Software Project Life Cycle Handbook

Jaska Ahlfors

Master’s thesis
December 2024
Information Technology, Full Stack Software Development

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Ahlfors, Jaska

Software Project Development Life Cycle Handbook

Jyväskylä: Jamk University of Applied Sciences, December 2024, 55 pages.

Degree Programme in Information Technology, Master’s thesis.

Permission for open access publication: Yes

Language of publication: English

Abstract

Software projects often follow a structured pattern, Software Development Life Cycle. While different teams and different projects have different methods and different workflows, the process should be more effective, if there is not too much variation inside the organization.

The aim was to document and standardize the commissioner organization’s software project processes, so they could be efficiently taught, followed and eventually improved. The resulting documentation, the handbook, was supposed to be viable for to be used as a template for another organizations and teams as well.

The handbook was to be implemented by applying Design Science Research methods by iteratively and incrementally adding and reviewing chapters to the handbook and pilot testing the resulting artifact with the commissioner organization’s software development team.

The resulting handbook successfully documented and improved the teams software development process and was deemed to be useful as a tool for later projects as well. The handbook was also seen viable for to be used as a template for other teams as well, with modifications and adjustment to make the actual processes and activities fit those of the adopting organizations.

Keywords/tags (subjects)

software development, software development life cycle, software project management, design science research

Miscellaneous (Confidential information)

Contents

Glossary ... **3**

1 Introduction ... **4**

1.1 Background And Motivation ... 4

1.2 Research Method ... 4

1.3 Research Process ... 5

1.4 Research problem and goal ... 6

2 Knowledge base ... **7**

2.1 Software Development Life Cycle ... 7

2.2 Defining and Improving Process ... 9

2.3 Configuration Management ... 11

2.4 Artificial Intelligence ... 12

2.5 Tools and software ... 14

3 The Handbook ... **15**

3.1 Project workflow ... 15

3.2 Planning ... 18

3.2.1 Schedule ... 19

3.2.2 Risk Assesment ... 20

3.2.3 Selecting the SDLC Model ... 21

3.2.4 Configuration Management plan ... 23

3.3 Gathering and Analyzing Requirements ... 23

3.3.1 Requirement Management ... 23

3.3.2 Writing Requirements ... 24

3.3.3 Traceability ... 26

3.3.4 Use Cases ... 26

3.3.5 Acceptance tests ... 27

3.4 Design ... 27

3.4.1 Software Architecture Documentation ... 28

3.4.2 Data models ... 30

3.4.3 Process modelling diagrams ... 30

3.4.4 Selecting the architecture ... 32

3.4.5 Selecting frameworks and libraries ... 33

3.4.6 User Experience (UX) ... 34

3.4.7 Mockups ... 35

3.5 Implementation ... 35

3.5.1 Version Control....35
3.5.2 Branching strategy & Branch permissions....36
3.5.3 Documentation....38
3.5.4 Unit testing....39
3.5.5 AI Tools....41
3.6 Testing....41
3.6.1 Integration testing....42
3.6.2 Regression tests....43
3.6.3 Continuous Integration....44
3.6.4 Definition of Done....44
3.6.5 Acceptance tests....44
3.6.6 Different areas of testing....46
3.7 Deployment....46
3.8 Maintenance....48
4 Discussion....49
References....52

Figures

Figure 1 - Implementation of DSR in the research....6
Figure 2 - Kanban board....16
Figure 3 - Software Architecture Diagram, made with draw.io....29
Figure 4 - Swimlane diagram created by ChatUML....32
Figure 5 - Example of a branching strategy....37

Glossary

AI	Artificial Intelligence
API	Application Programming Interface
CI	Configuration Item
CI/CD	Continuous Integration / Continuous Development
CM	Configuration Management
CMDB	Configuration Management Database
DSR	Design Science Research
SDLC	Software Development Life Cycle
UCD	User Centric Design
UX	User Experience
QA	Quality Assurance

1 Introduction

1.1 Background And Motivation

The author is working as a lead software developer and a project engineer in a software company. The company has already identified that the more the number of developers and the size of the projects increase, the more the need for standardizing the ways of work increases. The different workflows and tools need standardization. The standardization would serve both existing employees and new employees.

The company has already identified the value of standardizing workflows, which can of course be also seen as a paradigm these days. This report will address it more in the chapter 2.2 Defining and Improving Process.

1.2 Research Method

In this research Design Science Research (DSR) is used. It is a paradigm focused on problem solving via creation of artifacts that can be represented by constructs, models, methods, and instantiations (vom Brocke et al., 2020). Design Science Research is a suitable method, since it is focused on problem solving from both academic and organizational standpoint and the solution generated should be applicable in a more general sense (Dresch et al., 2015). The aim of DSR to generate knowledge on how things could and should be done can be used to work on enhancing tools or processes of an organization, which is the aim of the research (vom Brocke et al., 2020).

The methods used for gathering and analysis data for the research are content analysis combining data from previous projects and reading literature about best practices regarding different activities of software projects. The main method of evaluating the gathered data is pilot testing the artifact while parts of it are produced, so that the design can be confirmed useful and rational in practice (Stewart, 2023). Usability testing should also be conducted on appropriate stages to observe the clarity, readability, and overall user friendliness of the artifact in a more systematic way (Affairs, 2013).

1.3 Research Process

DSR framework usually introduces five steps for the research process. These steps are identifying the problem, defining its requirements and objectives, designing and developing the artifact, demonstrating the artifact and then evaluating the artifact (Johannesson, 2014). Some models can have more steps, like for example the sixth step of communicating which outlines informing the stakeholders about the findings (vom Brocke et al., 2020).

The process of DSR is an iterative one. Depending on the process model, the iteration happens from evaluation back to designing when needed or evaluating each step and returning when needed (vom Brocke et al., 2020). Thus, it seems wise not to try to build a too comprehensive and overwhelming guide at once, but to iterate and increment it to allow the team to internalize it more easily.

The plan is to adopt the DSR into the research as shown in Figure 1 - Implementation of DSR in the research First the problem is investigated and defined in order to justify the research. Then, the first set of requirements and objectives will be defined, i.e. we shall define phases and activities of the software project that need improving, documenting or otherwise researching. This is expected to produce an incomplete list of requirements, some of them possibly still highlevel and some refined to be selected as the topics to start the research with. After this we start the iteration of process, and we start to design and develop the first version of the artefact, which should aims to address the first set of requirements, i.e. topics for the handbook by identifying the underlying process, documenting it and comparing it to the best practices. Next, the version of the handbook will be demonstrated to the pilot team, which will test it in practice. After the test period the artifact's usefulness and the ability to fulfill the requirements will be evaluated. The process will be then iterated. In this process model the new iteration will start from defining the requirements again, as it is very likely that the pilot team will a better understanding of the areas of potential improvement as the research and the team's projects progress. After the final iteration the handbook will be demonstrated to the other stakeholders, mainly the management of the organization and the resulting artefact will be evaluated from the organization's point of view.

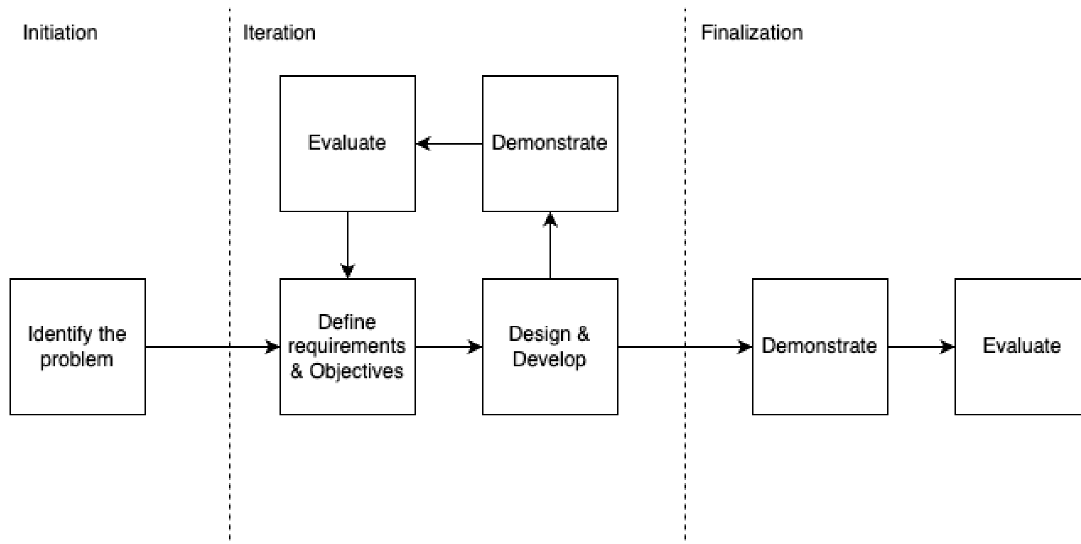

Figure 1 - Implementation of DSR in the research

1.4 Research problem and goal

The first step in DSR defines the research problem and describes the value for its solution (vom Brocke et al., 2020). The problem should be relevant, and the researcher should be able to demonstrate that there are no current means of solving the problem and in ideal case it should be a general one in a way that the solution will be helpful in a broader sense (Dresch et al., 2015).

The problem is to maintain efficient software development project in a software company, where variety can lead to defects, waste and other issues by for example the project team not properly documenting project requirements or not testing systematically.

There are many sources about Software Development Life Cycle (SLDC) in general and best practices can be found, but they often either compare the model's suitability for diffferent projects or talk describe the SDLC and their phases on a general level. Also many of the available sources do not encourage to closer inspect whether the general ideas and best practices apply to all organizations, teams and projects, but present them as the single right way to operate. Thus the problem can be summarised in that the company needs to find and document the best software project workflow.

The comissioner has identified the this work relevant. It should be applicaple in more companies, if done with adapting in mind. Some chapters should work in some or even most companies as well but the workflow is intended to work as a template, to which the company can identify and add their own processes.

Following Design Science Research methodology, the next step is to define the objectives and goals for the solution. In this case, the objective is to compile the project working knowledge into a single source and to create a standardizing guide that helps a software company to manage different areas of software development: requirement management, the actual developing, version control, quality assurance etc. The idea behind these objectives is to reduce lead-times and variations and improve productivity and quality of a software team and their software projects and other processes surrounding it, like training new employees. The actual outcome is a project handbook that is created for the company that is the commissioner of the thesis and a pilot for the handbook. The ideal outcome would be that the handbook could be applied to other companies needs too, probably with some adjustments.

For the DSR process the aim is to treat the handbook as the main artifact, but also to divide it's chapters as subartifacts. This way it would be comprehensible to treat each topic of software devopment as it's own problem and process.

Two different approaches and models for the structure was compared. One would be role-based approach, where the chapters of the handbook would be divided based on the different roles of the team. The other was based on the SDLC. The latter was chosen, as the members of the participating team work in different roles depending on the project and other factors. Also, because SDLC represents a widely used paradigm in software engineering, it should be applicable in more general sense too.

2 Knowledge base

2.1 Software Development Life Cycle

Software Development Life Cycle (SDLC) is the software process and defines the structure and the phases of the development of the software product. (Jain, 2011). Since many projects nowadays

follow some kind of model - or at least implement most of its phases - it seems logical to base the structure of the artifact onto the SDLC phases.

