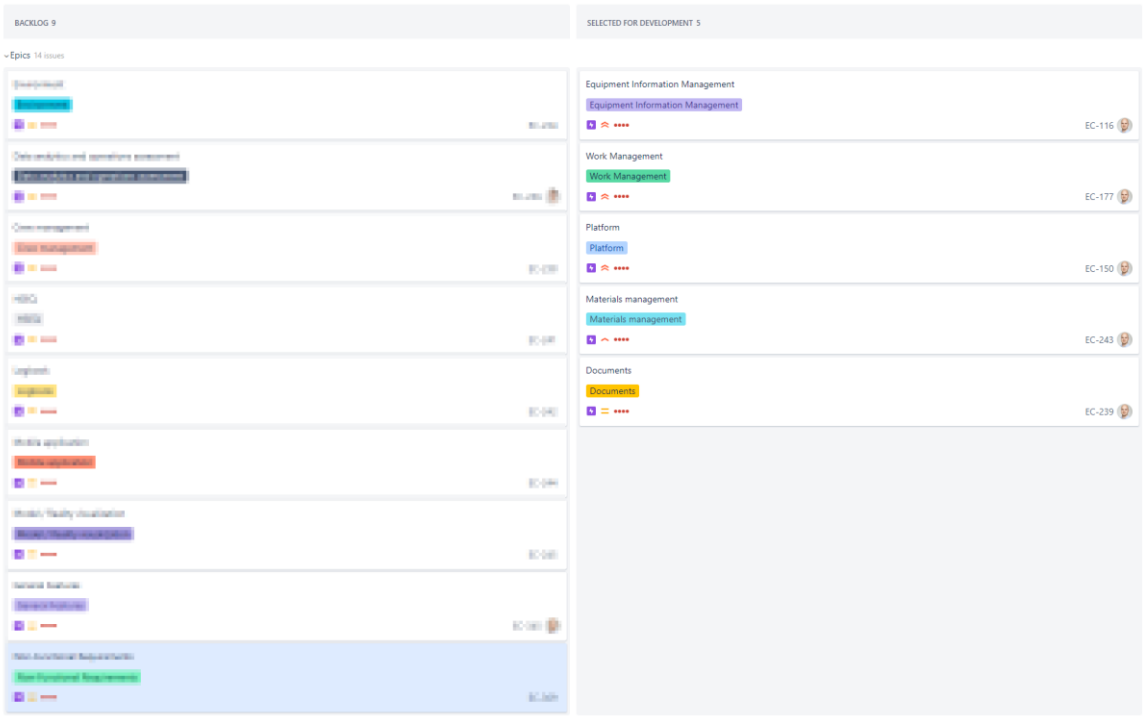

Figure 19. Case-project's "selected for development" Epics on Jira's Kanban board. Backlogged Epics are blurred because of NDA reasons.

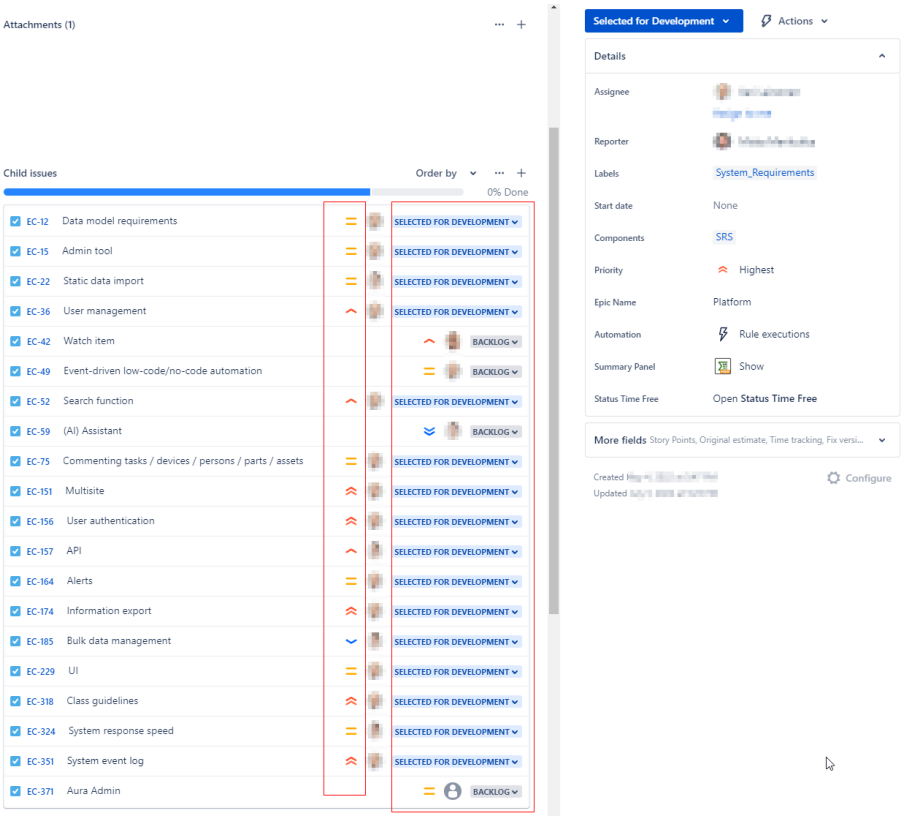

Figure 20. The Feature view after opening the platform -Epic. All Features have been assigned statuses and priorities as well as other information.

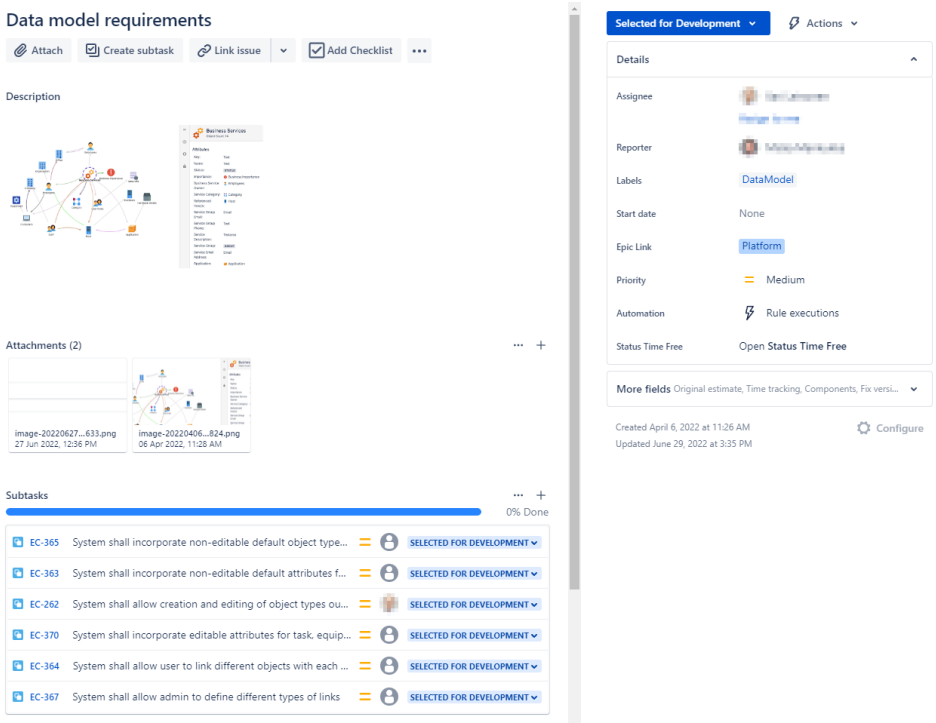

Figure 21. The view of requirements after clicking open the data model requirements -Feature. Same status designations apply.

Tips:

- Even with agile methods, there is a clear need for documentation and planning in the beginning of the project. If the development project is done with a partner, their personnel should be already present in the requirements elicitation activity. Non-functional requirements should be considered as important to elicit and document than functional.
- When negotiating the requirements, it is advisable to follow the guideline that nothing is agreed until everything is agreed. If the requirement list items are frozen one by one, there will be no room for negotiation later.
- Project scope is a logical discussion not only after requirements specification document has been produced, but after it has been meticulously inspected and approved by the software developer in order to avoid friction later.
- Risk management activities in a software development project require more resources than in a project containing only project risks. Also, the number of responsible persons for risk mitigation is likely larger.

7 Design process

The software design process can be described as a deeper iteration of the requirements process as solution evaluation and negotiation continues. During this process, various UML models are created to define the product more precisely and to provide alternative solutions to aid decision making. The ultimate goal is to provide usable input to the design. Standard 12207, Software Life Cycle Processes (IEEE 2020, 74-81) lists two distinct processes that fall under the design process: architecture definition and design definition.

7.1 Architecture definition

The purpose of architecture definition is to identify the components of the software system and their relationships to each other, so that all stakeholder concerns are addressed. A function of this process is also defining the external interface and boundaries of the system. The architecture should be as design-agnostic as possible and should remain an unchanging baseline throughout the development process, even though the software design may change. This is possible because the architecture definition deals only with those requirements that relate to the architecture, whereas the design definition process must take all requirements into account. A class diagram is a typical representation of the structural diagram type for architecture definition (IEEE 2014, 40). It shows different types of objects needed for the software, called classes. It also shows the relationships or associations between each class (Kasurinen 2017, 22). Classes are shown as rectangles with three compartments, each presenting specific information about the class. The compartments are name, attribute and operation. Associations are shown as different types of lines/arrows between the class boxes (see Figure 22). The main advantage of class diagrams is that they contain information in a form that is applicable to many programming languages, i.e. it is possible to generate code based on class diagram information. (Otero 2012, 43.).

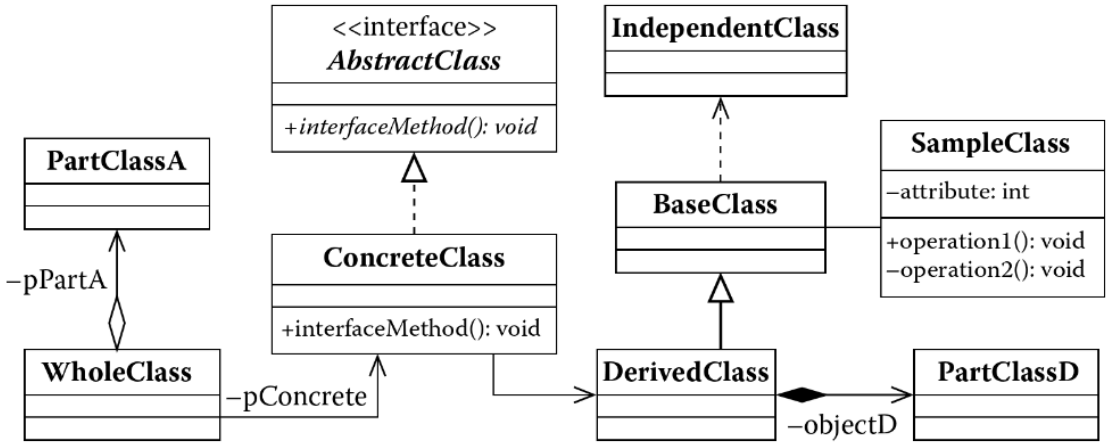

Figure 22. Example of a class diagram (Otero 2012, 45).

Another example is a state diagram (also called state machine diagram or state chart), which functions as an example of a behavioural diagram (See Figure 23). Whereas the class diagram is static, the state machine diagram is dynamic. The emphasis here are the different states an object on each class can have and the flow from state to state (IEEE 2014, 58). This is achieved by showing how an object responds to instances (Swain et al. 2010, 6-8).

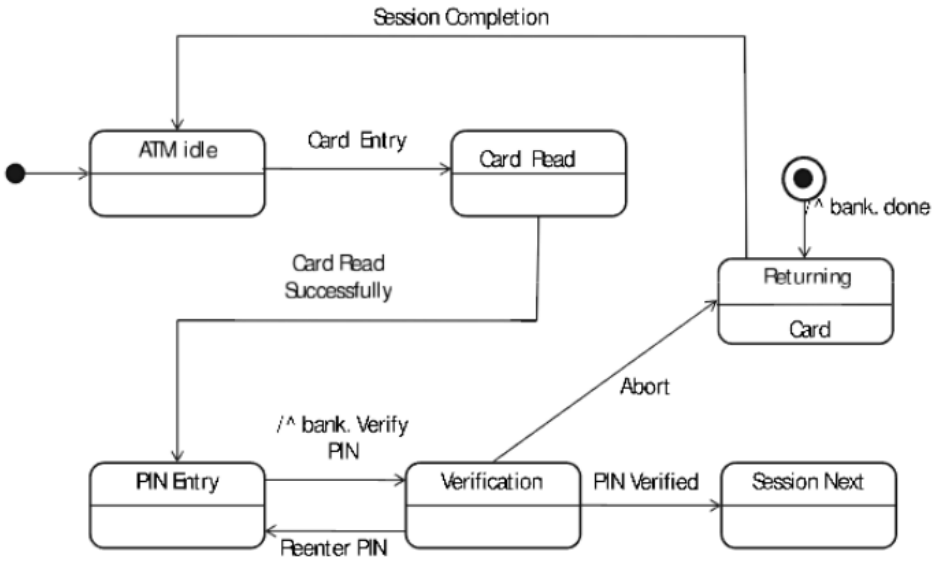

Figure 23. Example of a state diagram (Swain et al. 2010, 7).

7.2 Design definition

The design definition process has the same aim of increasing the level of specificity. It uses the architectural definitions as a basis for designing the behaviour and characteristics of each system element. The emphasis is on compatibility. The products of the design process should provide information that is accurate enough to start development. An example of a UML diagram where the interaction study has reached a deep level is the sequence diagram, as shown in Figure 24. The sequence diagram is also a behavioural and dynamic model, but it focuses on specifying object interactions, taking into account the order in which messages move (IEEE 2014, 58). The typical reading style of sequence diagrams is left to right and top to bottom (Otero 2012, 59). The diagrams show the event flow of each action as a step-by-step path.

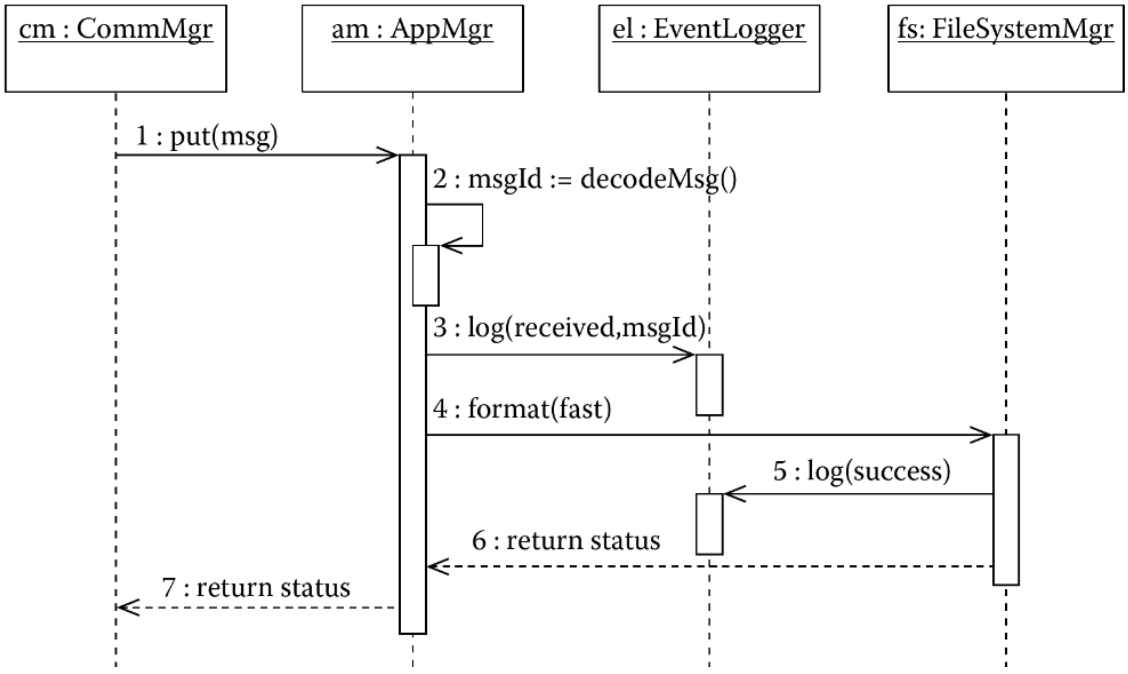

Figure 24. Example of a sequence diagram (Otero 2012, 60).

It may be considered unnecessary to create all of the above diagrams in the project because as documents are prone to change, maintaining and synchronising multiple diagrams can be challenging. This effort is nervetheless recommended because alternative abstractions complement each other and provide different perspectives on problems. (IEEE 2014, 228.)

The design process should also serve as a stage for making decisions that affect the subsequent construction process. When the whole system is known and documented, decisions can be informed and logical. IEEE (2014, 68-69.) identifies the following decision paths to be taken that have an impact on construction

- Communication protocol
- Programming languages
- Coding standards
- Tools

In addition to the above, it is also beneficial to make early decisions about service providers, such as cloud hosts. Efforts should also be made to assess whether there are parts of the system that can be bought off the shelf, rather than having the team build everything themselves. It is possible to use many open source software (OSS) and commercial off-the-shelf (COTS) products to significantly reduce the workload and enable scope fulfilment. COTS products in particular are often very well made and easy to implement. On the downside, off-the-shelf solutions may contain outdated code, so informed choices are key.