There are slight variations also in the ways the of software development lifecycle is presented, but most of them describe the following phases, included in some point of the development process:

1. Planning and gathering requirements.
2. Analyzing the requirements and designing a solution to fulfill those requirements.
3. Implementing the solution based on the design by coding and building the software.
4. Testing the software to ensure it meets the requirements and has no defects.
5. Deploying the software by building and installing a release version.
6. Performing maintenance by for example monitoring and addressing performance and defects.

Since the aim is to create the handbook for the software process, these phases probably should be addressed in the chapters

Of course, there is difference in the models. While this is true especially regarding the phases, there are significant differences in what is included in different phases and how they are carried out. For example, in the V-model the planning of the tests is significantly different than in other models (Jain, 2011). Teams implementing Scrum have a different workflow with increments and focus on customer collaboration (Stober & Hansmann, 2010). The goal, however, is to find development steps that are mutual to the different models and generate a template that can still be applied with slight modifications. However, even though the workflows are different, it is safe to say that many if not most of the activities are still included in all of the models. In every model there are some kinds of activities for example regarding planning, testing and development. Of course, iterative and incremental models with agile workflows leave a lot more room for example for changes in requirements, but they both may have to address requirement management still at some level.

One topic that varies in practice is documentation. Like Rüping (2003) concludes from Agile Manifesto, document should be light, accessible and useful instead of comprehensive. The amount of needed documentation of course varies between projects and customers. Network diagrams hardly apply to every project and while formal user manuals are more common, there are software that don’t necessarily need it, like system services or simple software with self-explanatory UI. Thus, during the creation of handbook it should me meaningful to look at documentation that

is either generally common or useful to the commissioner of the research. The organization deploying the template should review the documentation generally produced in their project workflows.

Though there are those, who claim that following a phased SDLC is against the idea of agile, there are those who define the process flow of the agile method as an iterative workflow that actually includes the same workflows. Cline (2015) for example describes the process as iterations of defining and analyzing requirements, designing, coding and testing, producing a deployable release to be delivered.

In summary, even considering the differences between SLDC models, and organization's workflows, the topics should be relevant to software development projects in general. When implementing them however, adaptation is surely required.

2.2 Defining and Improving Process

The theory behind the research is that documenting the standard project workflow and implementing best practices will improve the team's efficiency and project success. The theory is strongly based on the paradigm that documenting the process is the basis of improving it. The improvement comes from standardizing the workflow, which in turn saves time in figuring out the next steps and ensures that steps are not forgotten. Also, it helps the team to internalize the working methods and tools and also to train them to the new employees. These ideas are also supported by Fantina (2005), saying that the key to achieve these benefits is the documentation of the process. Liu et al. (2008) also found in their research that standardizing the process – as in procedures and tools, for example - improves the project performance.

One methodology worth noting is Lean Six Sigma. It would be out of scope to go through its principles exhaustively, but some points should be considered. As Carreira and Trudell (2006) have written, things such as inconsistent process, waiting and adding nonvalue can lead to a significant amount of waste. While these can be addressed with proper process standardization, it also gives us the opportunity to place and monitor metrics to measure the process. One additional viewpoint regarding Lean Six Sigma is that documentation should not be the purpose. Unnecessary content is waste and weakens the handbooks readability and efficiency.

While there are of course many existing process frameworks, there is a possibility that the company cannot just implement them as such. Small companies, in particular, have found process models like CMMI® and ISO too complicated and thus perceiving the invested resources are not worth the benefit (Larrucea et al.,2016; Oktaba et al., 2020). Thus, adjusting the project process standardization to the company needs and resources seems crucial in order for it to succeed. Fantina (2005) notes though, that the CMM® and CMMI® can be used as basis of the process improvements, even if the organization is not planning formal assessment.

Fantina also noted in 2005 that many process improvement manuals are more theoretical than practical. This still seems to apply a lot today. Guides about SDLC and the software process are often high level, and the more detailed ones are often single best practices not attached to a process.

As Fantina says, the process improvement has to start from initial assessment so the areas in need of improvement can be recognized. Fantina also says, that process improvement is free, but the initial expense will pay for the investment. As both parts of the statement are true, it eventually means that a feasible assessment and plan for the development has to be made also to convince the stakeholders to put their time and money in it. This means that a convincing plan is needed, and it has to show understanding to the target organization's processes and process improvement. Of course, all this applies to the task at hand, since the process cannot be really improved without understanding it first.

One heed of warning many authors like Fantina warn about that there probably will be resistance. Some kind of resistance has been experienced in practically every process improvement project worked on before this research. Of course, these are usually something to overcome, but also something to consider beforehand, so they can be dealt with. Like the question about the money before, careful planning and presentation should ideally convince the stakeholders, that the investment is worth it. Another phase where resistance often surfaces, is during the initial process evaluation. Some managers and team members can take process evaluating personally and feel reluctant to admit areas of improvement, as they fear it can be seen as incompetence. This too can be overcome, but often needs a careful and thoughtful approach.

2.3 Configuration Management

Configuration Management (CM) is a wide topic that relates to software process improvement. Leon (2015) describes it as the method of bringing control to the software development process. It links to software development throughout its life cycle (Fantina, 2005). It provides integrity, traceability and maintainability in the project.

SCM identifies unique versions of Configuration Items, like the project plan, design documents or the software source code. Configuration Control ensures that each change to a Configuration Item (CI) is authorized and documented. Configuration Status Accounting is the process of recording and tracking the status of each CI. (Horch, 2003).

Due the relative complexity and different options in methods and tools used, it seems best to get familiar with before implementing it in the organization, if the organization does not have an existing process for it. For example, there are variety of CM tools that differ in features and complexity. Then again Fantina gives an example of a basic example of a simple and minimal CM procedure that is based on a single folder on a shared drive accessible to all team members. After the organization’s configuration management high-level processes and tools have been defined, it is easier to focus on implementing it on the SDLC process.

Benefits of CM are for example ensuring that the team is aware of changes into requirements or design and know to take it into consideration when developing or testing the software (Fantina, 2015). Also, it helps keeping track of different versions of software both in the sense of subsequent (for example feature increments) and parallel releases or configurations (different features for different customers) (Horch, 2003).

CM processes may be informal or overlooked, because managing CM can seem to include a lot of work. It still pays the effort, since it saves time by avoiding troublesome problems (Fantina, 2005; Leon, 2015). These errors can include missing source code, trouble tracking why a change was made, or designing the system based on outdated requirements.

Because there are different CM activities throughout the SDLC, different activities will be addressed in the chapter of the relevant phase. Since it is already established, that CM tools and

workflows vary between organization by culture, size and project types, it is highly likely that the methods implemented throughout the project will need adjustments to fit into the procedures of another teams. The overall importance behind the explored activities should however be generally applicable.

2.4 Artificial Intelligence

Artificial Intelligence (AI) has developed a lot lately and it has brought many tools for different phases of SDLC. Artificial intelligence is already used in many fields in technology and the Global AI market is expected to grow 37,3% from 2023 to 2030 (Grand View Research). Gartner has also estimated that in 2027 70% of professional developers will use AI-powered coding tools, when now it is only less than 10% (Panetta, 2023). While AI is not supposed to replace software developers, it is still said to chance a lot how software developers work.

Based on conversations with software company representatives and discussion in professional networks, there has been discussion in many companies about the benefits and risks in using AI. As AI is powerful, it is also certain it can pose risks if adopted into company toolset carelessly and without research and strategy. However, experts warn companies that if they don't soon adapt to the new AI technology, they will soon fall behind.

Because AI can improve the process and efficiency of the software project, some tools will be discussed within this research. One should keep in mind, however, that they should not implemented carelessly without a strategy reviewed and approved by the organization.

While AI brings many benefits, it has its risks and tradeoffs too. These aspects need to be reviewed carefully before implementing AI in the workflow. Some of these risks are confidentiality, ethicality and credibility. Some of these risks are discussed in this chapter, while the potential benefits are discussed further in their respective chapters of the research.

Large AI language models like ChatGPT and Google Bard use a large amount of existing data to build its knowledge on and they often use user prompts, conversations and interactions to improve that knowledge. This is considered to have its trade-offs, like using the AI might expose your

critical intellectual property and other private data to be used in developing the AI or the vendor's other products. (Panetta, 2023)

It has been claimed that some companies developing AI have been using copyrighted material without asking or crediting Authors. (Reisner, 2023). It has also been claimed that AI training on copyrighted works harms the market and value for those copyrighted works. While there are authors who offer licenses that allow their works to be used in training AI, there are still companies that have used their material without obtaining them. (Coffman, 2023)

AI bias, also known as machine learning bias, refers to AI results that are prejudiced for or against certain groups. Potential problems of bias in AI include discrimination (FullStackAcademy, 2023). Some examples of this are bias in mortgage algorithms (Counts, 2018) and bias in health algorithms (Obermeyer et al, 2019). Also, OpenAI themselves have warned in their paper that GitHub Copilot is susceptible to bias (Chen, M. & et al. 2021). When inspecting this issue deeper, it is obvious that the bias problem affects more others than the rest. Depending on the software being produced and for example the level of usage of code synthesis, the risk can vary greatly. In the previous examples models affect bias, bias affects results, and the results affect people's lives. If we then look at an example, where the code generation is used to create basic level functional code, like mathematical algorithms or protocol code or in general when the user has the output already in mind but need AI to write it, the results are different. If the developer uses the program synthesizer for example to create a TCP-listener that will handle the bytes based on given symbols, the risk of bias seems much smaller. The users of AI should be at least aware of the possible bias in the code and how to spot it.

AI's capabilities have been widely recognized as it can often generate content quite swiftly and accurately. It does not always get everything precisely right, as have also been found out in the examples of the possible uses of AI. The AI has been reported of doing even big mistakes that affect people's lives and have led to lawsuits (Belanger, 2023). This reminds that when using AI, especially for business reasons, it is better to check the information that you get from it.

OpenAI has some time ago already brought up the problem of adversarial examples. They have provided examples how AI can be tricked with the training data and made to behave in unwanted

ways. Thus, they have stated that adversarial attacks have the potential to be dangerous and that the attackers could intend to do harm. (Goodfellow et al. 2017)

One motive for data poisoning is also authors protecting their copyrights on the data. As many authors feel their rights have been violated, some have seen poisoning data that is used for training the AI model is a way to protect them. One of these examples is Nightshade, that was developed by University of Chicago (Franzen, 2023). Also, Sun et al. reported in their paper in a Web Conference (2022) how they designed and implemented a prototype called CoProtector, that had the goal of arm repositories with data poisoning techniques and to defend against data exploiting of CoPilot-like deep learning models.

Research thus shows that there are multiple motives and means to trick AI into learning false information. This goes to emphasize even more that the user needs to check, confirm and understand the information that they get from the AI and evaluate the possible risks this could pose to their software and its functions.

From experience and previous research, the base requirement of using the AI tools is that the user must know what they need and what they expect to get and are always capable of understanding and validating the results. The confidentiality aspect must be evaluated both based on the tools and the projects in question.