7.3 Case-project insights for design process

Both the requirements and design processes were considered as a common “definition phase” in the case project. In terms of the elements and activities presented in the literature as belonging to the design phase, the project products fell short. The efforts of obtaining documentation from the partner at the beginning of the project were not particularly successful, the scarce documentation which was received did not exceed even the requirement of JBGE. This caused a considerable amount of friction between the personnel of the company and partner. The possible root cause for the problem is likely the fact that Data Architect was not involved in the project since the beginning, and other members that were, did not consider document creation as part of their work, so the Partner’s Scrum Master / Product Owner did not have anyone to assign documentation tasks to besides himself. When the Data Architect became involved, his diagrams did not follow the principles of UML, and did not produce additional value when compared to the initial data gathered by company personnel. This lasted for several months, because the message from the partner’s side was that the obscure diagrams would start to “make sense” when they were specified more. This never happened, and the Data Architect position had to be reassigned. In retrospect, a surprising amount of the documentation provided by the partner was not UML compliant. On the other hand, this was not part of the process requirements for the project. Some rudimentary diagrams (see example in Figure 25) were provided by the partner's Scrum Master, but they were close to the obvious level and contained errors that persisted for a long time.

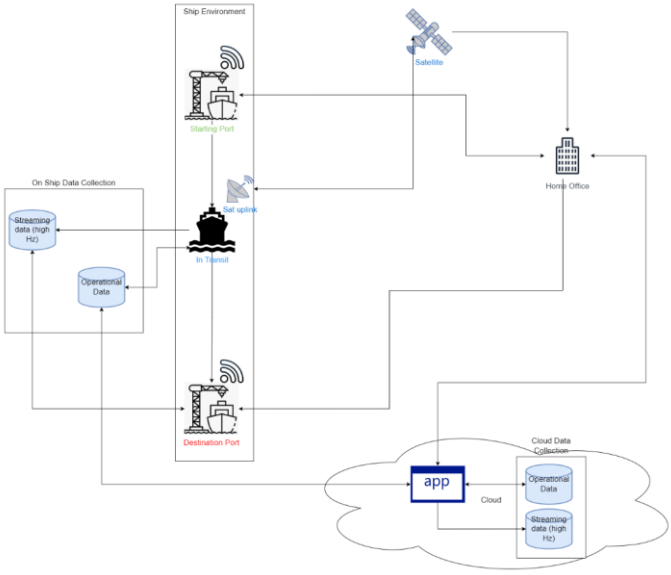

Figure 25. Example of the quality level of diagrams provided by the Partner during so called definition phase.

For the project, the message of the Company was for the team to utilize COTS and OSS components. Company's suggested targets for ready solutions were e.g. calendar and Gantt chart. It however turned out that the Partner was not especially eager to utilize COTS solutions, at least in such small pieces. The Company expected the Partner to provide necessary information for selection of suitable areas for using ready solutions, and additionally recommendations on the best solution. This proved to be unfeasible by the Partner. Some propositions were delivered, with highest recommendation that the Partner creates all aspects by itself. The recommendation did not however take into account the timeline and budget of the project and thus proved to be of low value.

Tips:

- Data Architect is a high-value role at the beginning of the project, therefore it is essential that the person selected for it has extensive experience and verifiable domain knowledge.
- Generally, developers are not necessarily equipped with the skillset and interest to create documentation. This should be discussed preferable in contract stage.
- All central project documentation should be required to follow UML standards.
- UML diagrams will not necessarily be created if the team is encumbered in work. It is advisable to assign responsibilities for document creation for the most feasible team members as early on as possible. These should be included as top-priority items in the backlog to ensure that time is allocated for creating it.
- Even though the software development company might be experienced, if the development team has not worked together before, the learning curve might require a certain length of time. A well-established company Best Practices -guideline helps in this aspect.
- It is possible to utilize COTS and OSS components, so the software does not need to be entirely built by the team
- Also with the software's UI, It is logical to adhere to well-established principles, such as IBM Carbon Design System because of their intuitiveness. However, as other products have likely done the same, many software products are homogenized and resemble each other in their design.

8 Construction process

The agile approach to software development is arguably at it's most concrete during the construction process. It poses some challenges to separate construction as an independent activity because of the ongoing concurrent design and testing of the constructed items. At its core, software construction is defined as "the detailed creation of working software..." (IEEE 2014, 66). The creation of software elements or units is done by writing code and integrating the created elements together to form the software build. It is common for the code to include notations to guide the work of other programmers and to later produce documentation, such as a user manual. The construction process typically produces the largest amount of documentation (IEEE 2014, 66). The construction of elements and their integration are considered by IEEE (2020, 85-91) as two separate functions, called the implementation process and the integration process, but from the project manager's point of view this distinction has little difference.

8.1 Construction process metrics

Software projects are no different from other development projects in the sense that both process efficiency and effectiveness are of interest to project management. Software process efficiency compares actual resource consumption with expectations, while effectiveness is the ratio of actual to expected output. Even if a process is highly effective, it may not produce efficient results and vice versa. (IEEE 2014, 156.) Measuring efficiency is always strongly dependent on the right context. A simple example of this might be: the number of user stories completed per sprint has increased, implying that the team's efficiency has increased. However, if it’s considered that the team has recently recruited two new members, it’s noticeable that the effectiveness has not increased enough in relation to the new efficiency of the team, in fact it has decreased.

8.1.1 Velocity chart

A basic measure of team effectiveness is velocity, i.e. the number of User Stories the development team is able to deliver per sprint. If the team size remains constant, the velocity should either stay the same or increase over time. This is in alignemt with the agile principle of keeping a constant pace indefinitely (Kent et al. 2001). However, this is somewhat dependent on the consistency of the sprint plan content. The velocity chart tracks the velocities of each sprint, so a trend can be established. It can also show the relationship between the number of story points completed and the number of points planned for each sprint, as shown in Figure 26. This comparison gives an idea of how accurately the team is in estimating it's performance in advance.

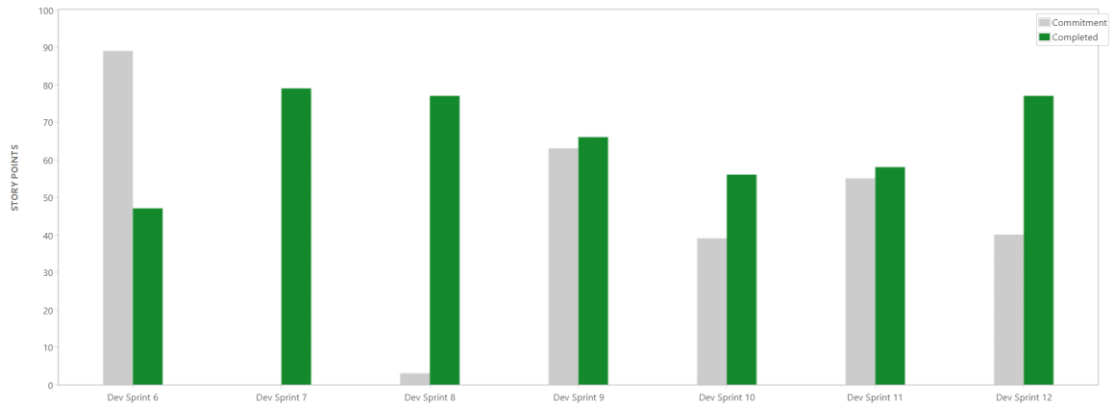

Figure 26. Example of a velocity chart.

8.1.2 Cumulative flow diagram

Cumulative flow diagram shows the statuses of User Stories within a selected time-box (Figure 27). It is good for determining bottlenecks, i.e. certain interim status is prevalent. An especially interesting aspect is comparing the amount of completed User Stories to the number of stories added to the backlog during refinement of Features. By the ned of the project, the amounts should be equal. If the amounts of added stories starts to grow, corrective measures are needed.

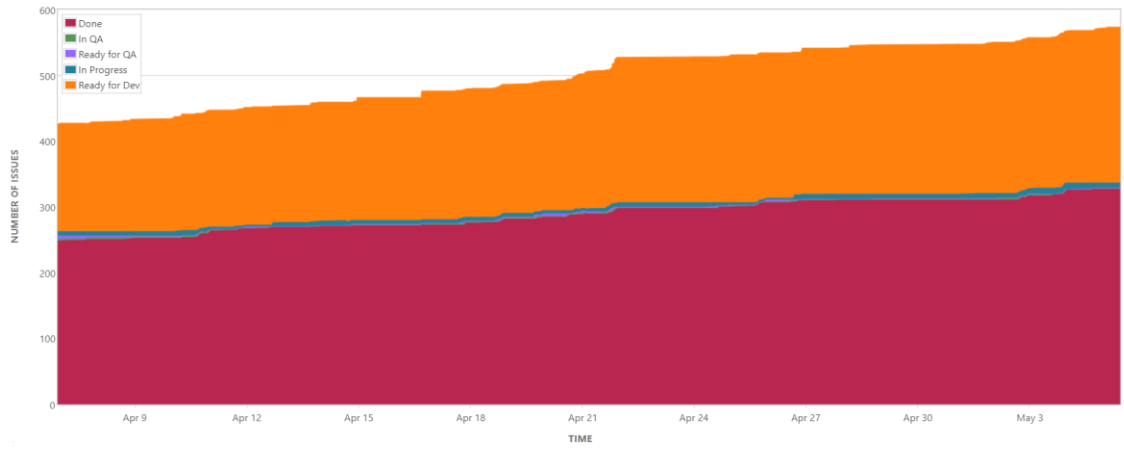

Figure 27. Example of a cumulative flow diagram (indicating that the number of developers could be increased).

8.1.3 Burndown and epic burndown charts

The burndown chart shows the percentage of activities completed within each sprint, as shown in Figure 28. Particularly when team members report only fully completed tasks rather than remaining story points per task, the information presented is highly contextual (e.g. there may only be large items under construction, so the burndown appears static until near the end of the sprint). From a Project Manager's point of view, the burndown chart is of little value and

is a metric more suited to the Scrum Master. On the other hand, the Epic Burndown Chart is more useful for management. It shows how many items within an epic have been completed in each sprint, how many items have been added and how many sprints it has taken in total to complete an Epic. If the Epic has not yet been completed, the chart provides the historical data needed to make a prediction. An example chart is shown in Figure 29.

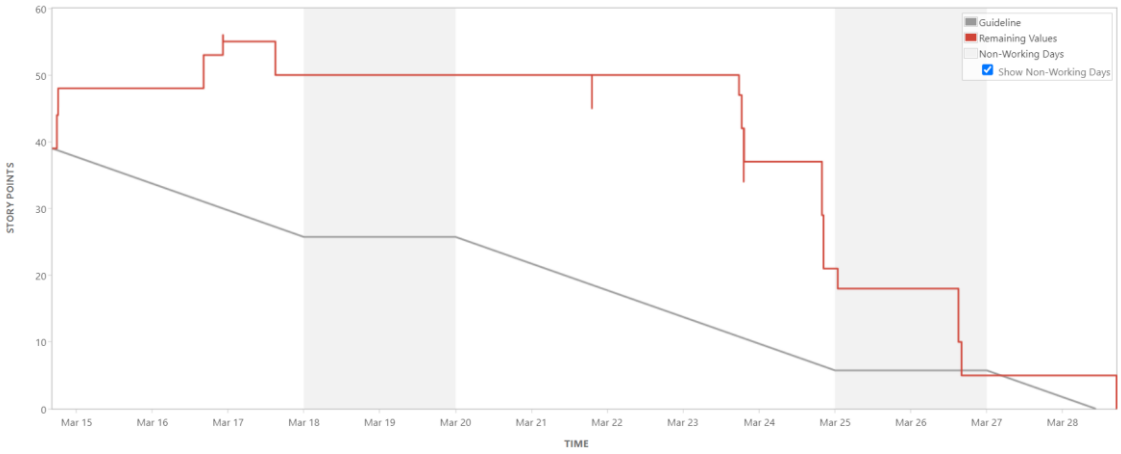

Figure 28. Example of a burndown chart.

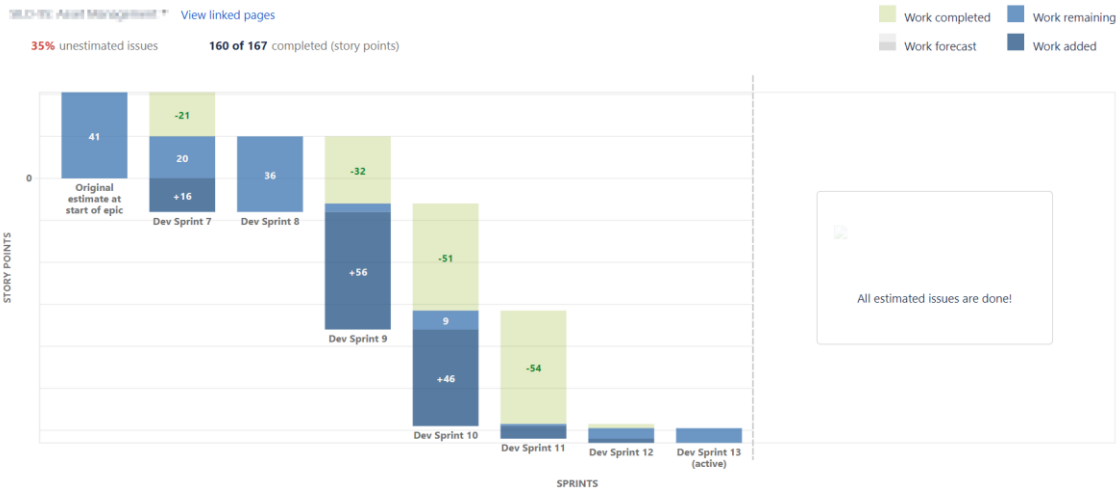

Figure 29. Example of an Epic burndown chart.

8.2 Environments

During this process, the software build resides in several different environments. The number and purpose of these environments will depend on the culture of the organisation and the type of software. Typically, the build is constructed and modified in a development environment. For consistency, only selected members of the development team usually have access to the development environment. The build environment may also contain 'work in progress' material, or messy mock-ups that are not meant to be seen by anyone other than the programmers themselves. This depends a lot on the version management approach. (IEEE 2014, 77.) For testing purposes, the build is copied to a separate environment, called the testing/QA/beta environment. All sorts of testing can be done there without risk of adversely affecting the build in the development environment. It is already possible to start User Acceptance Testing in this environment, but because it is different from other types of testing (see Section 9.2.5), it can also be done in a dedicated UAT environment, also called a staging environment. This environment contains mature software, with most bugs and problems already found and fixed. The UAT environment can be used to demonstrate the software to potential customers and to obtain feedback from end users. The final environment, where the software is "live" and functioning commercially, is called the production environment. (IEEE 2014, 90-91, 113.)

8.3 Meetings

Because development cycles are relatively short, agile methodologies are meeting intensive, especially during software development. To prevent meetings from taking up too much of the development time, some are designed to be very short. The exact number and function of meetings will depend on the choice of methodology and the organisation's preferences. There are also other types of meeting that are not directly related to a methodology, but are still recommended. For example, as part of standard project management, it is

common to have weekly status meetings between representatives and the client. In addition, it is often advisable to have a high-level steering meeting at regular intervals. As this is more or less standard procedure in any field, these meetings will not be discussed further in this thesis. Because Scrum is a widely used framework and because it contains such clearly defined meeting types (called ceremonies), this thesis will present Scrum meetings as the proposed choice. The four types are Daily Stand Up, Sprint Planning, Sprint Review and Sprint Retrospective.

8.3.1 Daily stand ups

The cornerstone meeting type in a scrum framework are short daily meetings called daily stand ups. The stand up occurs at the beginning of each working day. The team gets together, and each member quickly recaps what they have done on the previous day, and what they will do next. They will also report any possible obstacles, to which Scrum Master can react accordingly. Stand up meetings are very short, with a maximum duration of approx. 15 minutes. (Cobb 2015, 42.)

8.3.2 Sprint planning

At the beginning of each sprint there is a sprint planning meeting, where the team members evaluate together the workload of the User Stories and other items that the Product Owner has selected from the backlog to be included in the coming sprint. The meeting is a negotiation because at this stage it is still possible to change the content if too much or too little is selected (Cobb. 2015, 41-42). Instead of hourly or daily estimates, the workload of each user story is often conveyed by other figures. The simplest is the T-shirt model, where the workload of items is considered to be small, medium or large. The idea behind this is that a single person may only be able to complete one large item per

sprint, or two medium ones, or several small ones. The most common method, however, is to assign story points, also called action points. The actual numerical values of the points are arbitrary, what matters is their relationship to each other. If the simplest task is worth 2 points (one point to create, another to test), then a task with roughly twice the workload would be worth 4, and so on. A suitable and used set of numbers for this kind of evaluation is part of the Fibonacci sequence: 2, 3, 5, 8, 13, 21. There are items that require no testing at all, such as a UI design, but these are rarely listed as a single task, rather several designs relating to a particular feature are bundled together to create a workload that can be estimated using the Fibonacci sequence. (Cobb. 2015, 41.)

If the calculation shows that there are too many tasks for the upcoming sprint, they can be scheduled for later sprints. This allows the planning process to span multiple sprints, so that after each sprint is planned, the following sprints often already have items in them. Therefore, sprint planning doesn't have to start from zero every time.

Especially when the team is new, or the product is different from what the team members have done before, it is common for the calculated workloads of the project's first sprints to mismatch the actual deliverables. The accuracy of sprint plans should improve as the process progresses. If the workload is miscalculated and there is too much work, the excess is carried forward to be included in the scope of the next sprint. If the workload is too low, items planned for the next sprint can be dynamically added to the current sprint.