2.5 Tools and software

There is a vast range of tools to make the different activities in software development easier. Those tools vary between teams and organizations and while some tools are introduced in the research, the organization should choose and adopt them based on their needs, size, integration needs and other factors. By dictating the approved applications inside the organization, the security of the software in question can be reviewed and compared to the organization’s guidelines. It also helps the organization to ensure their team members have adequate skills and instruction to use the tools in question. In summary, the organization should choose the tools carefully, document them and link or create the instructions for their usage.

3 The Handbook

To find a place to start, the project was drafted on a mind map, the project here of course being the software development project handbook. The idea was first to clarify the goal of the project, figure out the audience/stakeholders of the project, so I could start by introducing the why and to who we are doing this project for. Another purpose of the mind map is to illustrate the workflow of the projects and to sketch out the initial form of the handbook. Since we are documenting – and hopefully improving – the process, we first have to know and document the process.

The project was presented in a weekly meeting of the main participating team. They were introduced with the goal, which would be documenting, standardizing and improving the workflow in iterative manner, testing and getting feedback before diving too deep. The Software Development Lifecycle (SDLC) and its phases as an option for the chapters were discussed. A theory that had surfaced while preparing for the project was reinforced: one should be careful when talking about the SDLC models and methodologies with software developers. In many conversations, the team members were found to have strong, contrasting opinions on things like how Agile they are, how rigid the sequential models are and whether testing should even be a separate phase from the implementation phase. Their focus was thus steered by breaking apart the typical shape of the SDLC model for now, dividing it to topics separate from each other. This seemed to be helpful for the team to concentrate on thinking what subtopics and areas of our workflow they include, and how they could benefit from documentation.

3.1 Project workflow

First going into the individual project phases, the team should establish the common procedure for transitioning from one to another. There are also many tools and software to maintain and monitor the project life cycle and activities. Establishing a clear view on what is in progress has many benefits. It is easier for everyone to work on a project if all participants know who is doing what and what in what order. There are many different ways for this, but one popular tool for this is using Kanban. Many sources already tell the history of Kanban, so it will be considered out of the scope in this research. It is not necessarily the right tool for every team, but it is nowadays a popular tool for many teams and gives a good view on what aspects we could standardize in the project

workflow. Different teams can have different columns in a kanban board, but Figure 2 - Kanban board shows one working example.

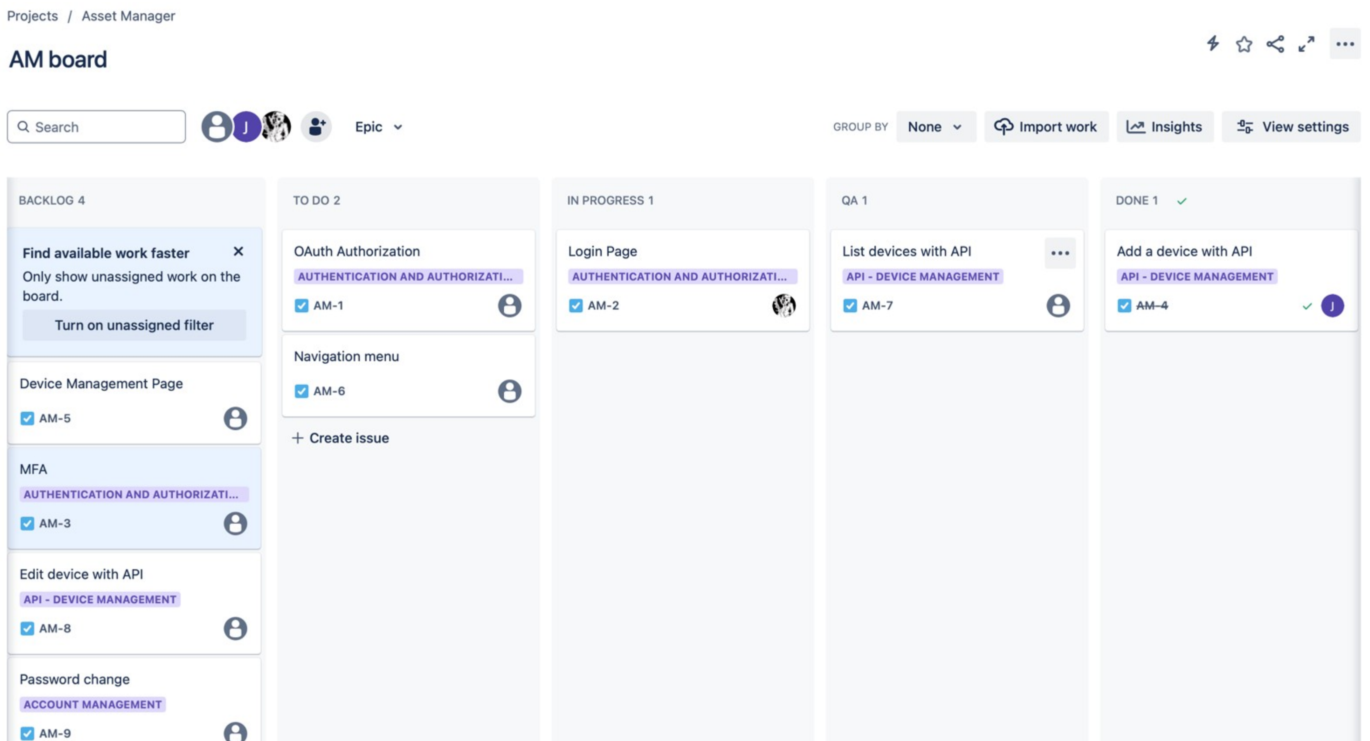

Figure 2 - Kanban board

There are also different nuances on how to work with it, but the following is an example from real life projects. There are also many tools for automatizing, visualizing and integrating the workflow, such as Atlassian's Jira, that is used by the team in this research. Such tool for example allows the team to link the work items for example with the requirements, source code and release versions.

Backlog

The *backlog* is often not visible in the Kanban board but is a separate view. It is the list of the tasks that will be (likely) worked on in the future but are not active right now. By keeping the list separate, we still have a clear picture of how much work we have left, and we can even prioritize it. Keeping it out of sight while the team is not working with the tasks however helps to maintain focus on the visible actions that are currently active.

Selected for development

From time to time, tasks are moved from backlog to *Selected for development* or *to-do* column, which is a list of active tasks to be done in a given timeframe. The tasks are picked from backlog regularly, like in the weekly meeting. The number of tasks should reflect a reasonable workload for until the next meeting. For example, in agile development like scrum it would be the workload the team has estimated they can finish during a sprint. Depending on the team workflow, the tasks can be assigned to the persons who will be working them, or the team can pick up tasks from the list when they finish doing their previous work. With the pilot team too, filtering the list of work like in smaller work packets has been found to be useful for maintainability but also the team has felt less overwhelmed about the workload. Also, when the responsible for the task order and progress creates the list on a regular frequency like on a weekly basis, they both save the effort of micromanaging tasks and work of developers. The board also gives a clear picture for the near future of what tasks are going to be finished.

In progress

When a developer starts working with a task, they move their task from to the In Progress column and assign the task to themselves, so the team can see that the tasks is being worked on and who is currently responsible for it. When using Jira and Bitbucket, the development or bugfix branch can be created for the current task simultaneously and semi automatically. By using the issue id from Jira in the branch name or commit messages, all the code for implementation will be linked to the issue and the code can be accessed, monitored and traced from the project management tool.

Many project management tools like Jira allow the team to enable Work in Progress (WIP) – limits to the kanban board. This simply means that there is a restriction on how many tasks are simultaneously allowed to be In Progress state. This kind of restricting is encouraged by Lean thinking and also found useful with the research pilot team and previous projects. Too many tasks simultaneously active has proven to increase lead times and multitasking complicates both focusing on the task and estimating its completion.

Quality Assurance (QA)

When the developer has finished the implementation and run the unit tests, they move their task to the QA column and create a pull request for the branch in source control. This is a sign for the team that the task is ready for testing. The tester reviews the pull requests and goes through standard procedures that are more discussed in the testing chapter.

Done

The *done* column indicates that the items inside it are fully implemented and tested and ready for release. Presuming that the tests passed in the previous phase, the tester moves the issue to the *done* column. If not, the task can be moved back to selected for development. If there was something lacking regarding the issue, but it still passed the criteria for release, the task will be moved to done, but a new linked issue will be created to address anything that still needed attention. Important thing for the team to consider here is the Definition of Done (DoD), that is also discussed in testing chapter.

3.2 Planning

Many sources do not include the planning phase. They often start from the requirements gathering and analysis and perhaps explicitly include it there. In many cases the SDLC starts with the planning phase, and it often includes high-level planning, mapping out business goals and challenges regarding features and evaluating feasibility (Tozzi, 2024). Now of course, in many cases these activities are already gone through during the presales (Suvorova, 2024). Keeping this in mind, the organization should define their software processes and evaluate, whether to document the planning phase in their guidelines. Still, as long as these activities exist in the process and belong to the responsibilities of the software team, they should be taken into consideration at some phase of the process.

Cline (2015) also states that many planning actions actually need to be done in agile development too, though the responsibility might belong to someone outside the developing team in some organizations. With the agile iteration approach though, not all planning needs to be iterated, so the planning can be divided into initial planning and the planning phase of the iteration.

The commissioner company's workflow was inspected and many such activities stood out that could be found from general recommendations for planning phase. Some of these would be defining the project goals, schedule and the high-level plan.

In the iterative and incremental projects like those using Agile methodologies, some planning may be done only once at the start of the project and some planning may reoccur through iterations. Thus, in the handbook, we should identify two planning phases: the initial planning and the iteration planning. This division should fit for both the iterative and not iterative process, as in this sense the latter can be thought to have only one iteration.

3.2.1 Schedule

Though not always, in many projects the schedule has already been defined at some level in pre-sales. It may well be that the customer wants to have the system or at least a part of it delivered at some certain date and this has already been agreed on as part of the purchase. This often applies in industrial environment, where the software is needed to drive the business. Still, while these kinds of existing agreements can apply some restraints, in the planning phase the schedules nevertheless have to be planned in more detail.

Again, different projects and project types have different workflows and needs, but in general it planning ahead is beneficial. In more traditional projects the project manager has to schedule the work packages that they obviously fit into the timeframe, but also get done in sync and tasks won't have to wait for others to finish. Planning like this does not really fit into the definition of agile development, since the deliverable and the priorities are shaped along the way. Still, there are things to plan, like release schedule or the release plan (Cline, 2015). Even though the backlog and the priority of features will change between iterations, the agile frameworks like scrum for example aim to have a deliverable release after every sprint of fixed length. In the projects reviewed for this research it has been found very helpful to establish the schedule describing the start and end dates of these iterations. This adds visibility to the project for the stakeholders developing team, the management and the customer. There have also been cases when the agile project needs to sync on some level with the more sequential projects. For example, another vendor or team with a more fixed schedule could be implementing a system that needs some level of functionality for the system integration testing on a certain date. While this can be viewed mainly

as a driver to prioritize the backlog, time frames like this should be synced with the project roadmap to avoid missing them and delaying projects dependent on the one at hand.