8.3.3 Sprint review / demo

After each sprint cycle, the outcomes are demonstrated to stakeholders. This meeting is called sprint demo or sprint review. These meetings are essential for the agile ideology of obtaining feedback as early as possible. The reviews should include a representative of the customer as a spectator, so that the

created content gets validation. On the other hand, if there are comments, tasks can be created in real time and added to the sprint plan for following sprints.

8.3.4 Sprint retrospective

The fourth and final type of Scrum meeting is the sprint retrospective, where the development team goes through the completed sprint (traditionally only internally), before initiating sprint planning and a new sprint. Following the Lean ideology, the team tries to find ways to be more effective by analysing what could have been done better or what activities do not add value. The findings can very quickly be adopted as new ways of working and tested in the next sprint cycle. Obstacles are also identified and put on the Scrum Master's desk. The obstacles are sometimes divided into those that can be dealt with internally and those that require external involvement, e.g. from management. (Resnick et al 2010, 251.)

8.3.5 Backlog grooming

Another meeting outside of the basic scrum -meeting framework but closely related to it, is backlog grooming session (also known as backlog refinement), which is a re-occurring meeting, where the development team is utilized to refine the product backlog. The meetings should include the roles of Product Owner and Scrum Master at the very least. The purpose of the meetings is to facilitate the progress of development by executing several different actions:

- Backlog item’s order is changed so that the user stories with the highest priority for any given stage of the process are at the top, meaning that they are next in line to be developed.
- Large user stories are continually broken down to smaller ones, so that they can be implemented piece by piece.
- User stories are modified or re-written, so that they are more specific, understandable by the team and meet the business requirements.

- Acceptance criteria are added to upcoming user stories.
- Unclear items and issues are clarified and discussed.

In particular, the process of prioritising items is complex, as they need to be analysed from the point of view of technical logic, business feasibility and project scope. For example, a particular user story that is considered to be a high priority from a business point of view may be left out of the next few sprints in the series, because in order for the story to be implemented, a lower priority feature (technical logic) may need to be developed first as a foundation. It should be noted, however, that in such a situation it may be possible to replace the low-priority feature with a temporary fix, e.g. a stub (see Section 9.2.2), in order to develop the higher-priority story, and it is actually the lower-priority item that will be left for development in later sprints. The sessions are an ongoing negotiation between the Project Manager, Product Owner, Scrum Master and the Development Team.

8.4 Other meetings

If several issues arise during the project, they can be dealt with in separate, dedicated meetings involving only the relevant people. This is to avoid taking too much of the whole team's time away from the development work. The separate topics can be, for example, architecture, UX, QA, business, etc.

In a case where several separate tracks are being progressed at the same time, the derivative is that project personnel also consist of several development teams/pods. It would be against agile principles to organise all the meetings for such a large number of people, as they would take much longer. In these cases, a coordination meeting called the Scrum of Scrums is established. This meeting brings together a representative from each team to discuss and ensure that the development paths are in harmony and that the product is coherent. (Resnick et al. 2011, 38).

8.5 Spikes

Spikes are a method of dealing with impending obstacles or complex challenges in a software development project. The method consists of relevant team members (can be anything from one person to the whole team) taking time out from creating software content to solve the problem or make preparations. The time-out can be arranged to occur either between sprints or during sprints. If the problem to be solved requires all or almost all team members, and especially if the challenge is very close to the future pipeline, it is preferable to address it as quickly as possible, with a spike between sprints. The duration of the spike is usually less than a sprint, with 50% being a good rule of thumb. If the spike involves only a few team members, and the challenge does not pose a risk of negative impact in the near future, spike activities can replace some user stories in the sprint plan, and the spike can last several sprints. In this way, production is not halted. (Resnick et al. 2010, 263-266.)

Case-project insights for construction process

8.5.1 Item status workflow

The first status a user story must have in order to be placed on the Sprint Plan and Scrum Board is "Ready for Development". Before that, however, the story has a separate round in which the Scrum Master, i.e. the most potential creator of the user story, moves it to be validated by the Product Owner. This adds status possibilities, as shown in Figure 30. Only when it is confirmed that the story is coherent and meets the business requirements will it be moved forward. The Product Owner also makes the final decision on the priority of the story, either moving it to the bottom or the top of the backlog. During construction, the story passes through five states (see Figure 31), the last of which is Done, a fully tested and viable increment of the product.

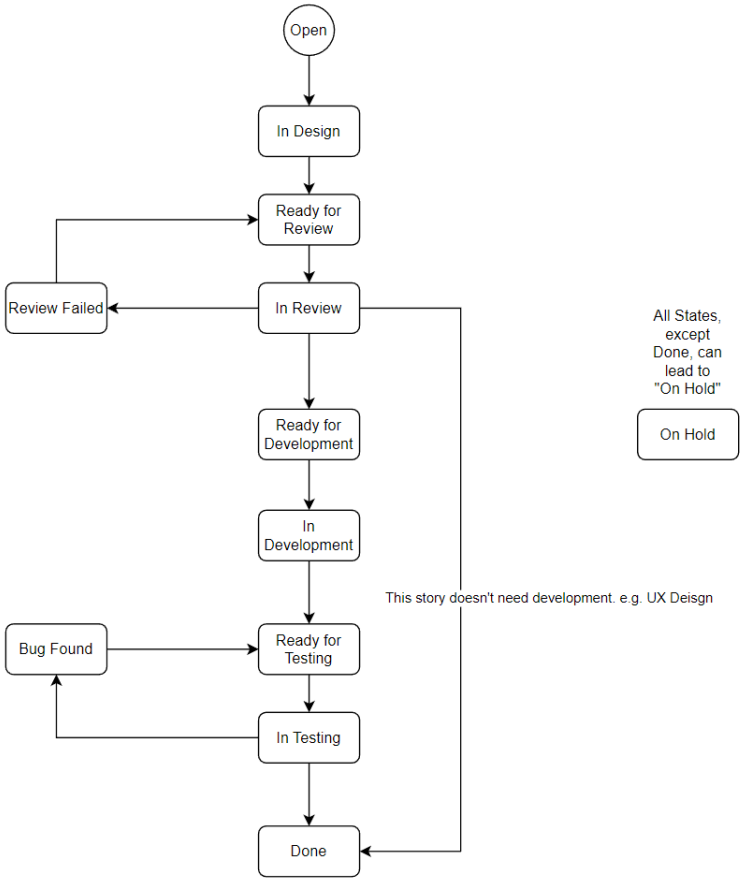

Figure 30. All possible statuses, and the workflow of a user story in the Case-project.

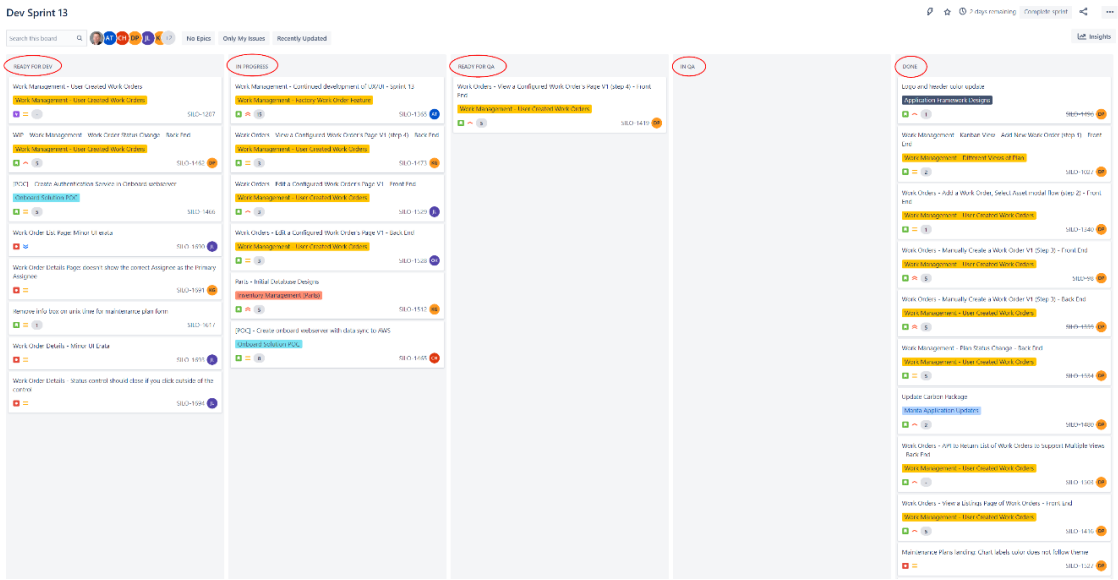

Figure 31. Kanban board view of an ongoing sprint. The statuses are: Ready for dev, In progress, Ready for QA, In QA, Done.

8.5.2 Case-project insights for construction process

All the types of meetings listed in chapter 8.3 were used during the case project. Due to the time difference of the partner, the meetings were mostly held in the evenings during the week, Finnish time. This meant that the company's staff had a considerable burden of evening meetings, although these were distributed between the different roles. The weekly meetings were the only common meeting for all. Fortnightly they were accompanied by Sprint Demo and Sprint Planning meetings. There were also separate backlog grooming sessions, architecture meetings and test status meetings. There were high-level steering meetings, but they were held at fairly long intervals. Occasionally, there were additional project meetings attended by Project Managers from both parties.

Company personnel were not able to attend the daily stand-ups. In addition, sprint retrospectives were held internally by the partner's team. On the other hand, company personnel had their own internal weekly meeting and several ad-hoc meetings, e.g. on marketing, brand, sales, organisation, etc. To avoid disruptions in the flow of information, the whole team was kept informed of general issues by providing memos or creating new Confluence pages on the topics discussed in the meetings. Example documentation is shown in Figure 32.

Start doing	Stop doing	Keep doing
•	•	•
When appropriate, add notes to Bugs explaining what's been changed in order to gauge scope of retests needed.	Over committing sprint scope.	Periodic F2F meetings to discuss more complicate functional pieces. 🅰
More time to break down the scope and review the UX side by side the story	Too much stuff in the sprint.	Keep the new story format and continue to tweak it
Stories for AT? 😿		Onboarding new people went well, I guess?
Would it help to break down stories to the subtask level? For "better" estimation?	not assign stories for testing towards the end of the sprint	Document decisions and specification in Confluence → make it the source of truth
Little more team effort on the data stories pre-sprint	Changing the story during planning. Should have already been solidified via grooming and a second check.	
Unit tests and documentation		helping other team newer member(s).
Start figuring out a plan for user documentation leveraging all our content		[redacted] seems to be working but can be tightened up.
Let's figure out the best path for working through the maintenance work and how we will break out the effort		
Prune old things in Confluence, or at least go through them if they are still valid		
Assign stories earlier to devs so that they can provide feedback on the story long before they need to do it.		

Figure 32. Case-example format and contents of a sprint retrospective memo.

Tips

- If the development team has access to software elements from the assets of previous projects, their identification and reuse is highly advisable for the purpose of reducing labor.
- Especially with dispersed teams, the meeting effectiveness should not be expected to be high from the beginning, but improving along the way.
- Cumulative flow diagrams may appear to be indicating stagnation, because new user stories are being constantly written as existing one are accomplished. It is more feasible to monitor the trend of created and accomplished stories.
- Sprints should preferably not be ended on Fridays, because end of week deadlines may cause stress and do not provide “slack” towards over-exteding the work (Resnick et al. 2011, 67).

9 Testing process

The output of a software project is validated through rigorous testing. This aspect is one of the main differentiators when compared to the process of a traditional engineering project. Software is a complex set of components and the code that controls their function. The validation criteria for software are mostly concerned with the programme behaving in an expected way with all selected execution possibilities (IEEE 2014, 82). This type of validation requires a structured testing process. The execution possibilities need to be selected and based on probability and risk severity, because even a very simple program input can be broken down into so many different variations and combinations (i.e. different behavioural possibilities) that a fully tested program is not a commercially feasible notion. According to IEEE (2020, 18), testing serves three purposes: 1. Testing detects and removes defects, thereby improving the quality of the software. 2. Testing generates information to support continuous improvement. 3. Testing builds stakeholder confidence. The activity of fixing defects found by testing is called debugging, but this activity is not grouped with the testing process (Homès 2012, 11).

It is important to note that, according to Kasurinen (2017, 9), on average only 10% of all testing work is automated. This means that most of the work is done manually, making testing a significant cost factor in software development. In contrast, the sooner a bug or error in the program is discovered, the cheaper it is to fix. The biggest risk of inadequate testing is the risk of a program that doesn't work.

In a software development project, the responsibility for managing testing usually lies with the QA Engineer. The amount of project documentation related to testing has several levels, such as the organisational level, which produces high-level documentation, such as the test policy. However, IEEE (2022. Part 1, 24) states that this level of documentation is usually produced in larger and/or

more technologically mature companies. It is perfectly possible to run a project without it.

This thesis focuses on the project level, which is common to all development projects and produces the most central documentation related to testing. As shown in Figure 33, the processes are test strategy and planning, test monitoring and control, and test closure. The documents are the test plan, test status reports and test completion reports.

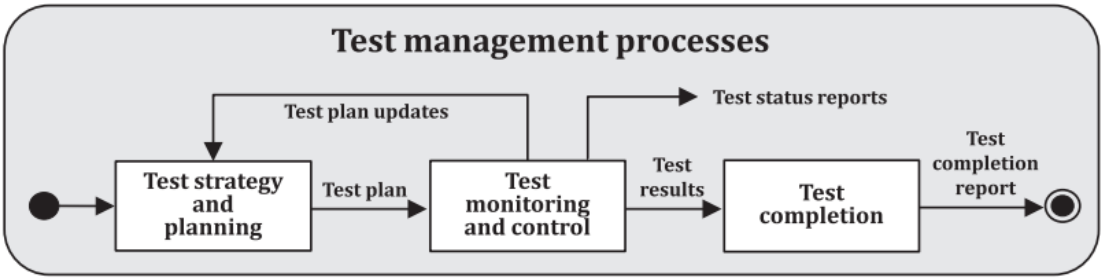

Figure 33. Test management process breakdown with central documentation on a project-level (IEEE 2022. Part 1, 26).

The test planning process forms the project's test strategy and produces the test plan document. Inputs to the plan include project plan, software requirements, risk assessment and organisational level documents where they exist. The Test Plan should be a clear indicator of the roles, tools and methods, schedule, KPIs, monitoring, evaluation and reporting practices of testing for the project. The test plan is a living document that may evolve throughout the project.

The test monitoring and control process concerns the execution and evaluation of the actual test work. It is the plan that is put into practice, and in particular the test completion criteria are scrutinised. This process may reshape the plan as the project progresses and may also produce status reports for work in progress.

The test completion process covers activities that occur after testing is complete, namely reporting, archiving and gathering lessons learned

information for the project. This process produces the Test Completion Report, which closes the loop by providing information to the organisational level about the state of testing in the particular project, as well as acting as a "sanity check" for the testing process. There can be multiple test completion reports, e.g. from different types of tests, as well as test status reports, which can be compiled at any time. It is important to note that this reporting is done throughout the testing process.

9.1 Functional and non-functional testing

At a high level, testing can be divided into two types (Kasurinen 2017, 37):

- Functional testing, also called dynamic testing is, in a nutshell testing of the program in action. This means that the program is actively used, inputs given and code executions made, in order to survey if the response is expected.
- Non-Functional testing, also called static testing is in a way, the opposite of Functional testing. Here, the system is not in use during the testing but the non-functional qualities, such as security and load-bearing capabilities are tested. Non-Functional testing also encompasses the study of the general structure and logic of the program. Non-Functional testing can be started very early on, starting from the software’s architectural diagrams.

9.2 Common testing types

9.2.1 Unit testing

Unit tests (also called component tests) are executed to individual modules of the software. The main characteristic is that the module is separate from other modules, so the testing can be isolated (IEEE 2014, 71). The tests contain criteria for success and failure, and failed test cases are flagged. Unit testing can happen whenever a module is mature enough, so the software can be tested piece by piece. The testing should happen during the same sprint as the unit's creation, and automatically whenever a code change is committed to their repository. It important to note that test scripts can and should be written already before the development of a unit, or software in general (IEEE 2014, 76). Generally, bulk of the unit tests should be executed by the developer who has created the unit, and the defects fixed instantly. The negative side for this is that a lot of defects are never reported, leading to skewed statistics. (Homès 2012, 60). Another challenge for unit testing is that the component might need the creation of one or several mock objects to simulate other, still non-existing, interacting units in order for it to work (Kasurinen 2017, 38). When testing is conducted by a developer instead of QA member, the tests more often tend to be happy-path -types, which means that the inputs to the system are those that are expected to produce the correct response from the program and possible execptional inputs are ignored.

9.2.2 Integration testing

Where there are multiple units to be tested, or where a unit and hardware can be combined, testing of their interactions can take place. This is called integration testing. Incrementally, units are added to the system and their interactions are verified through testing, eventually resulting in a complete system. If the number of units is still small, replacement components, called stubs, must be created to complete integration testing. Kasurinen (2017, 39) notes that the creation and maintenance of stubs can be the most expensive aspect of integration testing. The cost depends on the integration approach:

- Bottom-up integration: lower level components first. Low level of stubs needed.
- Top-down integration: Highest level component first, all lower level components replaced by stubs until real component is created. High stub usage.
- Sandwich integration: components are built simultaneously high and low, medium amount of stubs.
- Big bang testing: All components are put together and their integration is tested all in one go. Big bang test doesn’t utilize stubs, but is not possible to do before the project is already in an advanced stage and all components ready. The most likely use case for this kind of testing is when only small changes are made to the software.