In the agile iteration approach the planning phase begins with a iteration planning meeting, during which the team can estimate the size of each use case and thus how many of them can be completed during the iteration (Cline, 2015).

3.2.2 Risk Assesment

Risk assessment can be a difficult subject, if the organization has not yet implemented it as a default strategy. From experience the organization can consider it as a superfluous activity will only add the project workload rather than the opposite. The results from the projects practicing it have shown that it can actually save the project from delays or worse consequences, and it actually won't take much time, especially after the team has adapted to the process. Also, we have to remember that many projects surviving seemingly without it actually just don't use a formal process for it, but rather there is someone in the project team that is just experienced in doing it. Again, by standardizing it we can both make sure the process is not dependent on selected individuals and also that we can learn from previous projects' risks, both predicted and realized.

The risks should be first identified and analyzed by their probability and impact, meaning how likely they will occur and how they will impact the project. Comparing to these numbers the team should decide on the response strategy for each risk. The strategy can be avoiding the risk, transferring it, mitigating it or just accepting it.

One type of relevant risk in certain types of projects is supplying and procuring hardware as part of the system delivery. If the time between planning and deployment of some hardware is long, there is a risk that the hardware is already older generation, and the support and warranties have worn while waiting on the shelf to be installed. Then again, there is a risk of delays in the shipment of the hardware, especially for example if there is a component shortage. The ordered hardware can have defects, which can be handled by reserving time to order replacements or having the replacement in reserve. These are just some examples from experience, and many more exist. Good thing about many of these risks, at many times they apply to other projects as well, if not all, then many. This means that risk assessment does not have to be built from scratch every time,

since we can reuse a lot of earlier project's risks, just reviewing their applicability, probability and impact again.

Another important type of risk that should be reviewed is information security risks. Doing this ensures that the project has been delivered from planning to deployment (and maintenance) with cyber security in mind, for example designing the network segmentations, authentication methods and protocols according to sufficient security practices. Also, this helps the project to comply with standards like ISO27001, as they require the information security risk management into the project and the process of doing so documented.

Reviewing the risks of previous projects and implementing risk management to new ones, one observation has been made that speaks for the ease and efficiency of risk management. Within the organization the projects tend to have many similarities, which causes the same risks turn out to be applicable in the more future projects. This in turn allows the team to just filter out and the applicable risks for the new project, reducing the workload for the identification of the risks, which has been found to be much more time consuming than estimating the probability and impact.

3.2.3 Selecting the SDLC Model

When a project starts, it is necessary to plan ahead how to proceed with it and what model or framework to use in the project. There are many sources that say one is better than the other and even those that say that say that only agile framework is the right one and that waterfall model is outdated and should not be used in the modern software development.

In the reality though, not all projects can be done for example with all of the agile best practices. Some big companies and organizations have to follow the public tendering rules, and in those cases, you cannot be too vague about the cost, the time schedule or the provided features. Of course there are many industrial customers too, that need to have the specified functionalities in given time to get the project adjusted with other suppliers and keep the business running. Sure, the pitfalls of waterfall model apply here and there is a risk of the cost or time exceeding from the estimate, but it needs to be taken in account in planning. Also, some would say that the V-model is anyway much better for these projects, as the testing and quality assurance is taken into accord

much better. Still, like they say, waterfall is ideal for simple projects where requirements are understood clearly, and I think this has applied for example in projects, where an existing software is implemented with minimal to none changes in features. These kinds of projects indeed are quite straightforward to plan and implement.

In the same way, many projects would be too rigid in the waterfall method and really benefit from the agile approach. For example, I have seen iterative and incremental SDLC models with agile principles included work very well even in the industrial environment in certain type of projects. For example, development projects where the existing system is enhanced, the customer can have a list of features they would like to have added, but the features themselves or the business continuity is not dependent on any particular one of them. There are also projects where the customer process has to follow certain standards (like JIG standards in aviation) and has a lot of integrations. In theory, they can all well-defined and easy to follow through, but many times there are details that need to be communicated and confirmed with the specialists on those fields. Then again, they always seem to have details that have been understood differently in either side. Also, the best engineers of the projects can try to include this all into the Procurement Specification, but I have still not seen a project, where it was not necessary to communicate with the stakeholders, especially users to adjust these requirements to actually make the delivered system usable in the context. Many times, there are details in the workflow that are just hard to vision without seeing the users go through their workflow and listen for their input about what actually works or not.

One note about selecting the model or framework however is that the project manager and the team should not focus too much on the generalized description of the given model. Many sources claim that waterfall is too rigid and does not pay enough attention to the users, v-model is too expensive to be implemented and that agile methods are not planning or documenting. Each methods' attributes however can overlap greatly.

One thing to note based on experience from several projects though, that however sequential the model is or not, the communication and interaction with the customer always pays off anyway. In some projects the customer can have their own division of engineers to come up with a procurement specification and if you follow it through, at the time of delivery the actual users will tell you that some the features don't serve their purpose. Of course, in projects like this the developers

cannot usually just act on customers wishes as flexibly as in more agile environment, but the potentially rigid change request process should still be a lot easier before the feature has implemented.

3.2.4 Configuration Management plan

While implementing configuration management in the initial phase of the project can have many advantages, it can still fail greatly if the CM activities are not properly adopted and familiarized with. Leon (2015) mentions, that even if the SCM is implemented incrementally, it is still always better to have a configuration management plan.

3.3 Gathering and Analyzing Requirements

When reading about software development project related articles, books and other sources, requirements can be found very often surfacing as an important topic. Many sources say that handling requirements is one of the biggest reasons of project defects.

3.3.1 Requirement Management

First thing to document probably should be where the requirements should be documented. Different requirement management tools and workflows exist, but they should be accessible and traceable to the related requirements and test cases later implemented. Requirement management procedures should also take possible changes and new requirements into consideration.

Requirement repository should also be versioned, so it is possible to link the requirement into a specific version it applies to. It is one of the essential configuration items usually mentioned in configuration management. They should also be documented in a concise, because they should be reviewed from time to time during the project: the team needs to access them when new requirements are added or existing ones are changed, and when designing, implementing and testing the product.

It is very common that in software projects there will eventually be new requirements or changes to existing ones. Whether through formal change requests in the sequential models or new iteration in more agile workflows, the effect of the new requirement on the present ones should be

evaluated, so they won't contradict each other. Also, the change should be documented somewhere in a systematic version. There have been both sequential and agile iterative projects where some of the stakeholders are not aware that some requirements have changed and from their point of view see a defect in the system. While it may be that someone from the team remembers that the functionality was changed for a reason, there has been many cases where no one present remembers why and by who. Now, in worst cases, without justification the functionality has been restored to match the original requirement – only to be changed again when the reason surfaces again.

3.3.2 Writing Requirements

There are books and articles how to write good requirements, but some essentials have to be documented. A good requirement serves multiple stakeholders in the project. In many software projects the customer has asked for a certain feature. They have a problem in mind, and they have already come up with a picture of a solution in their mind. The problem is that they really are not that technologically oriented, so the solution they have come up with is hardly optimal. Then again there are also developers, who interview the user for their needs and come up with a solution. This time the problem can be that the developer does not really know the business field or the workflows that the certain user group work with. This can result to a feature that is far too complex, lacking in features or just unusable. Based on experience, best results have been achieved when the developer team works in collaboration with the users to ensure the need is understood and the solution actually serves the purpose.

A good format that is often recommended in books and articles is the *x needs y in order to z* – format, which results from listing the *what*, *who*, and *why*. *What* of course defines the initial feature or functionality we need to develop. The *who* can give us a lot of context for the requirement. The who can be an administrator or operator which can affect the user permissions for the feature. In some cases, like for example a truck loading system it also very likely affects the user environment, as the administrator might access their features using PC and the operator could have a mobile device or an HMI to access theirs. The why should tell us the business value of the feature, so it can be prioritized based on whether it actually adds value or is just a nice-to-have. Also, the why can give actually give us options on how to implement the feature. One simplified example from previous projects is when the user asked for the UI to show a number and a button to increment

it. When asked, the reason was to record the number of events handled by the operators. Knowing why, it was obvious that a much more optimal feature for the user was to make the system track the number automatically, as all events where already handled through the application. Another real-life example is when the customer, who was purchasing a tailored version of an existing software, specifically asked for a visual graph to describe production configuration. The request was based on software they had been previously using and a feature that they were using frequently. The feature was implemented and when the software was deployed, the users noticed that the software had actually existing features that covered the required functionality even better.

When writing a requirement, one additional question to add to the list might be when/where if applicable. There might be some conditions for the specific features that don’t apply for the whole software in terms of performance requirements. There might be a specific function that needs to work offline or needs to be operated in specific conditions. Even if these conditions are listed in the non-functional requirements, they should be clearly linked to the functional requirements, if they don’t apply system wide.

The requirement should of course also be clear, unambiguous and testable. If the customer and the developer have a different interpretation, there is an unavoidable risk that the developer does a great job in implementing the feature as far as they understood it – and still creating something far from what the user needed. The same problem can surface when the developer passes the implementation for testing. The tests can erroneously either fail or pass here, if the requirement is open for interpretation. One preventive measure here is to define and review the passing criteria while defining the requirement.

While individual requirements and user stories should be brief and concise, sometimes some background for them is required, for the team to actually understand what they are developing. Like discussed before, this can especially apply when the field of business is not familiar to the software. In these cases, creating use case diagrams or user flow diagrams can give the developer a better picture than just listing features to be added.

3.3.3 Traceability

The requirements should be traceable, meaning it should be possible to connect the requirement both to its implementation and tests. The traceability helps in both ways, as the tests and implementation can be reviewed reliably only if we need in what requirements they are based on. Considering the feature complete on the other hand can only be verified by finding the implementation and ensuring it has passed its tests.

One example case seen in real life was when a new version of a software was not behaving expectedly in a new environment. A developer was assigned to debug it, and they found that in a commit between the current and the new version, some new code was added to the middleware. No justification or comment was found for the change, so it would have been seemingly easy to just revert that specific change. After further inspection it was found though, that the change was an essential security update, that was not just self-explanatory. Here, it would have been helpful if the implementation would have been traceable to back to the requirement that was the reason for it.

Another example case is a case when in a certain project more functionalities had to be added to an existing feature. The feature itself was a seemingly simple form, but there were different actions to be executed based on the user's input. From the tester's point of view, the requirements regarding the feature, or the implementation's expected behavior were not clearly defined. Thus, black-box testing was practically impossible, and the tester had to infer the different outcomes from the source code. Again, if the at least the requirements would have been traceable from the implemented feature, the testing would have been a lot less time consuming and cumbersome. Better yet, the tests that had been previously executed could have been also documented and linked to the requirement and implementation.

3.3.4 Use Cases

Functional requirements can be presented with use case diagrams, that describe the actor and the use case. The actor can represent a specific user group while the use case tells what how the user might interact the software. This helps the team to capture and communicate the requirements of the software and also serves useful information for the designers and developers.

There are different styles and forms of using use cases from more minimal description to a more detailed one, where in addition to the actor and the summary of their interaction, also for example preconditions, triggers and detailed flow with alternatives are described. The detail of the use case should be chosen based should be partly based on the complexity of the use case, since it should provide value by communication with the stakeholders and usage in later phases, rather than being documented for the sake of documenting.

Use cases can be also presented as use case diagrams to add clarity for communication with non-technical stakeholders. UML has proven to be a clear, but professional enough notation for this kind of communication and has been easy to learn and use for the team members.

3.3.5 Acceptance tests

In models like the V-model, planning the acceptance tests is described as an action parallel to Requirement analysis. This has proven to be effective for multiple reasons. One is that by doing this, the project team can go through the tests to better ensure that the requirements are fully understood. If both parties' views differ on whether the acceptance criteria is sufficient or has some errors, the requirements can be still reviewed and changed before anything has been implemented. Other useful aspect is that when the team is designing and implementing the product, a clear and comprehensive acceptance test plan can also serve as a guide for both desired functionality and the scope. Acceptance tests are discussed more in the chapter of testing phase.

3.4 Design

Different projects include different design documents. Still, the form of the certain documents should be standardized. One example for this that has been seen problematic in different organizations is network diagrams. Different designers prefer different styles, for example different symbols for servers or different lines to describe different connection types. Ideally this should be standardized, so everyone in the organization is able to familiarize themselves with the diagram by looking at it. This is not always possible and even if it was, there is a high probability that the customer or other collaborating parties in the project use different way of diagramming. Thus, the diagrams should essentially include a legend of symbols used. This minimizes the ambiguity and amount of guessing, which in turn results to less errors or at least less time spent on interpreting

the diagrams. In case where a standard notation can be adopted, there should be a guide for the user to use it effectively.

3.4.1 Software Architecture Documentation

Documenting the software architecture helps developers in designing and developing the components of the system while taking the whole system into account. It also helps to clarify the system dependencies and interfaces. At the same time architecture gives constraints to the design, making sure that the goals are met and records the decisions made related to the design. Of course, as the size and complexity of the projects can vary, not all projects necessarily benefit from the architecture documentation same way as others. Thus, architecture documentation should be created based on the usefulness, not just for the sake of doing it. Still in many projects like some Agile projects it is still useful to do some of the documentation, but to leave out some of it based on the cost-effectiveness. (Clements, 2011)

As Clements states, architecture documentation should be concise enough to be understood by new employees and it should be also concrete and nonambiguous so the implantation can be done based on it. In its most basic form, the software architecture documentation may consist of box drawn with a pen on a paper. While it may not contain all the information it could potentially relay, it still can and has proven to be potentially useful in many cases. It is already a tool to communicate the stakeholders of the system components, their relations and communications and interfaces. Clements also says that a sketch is enough, and especially in small agile teams the value is in drawing them.

The architecture documentation can be of use even long after the implementation of the project. A team might want to implement monitoring on the system network or to audit its cyber security and thus review the dependencies and communication between the components. Also, while scaling the system is significantly easier if it has been taken into consideration in the architecture design, it is also easier to implement when the documentation can be reviewed for the possible bottlenecks and scalability. There have been some projects that may have been created without architecture documentation – meaning, the architecture has most likely been planned and discussed, just not written down – that have ended up creating the documenting afterwards for these same reasons.

Figure 3 - Software Architecture Diagram illustrates a generalized example of a software diagram, made with draw.io, used as high-level software architecture documentation. It shows how a relatively simple diagram can describe system boundaries by network segmentation, the components that the system consists of, and the interaction and communication between components. In its current form it does not describe their functionality or purpose or for example about the scaling of the different components like the database or application programming interface (API). Depending on the function of the documentation the aforementioned details could be applied to the same document, or a completely different document could be created to show us how the API actually is a collection of scalable entities implemented in a microservice architecture. Also, a deployment diagram could be used to describe on what instances like servers or nodes they are deployed in. This diagram already gives us useful information for the design, developing and maintenance. Also, similar diagrams have many times proven to be useful when the cyber security team wants to inspect the system for its segmentation and communications.

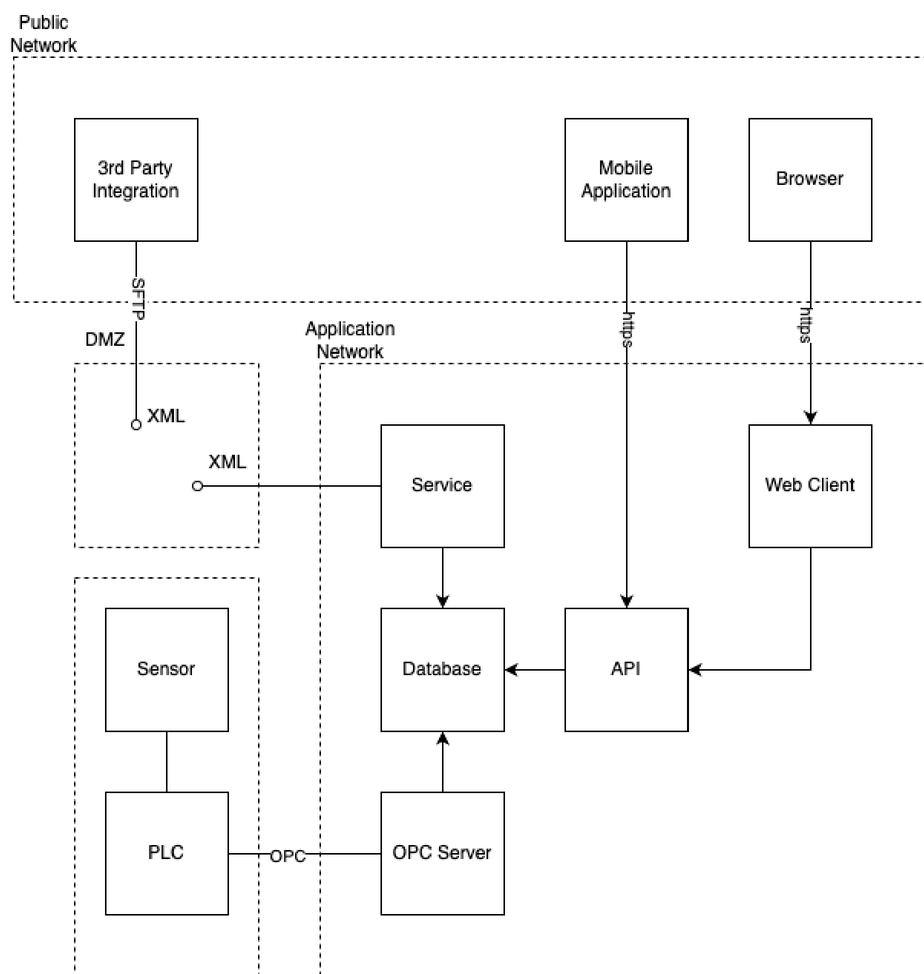

Figure 3 - Software Architecture Diagram, made with draw.io

3.4.2 Data models

By creating a conceptual model, the developer team create an overall big picture of the system. This model can then be reviewed with the team and in collaboration of the customer to go through the relevant entities and their relationships between each other. This is a good way to ensure that the relevant entities have been captured in the software. Also, it serves as a base for the logical model

The logical model is a way to design the data structure for the system. It is essentially the blueprint for the entities, their properties and relationships in more detail. With the logical model we can take the conceptual model and use it to design the data structure for the database.

Both the conceptual model and the logical model are being overlooked in many teams, as they are often seen to add workload. The experience shows though, that errors in design are easier to fix in the model than the actual implementation of the data model. When the developer finds out after few migrations that the relation should be many-to-many instead of one-to-many, the effect of changing it can be surprisingly big. Also, nowadays many tools exist, that can translate a logical model into a database. Thus, creating these documents should hardly bring any extra work, but in most cases should reduce it.

One thing to take into consideration too is that software and systems have often seen to grow in time from the original plan. When this happens, trial and error has shown that it is always easier to update the existing plans than starting from scratch at this point, when the documents usually have to be reverse engineered from the original system. This doesn’t mean that the latter should left undone. These documents have proven to be very useful for future updates, maintenance and familiarizing new developers with the system. Thus, the work of reverse engineering these documents have been found useful even after the system has already grown.

3.4.3 Process modelling diagrams

When designing functionality, starting with creating diagrams to describe it has proven to be a useful step. For teams that are not used to it this often first sounds like a superfluous task that only adds work to a task that they can easily solve in their head. It has many benefits, especially if

the functionality spans over multiple systems, like API or automation system. The diagram can serve as a wireframe of the functionality to confirm and communicate with other team members and possible representatives of the integrations. It can also be used as a as a blueprint for both the development and the testing. Diagrams like this have also repeatedly proven to be useful in the maintenance phase too, since many times it is helpful to debug the system by comparing the behavior with the diagram to find out where in the process the possible defect occurs.

Creating diagrams for the software tends to come easier the more we do it, and the initial work has time and time again proven to save from more trouble afterwards, because usually not having documentation on the system functions has often in real life lead to the team trying to decipher it from the source code, which comes more cumbersome the more there are components like integrations in the system. The are different notations for creating diagrams, and while many organizations give a lot of priority to professional appearance, it is also important for efficiency and accuracy that it is easy to use and easy to read. UML is one that has simple enough notation to be communicated unambiguously. Now that AI is developing fast, there are of course tools for creating diagrams too. User can describe the sequence to the AI, and it will create a diagram like the one in Figure 4 - Swimlane diagram created by ChatUML.

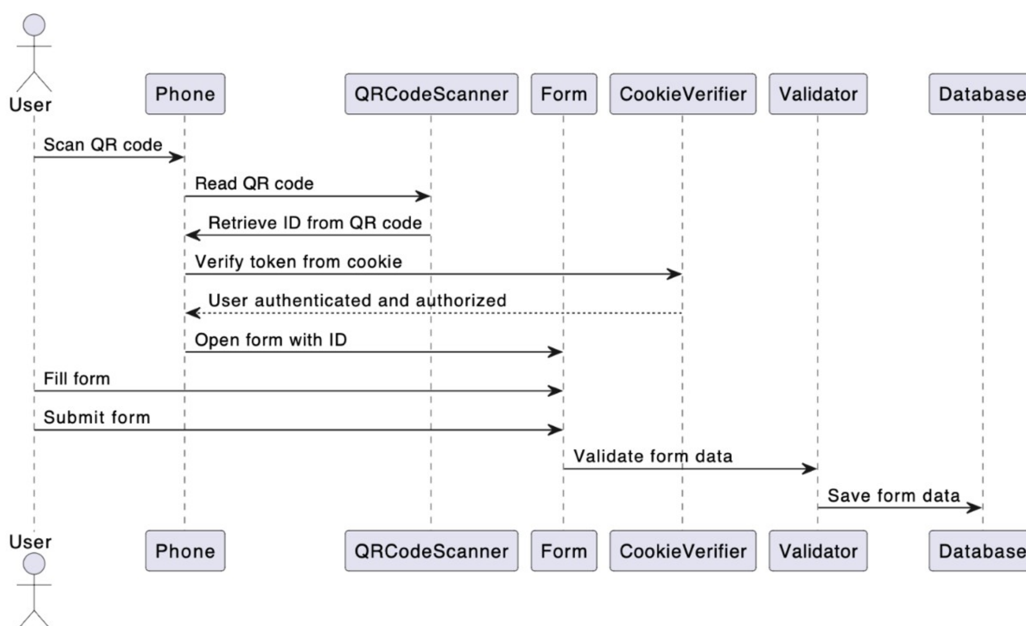

Figure 4 - Swimlane diagram created by ChatUML

3.4.4 Selecting the architecture

Selecting the right architecture depends on several factors like for example the project size, number of users and of size and skills of the team behind it. The microservices add a lot of scalability compared to a monolithic project and with individual components it possible to update and replace parts without taking the whole project down for maintenance. Then again, when debugging the system comes more complicated, when the defecting unit has to be tracked down amongst the applications and containers and the interaction with other components might need to be reviewed in order to understand the behavior. Likewise, server-rendered software differs from client-server architecture by the data they have to send over the network and scalability.