Unit and integration testing are the two most common test types, and are actually considered as a pair under the term construction testing in SWEBOK (IEEE 2014, 71).

9.2.3 Regression testing and retesting

Testing that is performed after corrections based on a previous test is called regression testing (Kasurinen 2017, 43). However, this statement can confuse regression testing with retesting unless their end goals are clarified (IEEE. 2022, 87). The end goal of regression testing is to check for possible unintended side effects of the fixes for the system or it's parts, whereas retesting is only concerned with validating that a previously found defect has been fixed.

9.2.4 Exploratory testing

Exploratory testing is the least structured of all the types of testing presented. It is based on the tester's expert ability to search and find defects based on experience. Testing is spontaneous and heuristic, which is why it is also called experience-based testing (IEEE. 2022, 28). The work is in a way free of documentation, which means that it doesn't have to follow a created plan (Kasurinen 2017, 47-48). However, exploratory testing should take into account the major risks associated with the product and focus testing based on probability and impact. SWEBOK (IEEE 2014, 89) defines exploratory testing as "simultaneous learning, test design and test execution".

9.2.5 User acceptance testing

User Acceptance Testing (UAT) focuses on the end user's point of view and the main objective of this testing is to obtain acceptance of the software. Acceptance is obtained when the users' requirements are met. (Homès 2012, 64). Since most users do not have technical knowledge, they evaluate the user experience (UX) instead. This means the usability, intuitiveness, logic and aesthetics of the software's user interface (UI), as well as non-functional characteristics such as speed and performance. Users can make comments and compile a report. In addition, usage situations can be recorded to obtain more specific data, such as buttons pressed, search paths, etc. Importantly, UAT can begin at a relatively early stage of the project, using prototypes such as the UI demo created by the UX designer.

Very common terms in the field of software testing are alpha and beta testing. These terms refer to testing with a focus group. Often the group in alpha consists of participants from the customer's staff and possibly independent testers from the software development team. It is defined as internal approval testing (Kasurinen 2017, 47). When Alpha testing is completed, the software is introduced to a larger group of potential users, and the test becomes Beta testing. The common denominators for alpha and beta testing are that although the testing usually takes place on the actual platform of the software, the software has not yet been released, and that alpha and beta testing are usually uncontrolled due to their unpredictability (IEEE 2014, 87). Beta testing among potential customers can also be seen as a sales activity, since at this stage there should not be many bugs and the goal is more to generate interest in the product.

9.2.6 Cyber Security testing

A crucial non-functional requirement for software is that it provides security against cyber-attacks. This is an aspect that can't be considered as a separate element, but the whole system must be built according to selected security standards. Testing for preparedness against malicious cyber-attacks is usually carried out by an external party in the form of “friendly” penetration testing, which means that the external expert will attempt to access the system in various ways, but will not cause any damage if successful. After the attempts, the expert prepares a report on the level of security of the software, together with possible recommendations for further improvement. (Det Norske Veritas 2023.) There are also many types of cybersecurity certifications. According to the case-project documentation, the appropriate time to consider performing a penetration test is when the software is about 80% complete. If some high-risk features are added afterwards (e.g. document upload), it is possible to repeat the test. External cyber security services in addition to penetration testing are design and code reviews, in which the software’s structure is assessed.

9.2.7 Testing automation

Deciding where and when to use automation, and the ability to script it, requires extensive technical knowledge. The basic rule of thumb is that if a particular test is likely to be run multiple times during the project, automation should be considered. Multiple in this case means 4-20 iterations. This is because manual testing always takes about the same amount of time, whereas test automation initially requires more resources, but is much easier to repeat as many times as required (visualised in Figure 34). The most favourable target for test automation is regression testing at unit and integration level. (Kasurinen 2017, 49-50).

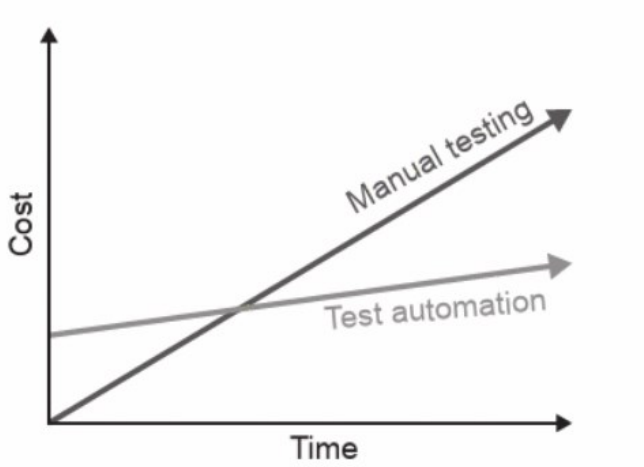

Figure 34. Costs over time between manual and automated testing (Jose 2021, 6).

9.3 Case-project insights for testing process

Testing activities (excluding developer-led unit testing) in the Case project were carried out by a dedicated QA engineer. For the first half of the project, the workload was such that only one person was required, but as the development team has grown and the system has become more complex, additional staff will be required. According to the Scrum Master, the recommended ratio of test engineers to developers on a software project is 1:2,5. The QA engineer was mostly involved with the partner's team, but in addition, a bi-weekly testing meeting was organised to summarise the testing activities of each sprint to the project management. During the Case project, unit testing was entirely the responsibility of the developers, while QA took the lead on other types of testing. This is because unit testing in the project consisted only of validating any changes made to a unit. Testing the unit's functionality was separated under the term functional testing. At the time of writing, the UAT environment is up and running and the user acceptance process is underway, both by the company and by representatives of a potential customer.

The testing documentation is extensive. The main database for the documentation is the partner's instance of Confluence. The documentation includes, for example, the test strategy, including planning for the types of tests, the test process, the automation plan, the environments in which the tests will be performed, and the test management workflow. The information is not in the form of UML diagrams, but rather text and diagrams, an example of which is shown in Figure 35.

Type	**Primary Execution Environments**	**Notes**
Unit	Dev Local, Dev	Execution is fast, so there's no reason to not execute these tests in most of the envs.
Integration	Dev, Stage/QA	Requires integrated components. Automated tests could be run in *any* environment (including Dev Local)
Functional	Dev, Stage/QA	Requires relatively *stable* code. i.e. Should be tested on essentially Feature Complete code. Automated tests could be run in any environment.
UAT (User Acceptance Test)	Stage/QA	UAT are used to determine if item is releasable. So the primary environment is Stage/QA. *However*, it is ideal that UAT are checked in *every environment* as they are the minimal requirements that must be met. It may not be feasible to test in Dev Local since you may be implementing only a fraction of the Story.

Figure 35. Example of the contents from the Case-project's test strategy - documentation, the environment possibilities for different kinds of testing.

Documentation for test monitoring, control and completion was created and stored in Jira. It took the form of sprint-specific test plans, bug lists with bug statuses, and sprint test reports. Test plans and test reports had their own item types, and items were filled as tasks and their subtasks, as with user stories, for example. As tests are executed, they are logged in the system and it is possible to create a test report from the activities during the sprint. An example of this is shown in Figure 36.