There are stories of caution about each of the software architecture patterns, and mainly this is because each of them is suitable for different things. For building a small and simple, perhaps even short-lived software the monolithic application is actually easy to build and maintain and for new developers fast to learn, but when the software grows bigger, and you have to scale it or migrate some of its components to use completely different technology it can become horrible. Client-server architecture can be great when the backend of the software may need to scale or it is the same backend needs to be used for different kinds of frontend applications like desktop and mobile, but it already adds complexity for example when the updates on the backend need to be considered on the front-end side which can be invisible to the developer due to loose coupling. Microservices are great for scaling specific functionality without scaling the whole application and adding new services without modifying the others. Of course, here the complexity grows even more, changing service components can still affect other components if they are interdependent and maintaining and documenting communication between components becomes more work. There are bad examples from both extremes of complexity: huge monolithic software that consist of huge number of features, or even applications if you look at the diversity of functionality. Then again there have been projects where the team is sure to implement microservices architecture to a system that is actually a very small application, where the none of the components will ever actually need scaling or to be used outside the one project, leading to overengineering.

As it has been established, all of the software architecture patterns have their own uses, and it is important to guide the team to select the right one for the project. Sometimes it is hard to predict the evolution of the software over time, but often there are some attributes that can be reviewed

in order to estimate the best solution. For example, whether the software is going to be locally hosted in a closed network like in an industrial setting, or if it is going to be a Software as a Service hosted in the cloud for unrestricted users like with public services, can give a rough estimate of the needs for scalability even in the future.

3.4.5 Selecting frameworks and libraries

Though in some projects the programming language and frameworks and libraries used with it might be clear from the planning phase, the design phase is often where it is either decided or at least documented. When starting new projects (or creating new modules) there is often a temptation to implement it with one's own favorite language or using the currently trending framework. There are things to consider, however. Using existing libraries versus the organization creating their own usually seems to be an easy pick with the existing libraries being less work and creating own libraries being referred as "reinventing the wheel." Then again, in many cases the existing libraries need a lot of learning, their documentation can be missing, their updates can introduce breaking changes, and their support might just end. And this is just to name a few. Of course, implementing the functionality from scratch is often time-consuming but it can be even risky and beyond the team skills, at least considering the scope of the project. This kind of functionality could be secure authentication methods or complex communication protocols. To summarize, in most, especially standard cases using 3rd party libraries is more efficient. The team still should have a strategy and review the dependencies used without just picking them blindly. modules in the project, they should be well documented, at least their purpose and version in the current CM baseline.

One example regarding frameworks and libraries comes from an organization that had bought the source code for a software and needed consulting for getting it running again. The NodeJS – project had dozens of packages used, with no documentation for their purpose. Some of the packages had no mention of their compatible version, using just a "latest"-tag. When trying to build the software, the packages were not compatible with each other and the software did not compile. There were hours of work upgrading and downgrading relevant packages and removing those that were not relevant from the beginning.

3.4.6 User Experience (UX)

The visual design is considered a relevant part of user experience (Gordon, 2022). Also, it is often important for the organization and its brand that the different software and their components are represent the same family and do not feel too disconnected. In many projects some high-level guidelines for this have been done in collaboration with the organization's marketing and management.

Another important part of user centric design (UCD) and UX is usability testing (Voil, 2019). As Voil highlights, one important measure of usability is how effectively the user can achieve their goal and how satisfactory it is to accomplish. One simple example for this is the number of clicks or touches the user needs to commit in order to finish their actions with the system. The baseline premise based on years of software projects is that the user does not want too much superfluous interaction like clicking sub links, buttons and confirmation dialogues. Then again, too much content at once can be hard to navigate and lack of confirmation can lead to the user for example submitting a wrong order. Thus, designing UX is often finding a balance of things like accessibility and reliability, rather than maximizing one aspect. Then again, another user that controls and monitors the automation process needs to see the relevant info at one glance, and while they need to be able to send commands with minimal delay, they cannot necessarily afford misclicks. The usability has to be reviewed in different context, like how the different user groups and platforms and other factors affect the way the user interacts with the software.

While testing the UX has to be considered in the testing phase of the SDLC, it should be noted that evaluating aspects like visual design and usability can and should be tested already in the design phase. This meaning getting feedback from the users with prototypes, surveys and other tools and methods of UCD.

3.4.7 Mockups

Mockups are an efficient way to start designing by prototyping the visual representation of the feature, like a component or a page in web application. With mockups the design can be created using appropriate visual tools rather than coding it with the project's programming language. This saves development time, as it takes less effort to create a version to be reviewed both inside the

organization and with the users. This way the prototype can be used to visualize and validate the idea for quicker feedback and possible reiteration in case it needs to be changed or enhanced.

There are many tools that can be used for mockups, like Figma or Penpot. These kinds of tools allow the user to select a canvas representing the target platform's screen ratio, create templates for their most used components and layouts and to create interactions between the views, for example to simulate button clicks and transitions from view to another.

3.5 Implementation

3.5.1 Version Control

One of the most important configuration items in configuration management is the source code (Horch, 2003; Leon, 2015). From the configuration management's point of view, it is essential to record its state for example in each released version. This can be accomplished by version control, which should be a basic tool in most, if not every of the software companies nowadays.

Different workflows for version control exist, and it should be implemented with efficiency in mind. Not so long ago I have been helping a small software company to transition to using git. Their previous, quite minimal version control procedure was to archive the directory of the released version to a local server and writing the changelog manually into a text file. While this works for keeping a record of the versions, it still has several weaknesses compared to modern version control. With the simplistic method the only way to know who had implemented the change and when was to write it as a comment into every changed section. Also, while working on the same software was in theory possible, it easily got chaotic to merge the changes if the developers worked on the same files or same features. Sharing the code too had to be done manually by copying the files to each other's work machines.

Version control is also used in CM to identify parallel version, for example different feature development versions, or in case where a different fork of the same software exists for different customers. This can be achieved with git for example using a proper *branching strategy*.

Version control tools can usually be integrated with other software. Some of the most popular tools for version control are GitHub, Bitbucket and GitLab and the team participating in the research uses Bitbucket. For example, by linking the source code from Atlassian's Bitbucket source control to the project management tool Jira from the same company, the development team can automate some of their workflow and improve the traceability by linking the implementation to the project management and issue tracking. For example, the integration makes it possible to automatically link the implementation in the source code to the original issue or user story.

3.5.2 Branching strategy & Branch permissions

A good branching strategy helps the team to separate features still under development, code that has gone through integration testing and is waiting for release and the production release versions. It also allows the team to create a hotfix to a bug found in a production version and release it into production without introducing any unfinished and untested code from the development. Different variations of branching strategy exist, and the organization should plan ahead for the model that suits their workflow best. Figure 5 - Example of a branching strategydescribes one commonly used model for the branching strategy.

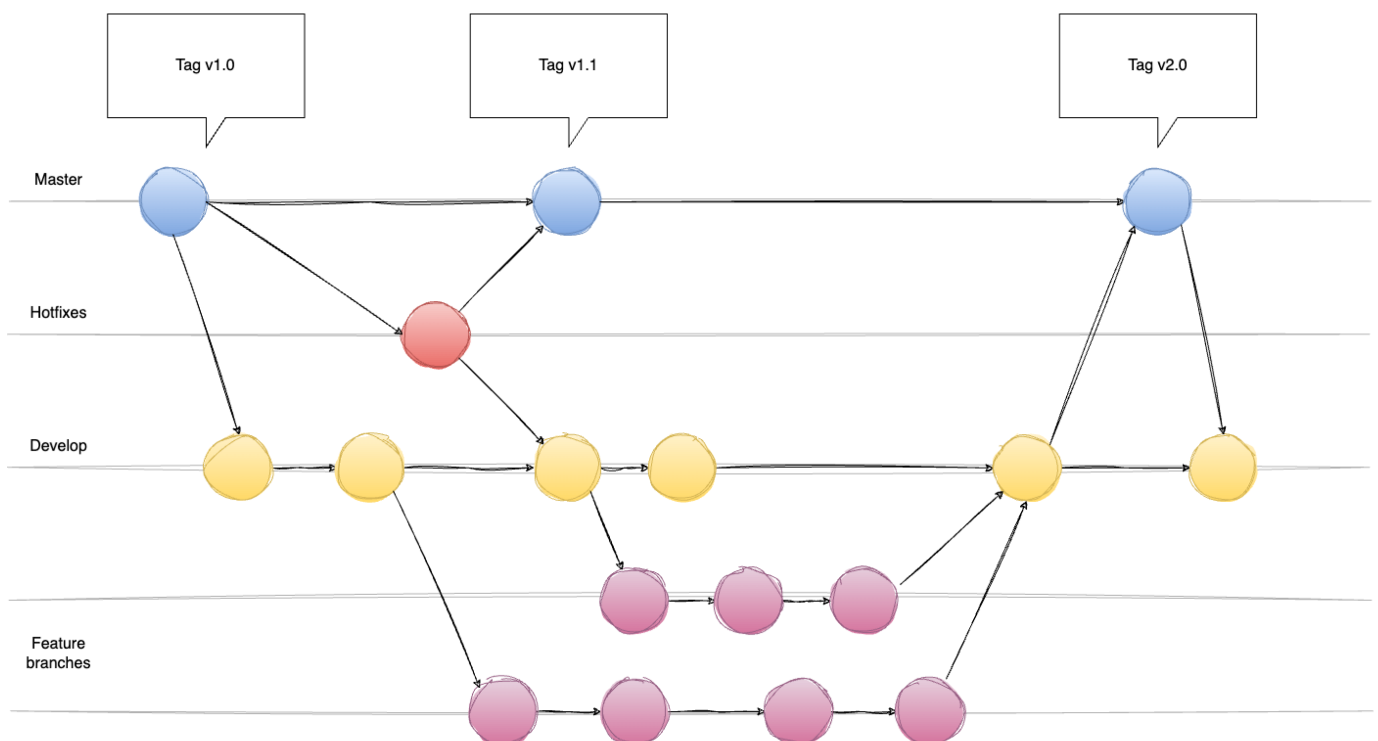

Figure 5 - Example of a branching strategy

In this model the developer team has one master branch, one development branch and one branch for each new hotfix and a feature under development to have their own respective branches. Some models also have a separate branch for a release version in staging, but like Radigan (n.d) states and it has been experienced with the development teams, having separate release branches can introduce issues within the workflow. The main benefit of having one has considered to be when the team needs to freeze a development version for release while others working on the project continue working with the development branch. One example is when all the features for the next release have been implemented and integration tested, and the operations team needs to deploy it in a preproduction environment for testing and monitoring before taking it to the production. Meanwhile, development team continue developing features for future versions and integrating them to the development branch. Now, if the release version needs minor changes that are not considered new features completely, they can still patch the release version without getting the new changes to the development branch. While this is can prove to be useful, it has felt more redundant in smaller teams and smaller projects where the development does not continue until the release version has been successfully deployed. In these cases, using the development branch as the release branch too has been more efficient and harmless.