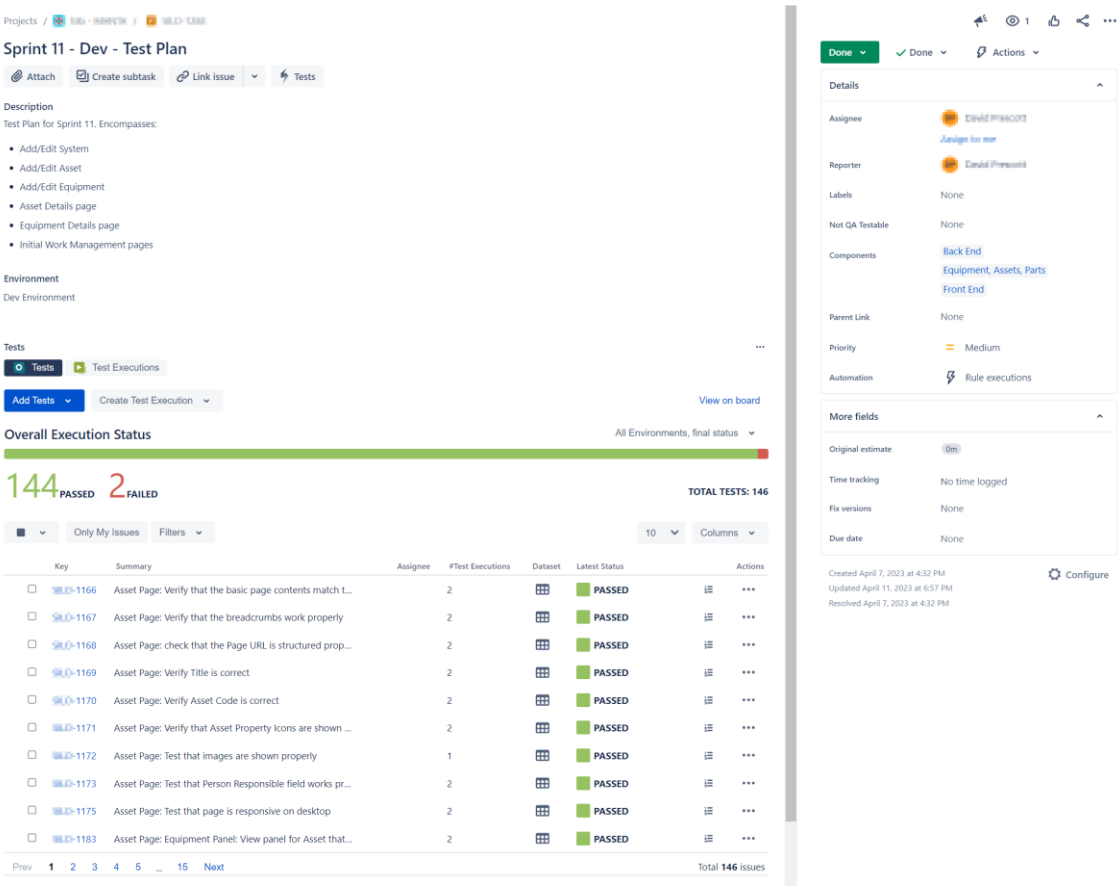

Figure 36. Example of a sprint's test plan, which can be viewed as a test report after the sprint.

Tips

- User acceptance testing is an activity which requires resources, and it is advisable to have a dedicated person doing it. Often the effort is shared with multiple persons.
- The switching of system requirements into UAT acceptance criteria is rather straightforward, so well-defined requirements for both functional and non-functional aspects reduce the workload in the testing.
- When considering the external party for e.g. penetration testing, the selection should take into consideration which candidate gives the maximum benefit for the product and is most relatable to the product's business field. In the case-project, the best selection would be a Classification Society, because they are well-known and closely related to the field.

10 Concluding assesment

The biggest challenge in writing this thesis was the breadth of the chosen topic. Attempting to narrow down the subject matter and summarise the essence of each topic in a small chapter proved difficult and carried the risk of being superficial. It might have been more fruitful to concentrate on the topic of one main chapter for the whole thesis. On the other hand, the aim of the thesis was to provide an overview from which to build. The benefit of this approach to the author's employer is arguably greater than a deep understanding of only one part of the process.

The selection of the main themes based on the content of a well-established publication proved to be a logical solution to build the framework of the thesis, but on the other hand it led to possibly too much reference to a single (albeit extensive) source. As a result, this thesis falls short if it is considered purely as a literary review. The selection of supplementary sources proved to lack a systematic approach. Many books that could be considered key literature in the field were not available without purchase, which added to the challenge of selection.

The risk of superficiality was countered by adding practical depth in the form of the case-project. The experience of an actual software development project, and access to its documentation, added a layer of pragmatism to the content that would have been lost if the thesis had been based purely on literature. Delaying the completion of the dissertation until after the case project had been completed would arguably have produced more material, particularly in relation to the construction and testing processes.

The questions raised at the beginning of this thesis have been answered, but perhaps an additional question "What are the biggest differences between theoretical knowledge and the implementation of the case project?" could have been asked and answered by highlighting the points where theory differed from practice and explaining possible reasons for the deviations.

The tips at the end of each chapter are undoubtedly useful, but are almost entirely unreferenced. Some of the tips are also conclusions that could not necessarily be drawn from the text, but rather from experience. However, they offer insights and have been included at the risk of partly decreasing the coherence of the thesis.

List of references

Aleryani, A. 2016. Comparative Study between Data Flow Diagram and Use Case Diagram. International Journal of Scientific and Research Publications, Vol. 6 Issue 3.

Alt-Simmons, R. 2016. Agile by design: an implementation guide to analytic lifecycle management. 1st Edition. Wiley.

Atlassian 2023. About Confluence. Referred 23.4.2023. https://www.atlassian.com/software/confluence/guides/get-started/confluence-overview#about-confluence.

Atlassian 2023. Welcome to Jira Software. Referred 21.4.2023. https://www.atlassian.com/software/jira/guides/getting-started/introduction#what-is-jira-software.

Cobb, C. 2015. The Project Manager's Guide to Mastering Agile : Principles and Practices for an Adaptive Approach. John Wiley & Sons.

Dennehy, S. 2009. Agile Requirements – irritation or Opportunity? Conference paper. Researchgate.

Det Norske Veritas 2023. Testing and verification. Referenced 19.05.2023. https://www.dnv.com/cybersecurity/services/cyber-security-testing-and-verification.html?utm_source=DNV-services&utm_medium=weblinkredirect

Evans, I. 2004. Achieving Software Quality Through Teamwork. Artech House.

Extremeprogramming.org n.d. The customer is always available. Referenced 21.05.2023. http://www.extremeprogramming.org/rules/customer.html

Gonzalez, N. 2022. Drive-Thru Use Case Diagram. Technical report. Researchgate.

Gross, J., McInnis, K. 2003. Kanban Made Simple: Demystifying and Applying Toyota's Legendary Manufacturing Process. AMACOM.

Holcombe, M. 2008. Running an Agile Software Development Project. John Wiley & Sons, Inc.

Homès, B. 2012. Fundamentals of Software Testing. John Wiley & Sons Inc.

IEEE 2014. Guide to the Software Engineering Body Of Knowledge. Version 3.0. IEEE Computer Society.

IEEE 2020. SFS-ISO/IEC/IEEE 12207:2020:en: Systems and software engineering – Software life cycle processes. Requires purchase. IEEE Computer Society.

IEEE 2022. IEEE at a Glance. ieee.org. Referenced 29.04.2023. https://www.ieee.org/about/at-a-glance.html.

IEEE 2022. International Standard for Software and systems engineering, Parts 1 & 3. Requires purchase. Institute of Electrical and Electronic Engineers Inc.

Infinity Business Insights 2023. Global Marine Maintenance Software Market Research Report, 2019-2030. Requires purchase.

Jose, B. 2021. Test Automation: A manager's guide. BCS Learning & Development Ltd.

Kasurinen, J. 2017. Ohjelmistotestauksen käsikirja. e-book.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. 2001. Manifesto for Agile Software Development. Referenced 01.02.2023. https://agilemanifesto.org/

Loyola Marymount University 2023. An overview of UML. Referenced 01.05.2023. https://cs.lmu.edu/~ray/notes/umloverview/

M-Files 2023. Overview. Referenced 24.4.2023. https://userguide.m-files.com/user-guide/web/latest/eng/web_overview.html.

Measey, P., Berridge, C., Gray, A., Wolf, L., Oliver, L., Roberts, B., Short, M., Wilshurst, D. 2015. Agile Foundations Principles, practices and frameworks. 1st Edition. BSC Learning Ltd.

Mobley, K. 2004. Maintenance Fundamentals. 2nd edition. Elsevier Science & Technology.

Nath, S., Stackowiak, R., Romano, C. 2017. Architecting the Industrial Internet: the achitect's guide to designing industrial internet solutions. Packt Publishing.

neurospace 2019. Condition-based Maintenance vs Predictive Maintenance. Neurospace.io blog. Referenced 28.01.2023. https://neurospace.io/blog/2019/08/condition-based-maintenance-vs-predictive-mainte-nance/#:~:text=Condition%2Dbased%20maintenance%20uses%20conditions,need%20to%20service%20your%20equipment.

Otero, Carlos. 2012. Software Engineering design: Theory and Practice. CRC Press.

Outsystems low code platform 2023. What is Rapid Application Development? Referenced 14.4.2023. https://www.outsystems.com/glossary/what-is-rapid-application-development/.

Resnick, S.; Bjork, A.; De la Maza, M. 2010. Professional Scrum with Team Foundation. Wiley Publishing.

Rosenzweig, E. 2015. Successful User Experience: Strategies and Roadmaps. Elsevier Science and Technology.

Ross, Sean. 2022. CapEx vs. OpEx: What's the Difference? Investopedia.com. referred 28.01.2023. https://www.investopedia.com/ask/answers/112814/whats-difference-between-capital-expenditures-capex-and-operational-expenditures-opex.asp.

Schwaber, K.; Sutherland, J. 2020. The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game. scrumguide.org. Referenced 07.05.2023. https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf#zoom=100.

Swain, S.; Mohapatra, D.; Mall, R. 2010. Test Case Generation Based on State Activity Models. Journal of Object Technology 9(5):1-27.

Uppuluri, K. 2018. Enterprise Asset Management (EAM) vs. Asset Performance Management (APM). Article. LinkedIn.com. Referenced 29.01.2023. https://www.linkedin.com/pulse/enterprise-asset-management-eam-vs-performance-apm-krishna-uppuluri.

Westfall, L. 2016. The certified software quality engineer handbook. ASQ Quality Press.