With branch permissions the teams can maintain which roles can interact with the branches and in what way. One essential branch permission rule is to secure the master branch in a way that it cannot be pushed directly in, but only merged into from other branches, preferably with a pull request after passing required conditions like pull request reviews. Branch permissions also help the team for example to maintain commits of new employees or trainees by allowing them to commit into feature branches and create pull requests to the development branch. Branch permissions also allow the team the define who can merge the branch into another, in case the activity is allowed only for certain groups like testers.

3.5.3 Documentation

One type of documentation that has been seen lacking from different teams in different projects, is the settings, environmental variables and other factors that can change depending on the customer, deployment target, integrations or other reasons. Often it is heard that these kinds of files should not put in source control, because they can contain sensitive data, at least after someone uses them with production variables and accidentally commits. If not committed at all, the next

one trying to run the software will have to decipher a lot of variables from the source code. Thus, a good practice is to save template files with only the keys left and values stripped. The template files should also be saved with fixed names to indicate their purpose and to allow the actual files to be excluded from source control, for example by putting them in gitignore.

One another type of document that by empirical study is even more neglected is user manuals. There are different opinions on whether the user manuals should be done by in the implementation phase or in the testing phase. While the feature can be enhanced in the testing phase, it should not go through such dramatic changes that the usage would differ from the documentation's point of view. Of course, if this is not the case, the team should just consider moving the activity to the testing phase. There are at least two benefits of creating the implementation phase. One is the evident fact that the developer should know how the feature works and the other is that the user manual can be reviewed too in the test phase and even be used for the test. Based on similar reasons, some might want to even move this activity to the design phase, where the layout and the functionality could be already confirmed, thus giving the developer a clear guide for the implementation. In this case, however, some content like pictures and specifics on wording most likely need to be adjusted afterwards, as the implementation is often not one hundred percent accurate image of the design.

There are also AI based tools for creating user manuals, if the organization's strategy approves using them. Such tools can for example record the interaction with the software, produce screenshots from activities like clicking an action link or filling an input. It can then produce brief text instructions to support the pictures and let the user edit it or export the document as PDF. Now, AI tools should not be adopted for the sake of using them and the team should consider whether the tools save time after retrying and editing the captured interaction and whether the style of the produced document fits the organization's guidelines or if they need further visual formatting too.

API-documentation is also a form of documentation for API software that is essential for the users and developers as it describes how to use and integrate with the API. Using standard specification will improve both the readability and maintainability of the documentation. OpenAPI Specification for example is widely used, is clear and has many tools like the swagger editor and integrates with

development tools like Visual Studio Code to create a more visual and user friendly documentation created with the specification.

3.5.4 Unit testing

I would like to think that no one actually does testing in the actual Testing - phase. In the real life I have seen this happen though, as the developer has coded the feature all way through without ever trying to run it. In the worst cases the feature did not work as intended and either the software did not even build or then it threw a fatal error while starting the software. Needless to say, this causes in the very least unnecessary workload when the feature is passed on and sent back as a boomerang after tests that could not pass in the first place. Of course, in cases like this the rest of the process has to be good enough to catch errors defects like this to avoid them ending up to the actual release. Like in some of these cases, it might be tempting to implement a 'simple' feature without testing, thinking that running tests at this point only takes too much time and complicates the process.

Writing unit tests helps to achieve sustainable growth for the software project and in the process also can significantly contribute to better design, as the developer has to think about testability and coupling (Khorikov, 2020).

I have noticed the benefit of writing unit tests in many projects and reflecting those findings to the projects practically without them, it is clear that many problems could have been avoided. When someone else has to come back to the feature for example to add some functionality, testing can be a pain if the tests are not planned and documented beforehand. In the worst case the tester has to decipher the test cases from the source code, which can easily lead to forgetting important details. Also, when introducing new members to the project the unit tests can serve as one form of documentation on how the software should function. As Cline (2015) says, unit tests should be both written and run by the developers, and it seems logical, that when the developer writes a function, they already have an idea how it should work and thus be tested.

Unit testing has nowadays different kind of libraries for practically every significant code language, so documentation on how to write them should be found. In addition, many of those libraries can be used automate unit tests, so every function does not have to be manually tested. Thus, the

tests can also be included in Continuous Integration / Continuous Development (CI/CD) activities, allowing the unit tests to be run for example when merging a feature branch into a release branch with passing the tests as a condition of a successful merge.

As Khorikov points out, it is not enough to merely write unit tests. There are both bad and good unit tests and while bad tests can help to maintain the software integrity at the beginning, they start to just hinder the development speed in the long run. Also, it has been seen in practice that some unit tests can really be just meaningless, thus adding the workload with no practical gain. As a proposition for solution, this research suggests the unit tests to be reviewed with the actual implementation. When the unit tests will be inspected along with the code reviews and pull requests audits. This workflow should also feel natural, as comparing clear unit tests with the implementation should also serve as documentation what has been done. Also, sometimes comparing unit tests to the actual requirements or user stories can give an idea of whether the developer has grasped the essential functionality of the requirement, though not all unit tests should necessarily be traceable to more high-level requirements, as their purpose is more about verifying more specific and technical functionality. As Khorikov states, a good test should at least be clear to read and understand, should not need refactoring every time software source code changes and should not raise false alarms to be dealt with. Also, unit tests should not be written for the sake of writing them, but they should be reusable and cover important functionality, or they will just grow both the workload and the codebase, which hinders both productivity and maintainability.

3.5.5 AI Tools

There are AI tools for the implementation phase too, one of the most known being GitHub Copilot. Copilot for example can reduce simple repetitive tasks for example by completing code and creating functions from comment blocks. AI can also be used to create unit tests and to debug them when they fail. Needless to say, this can add the team's efficiency and productivity. Again, the use of such tools needs to be reviewed and approved within the organization. In addition to the general risks to review, one thing to pay attention to is that the developer understands what Copilot or the alternative writes. As Povarov and Penchikala (2022) point out, the work of the developer is much writing new features on top of existing programs. This means the developer spends more time reading and understanding existing code than writing new code. Thus, we cannot evaluate

the outcome just by its usability, but also by its understandability, or it can increase the workload later.

3.6 Testing

Like already established in the requirements analysis phase, it is essential for the tester to know how the software should normally work. The tester should be aware of the different possible outcomes, but also the different conditions on which the test should fail. For example, a software should turn specific features on when the specified conditions are met, but otherwise. In addition, with the correct user rights the conditions should be changeable, but without the permissions they should be view only. All of these need to be successfully tested in order for the tests to pass. It is often seen in real life though, that without proper test planning the tester can easily miss some of these tests. Again, if the requirement is clearly written to include all these needs and the requirement is properly linked to the implantation, the tester has already much better chances to succeed.

From experience there are multiple benefits for documenting the tests and their results. By doing this, the tests themselves can be reviewed and audited to see no important aspects were missed. The other one is reproducibility, as the tests can be reliably run again to verify results. Also, by documenting the tests other stakeholders like the customer in case of acceptance tests can be properly informed what was tested and with what results.

Depending on the organization, there might be a separate team for the testing, or the development team does it by themselves. In either case, it has proven to be a useful practice for someone else testing the completed implementation than the team member actually developing it.

With the pilot team the developer finishes developing, runs the unit tests and then submits a pull request to be reviewed by other team members. The developer many times may come blind to their own mistakes and develop a kind of a tunnel vision to their testing.

3.6.1 Integration testing

In this phase, we assume that the individual components were unit tested in the implementation phase. With integration tests we make sure that the components work with each other (Spillner & Linz, 2021). For example, different parties could be working at the frontend and the backend at the same time using test stubs in the implementation to simulate the unimplemented components and to verify the functionality of the individual, isolated component. After both components are ready, their interaction needs to be verified. The same applies when one functionality allows the user to fill out a form and another functionality is for reviewing the submitted data. Sometimes in the integration tests we find, that while both components worked as their own units, unexpected behavior arises when actually using another feature's output as the other one's input. Likewise, we sometimes need our software to work with external systems and while in development phase we run the tests with expected, simulated data, at some point we need to run the system integration tests to verify the functionality by interacting with the actual systems.

Many software use some kind of database to store and fetch data. While unit testing can many times be performed with mock data and stubs to replace the data, eventually the software needs to be tested with real data sources, as the final version of this kind of software won't probably work without the database component. In many projects I have seen different developers and testers use their own instance of a database with different kinds of test data and get different results from the tests. The test passes for one tester, but the other one gets an error while getting incompatible data. Thus, it helps to get systematic test results if there is some kind of uniform way to create the test environment, like having a script to establish and populate the database with predefined datasets, especially to cover the relevant use cases. These scripts can be stored in the source code repository where they can be maintained and version controlled, which helps to link the data structure to the version of the software and also for reliably testing a different version of the software. One thing that needs to be evaluated based on the project is whether snapshots from production database can be exported to be used in the test environment. Though using production data should give most accurate test results, many times the production data can cause privacy issues, and even if could be anonymized properly, if the customer's policies can prohibit it.

Also, software nowadays is often built in several tiers. The software may consist of only backend, frontend and the database, or it can implement the microservice – architecture and have lots of

individual components. In these cases, the manual way of doing integration tests can be very cumbersome. All the components need to be launched and possibly configured to match the same environment before that, and on top of that different developers may have different dependencies or other contributing factors in their own development environment. This is why using containerized environment like Docker can both enhance the productivity and also reduce the variables and human errors in setting up the environment. Using docker compose all of the containers can be started and cleaned up simultaneously and automatically and they can be easily set up to match production-like configurations. The scripts for containerization can be also saved in the source code repository. Container environments can also be utilized in CI/CD practices, which will be further discussed in the deployment chapter.

3.6.2 Regression tests

Regression tests verify that the existing functionality are still working as intended and that the new changes to the software do not alter or break them. Ideally, since we are testing something that we have already tested, we should be mostly able to run tests already documented, given that they were designed well to be repeatable. Depending on how well the tests can be automated, the functionality to be tested should be prioritized, because testing everything can be very laborious. Of course, with automated testing the workload is significantly smaller, and the time used for testing should not be a such a problem, since many test frameworks can run test cases parallel to each other. (Spillner, 2021)

Of course, a clear picture of dependencies can prove to be essential with regression tests. The change in the code can very well affect a component in the integration level, like changing code to an API can break the functionality of a component that is calling it.

3.6.3 Continuous Integration

With Continuous integration practices, the development branches can be kept short lived and integrated to the development branch frequently. The most used source control repositories have a feature called CI/CD pipelines, that allow running automated tests on certain triggers, like when a user pushes a commit, creates a pull request, or merges a branch The pipelines can be set to run for example only on certain branches, like the development, release or master branch. Of course,

it is possible to run pipelines on every commit on the repository, but as the build minutes can cost money depending on the provider and the platform, the team should make a strategy for the CI/CD beforehand. Using these tools, extensive integration, unit and regression tests can be run automatically every time the stable versions are updated, which become a huge amount of work if done manually. The pipelines can be configured to notify users for failed tests and the passing of the tests can be used as additional requirement for the merge of a pull request.

3.6.4 Definition of Done

Definition of Done is a set of criteria agreed within the team beforehand, that defines the requirement for the product increment to be ready for release. Thus, since the issues in *done* column of a kanban board indicate that they do not need further attention before release, they need to fulfill the criteria of DoD before they can be moved there. Some generic useful criteria of DoD would be that the tests have been run, and documentation has been created.

3.6.5 Acceptance tests

The planning of acceptance tests was discussed in chapter 3.3.5. Now that we have undergone several different types of testing, it should be noted that while the other testing focused on the system testing and software functions working as independent units and as a whole, the acceptance test is carried out to verify that the system actually fulfills the defined needs, i.e. requirements.

Some common forms of acceptance testing are user acceptance tests and operational acceptance tests. User acceptance tests are often created by the customers and are thus defined more often in business language. The operational acceptance tests focus on functional and non-functional requirements like stability and performance. In some bigger projects, especially in the industrial field, the operational acceptance tests can be repeated in different environments. For example, factory acceptance tests (FAT) are carried out in the software providers own environment to ensure that the solution actually works as intended. The same tests can then be repeated during site acceptance test (SAT) after installation and deployment to see that the same system still works in the intended manner. Repeating the tests in different environment like this ensures that the re-

quirements are met, but it is also helpful with bigger projects, because as the deploying the software into an environment with networking and integrations, possibly by different vendors, the test results can then be compared to pinpoint at what point the behavior has changed.

While the user is often the one to carry out the user acceptance tests, it is still a good practice for the developing team to ensure beforehand that the tests will pass when the time comes. Of course, when the software changes – especially in iterative development – it is possible that contradicting requirements or other factors may change the software so that some of the requirements are not met and thus the tests do not pass anymore. This is why the user acceptance should many times be automated too, and there are good tools for that too. For example, many teams have found Robot Framework useful for acceptance testing, for it is capable of testing many kinds of frameworks and doing end-to-end testing by acting with the actual user interface, for example by running a web application and simulating clicks and inputs in the UI. Another benefit of tools like Robot Framework is the keyword-driven style the tests are created on, which has been considered very intuitive and clear to write and read by others than just developers. This helps the non-technical stakeholders to participate in writing tests, which is essential since user acceptance tests often include more complex business logic that often requires some acceptance testing done completely manually.

3.6.6 Different areas of testing

When writing test, running them, or reviewing test results the one responsible for the testing should remember to test not only functionality, but also other attributes. For example, visual design and usability should be tested, so that they match the design as referenced in chapter 3.4.6. Other things to test could be security and localization.

While many non-functional requirements can be tested with automation too, some of them often require the participation of the tester to at least to verify the results of these tests. For example, the testing automation could save screenshots of the views of the software while testing and the tester could review these images to ensure that the appearance matches with the organization’s guidelines. Then again, the usability might require a human to test the “feel” of the application as they go through the features.

3.7 Deployment

One thing to consider, that not too many guides and books discuss about, but experience has proven to be very important, is the deployment plan. Whether formal or informal, it is better to at least have the steps planned ahead and then to have some idea what to do if it all fails. One example comes from a certain organization that several times called for restoring the whole virtual server from a backup, because the deployment of the update for a multitier software went so wrong the team did not know anymore how to even begin to fix it. If they had planned ahead, they would have either come up with a rollback plan or decided not to deploy the software until there was a decent one. Another type of testing relevant to user centric design and UX is usability testing.

Even more recommended is to have a preproduction or beta testing environment. It should not replace the systematic plan for the actual production deployment, but it still helps to prepare for it, since it gives us some idea of whether the system is working without errors. Perhaps the most simple solution is to have the same software duplicated in the same server environment and just creating another way to access it, like accessing it from a different URL in case of web application.

After deploying the new production version, it is common to merge the release branch to the master branch and to tag the commit with the version number or name. Thus, we have a branch where we know the most recent commit is always deployable and do not need to search the branch for one. Also, by numbering or otherwise identifying the version it will be traceable, and we can link for example what bugs and defects are introduced in which version and in which they are fixed.

Following CM best practices, we should also create a new baseline of the whole configuration at this point. For example, this means we record the state of the requirements at this point of the project life cycle. This helps for example, when changes to requirements are made and new versions are created. In some projects the customer has claimed that the software does not meet the requirements that were specified at the start of the project. Reviewing the version baselines, we can see that the first release version indeed fulfilled the original requirements, but later on new changes were requested and the new version had adapted to meet those requirements. Also, the settings and environmental should be recorded somewhere. Sometimes, especially when they

contain secrets like connection strings and API-keys or customer specific configuration, they can be too sensitive to put in the source control repository. Still, if we want to replicate the production environment or to test features against different configurations, it can be from hard to impossible without knowing the right configuration.

In case we are deploying the product for a specific customer or otherwise have different release versions of the software, it is also crucial to record the instance. This helps us to monitor issues like defects and vulnerabilities and map which instances or customers they affect. Also, when the customer reports a defect, we know what version we can reproduce with it and also to which version to update their software or which version to apply a hotfix to.

By using continuous deployment, we can automate the release of the code changes into the production (Susnjara & Smalley 2024). This means that we can use tools to monitor for merges to the master branch and act on them running scripts to complete the steps for deploying the software. Of course this is not necessarily possible in all environemts, like closed systems. Also, in industrial environments it is common that someone is required to be even physically present to monitor the deployment, for depending on the system it can have severe effects on the business. Where it is applicable, it can save a lot of time and work. Deploying the software can be a cumbersome task and there can be human errors on the way too. Of course continuous deployment needs preparations like the branching strategy and permissons and good continuous integration to avoid accidental, lacking or defected commits to trigger the deployment. These topics were discussed in the previous chapters. While not all environments allow the continuous deployment in its fullest, it is still useful to use CI/CD pipelines to build the release version, that can be downloaded from the source control repository. This ensures that the build of the software has been done in a clean environment and all the proper steps are done, in contrast to the team members building the versions of the softwre on their own machines, ending up with potentially seemingly same versions of the software with slight variations.

3.8 Maintenance

R. S. Pressman emphasizes in his foreword that software maintenance is *the* important phase and while challenging in itself, it becomes a pain if not planned ahead accordingly (Reifer, 2011). Changing requirements, updating design and code and retesting are listed as the easy parts of the

maintenance. Reifer also mentions, that many times maintenance is not properly scoped, planned and budgeted. These both statements support the point of view that the maintenance phase needs standardizing both in thinking ahead and then the actual doing. It can sound obvious to say we need to do it properly, but also, we need to plan it ahead, because in the real projects we cannot do it properly if we have not allocated resources and made proper plans for it.

Comparing the iterative and sequential project flows, some activities fall into the maintenance phase, and some are part of the new iteration's beginning. For example, the need of new requirements and enhancing existing ones can be viewed as maintenance, if the sequential project has already been deployed. In iterative project however, this is can be part of the new requirement gathering and analysis phase. In both cases, however, after the first deployment there is often the need to monitor the system and for example provide updates for vulnerabilities.

One important activity in the maintenance phase is issue tracking. It should be safe to say that most system are prone to defects, errors faults or failures. On top of that, as technology evolves, vulnerabilities are introduced in many frameworks and dependencies used in the system. It has been found crucial to properly log these issues in a way they are traced to the versions of software they affect. There can sometimes be situations where a user reports a defect in the system and the problem is solved while troubleshooting it. Maybe there was a problem in the connection, and it resolved when the user restarted the software. In these cases, the problem might not even make it to the issue tracker when it is already solved. It is still important that the problem and the solution are logged into the system, as it can prove to be useful when debugging similar issues in the future. Also, some issues might be reviewed by their frequency and impact on the user experience to determine their priority. By logging these instances of occurrence, we can see statistics on how frequently the defect has been encountered and also in which version of the software it has been first discovered.

4 Discussion

The objectives of the research were to compile project working knowledge and to use that knowledge to standardize and improve the software project process by helping in managing the areas of

software development like requirement management, developing and quality assurance. The improvement of the process was aimed to reduce variation and lead times and improve productivity of the team and the quality of their software projects and some supporting processes.

The research was carried out by identifying, improving and documenting the pilot team's software project processes. These documents where compiled into one, building a standardizing project guide for the organization. This was done iterative and incrementally, so the results could be tested in practice and adjusted when necessary, so the result for the changes could be reviewed independently from each other and the team would not be overwhelmed of too big changes at once.

As the resulting artifact, a project handbook was created, that covers the processes identified and documented while doing the research. As the result of the research, some processes were also enhanced by applying the best practices of project management that were studied as part of the research. The handbook prevents variation in workflows and thus in results, as people have the same procedures to be followed and also a single source for the team members to check them so nothing is forgotten.

Some parts of the early versions of the handbook were introduced to an interim employee, which showed promising results regarding the premise that the handbook could be used to onboarding a new employee. While there were no new employees at the time of the research, some new methods were adopted as part of the research, and they were successfully introduced to the old team members through the handbook.

The Design Science Research method itself proved to be an applicable and useful method for the research. The phases of the DSR are very relatable when researching software development research, as they have many similarities, with planning, requirements gathering, designing and developing and then testing as a comparison to demonstrating and evaluating. The iterative and incremental approach also familiar from software development suits well for the research, because it allows the researcher and the pilot team to focus on small portion of improvements and their results at a time.

While the research was deemed a success, it should be noted that it could benefit from a bigger sampling preferably with more variety in size and type of projects and teams. For example, a bigger team, team with more specified roles or a team fully implementing Agile methods like Scrum could have variations in the workflow.

Regardless of the limitations on the sampling, based on the literature research and previous experience of software projects the results should be useful in these cases too, though with some adaptation. The handbook created here should be viewed as a template, where the organization should insert and edit their own standards on how to implement the process, while some of them could be similar to the ways of the ways of the pilot team. Some teams might want to add activities, and some might choose to go with a lighter version, based on the workflows and projects used by the team.

For further reading, different areas of the software project life cycle could be researched more deeply. Though the research introduced many good practices for its different phases, the different methods and tools were not addressed very deeply, since the focus was more on the improvement by standardizing and documenting them. Also, as there are variations on how those different activities should be adopted on different organizations and teams, it is possible that the team wants to look into alternatives ways of implementing the specific topics. For project managers, leads and other roles responsible for the project as whole, should possibly familiarize themselves with the whole lifecycle to the extent of their responsibility. This also applies to organizations with smaller teams like the pilot group. Those team members with more specified roles like specific testers might want to focus on the phases relevant to their work, so the team members also know their work's best practices and can be part of improving them.

As process improvement is a continuous effort and the technology and methods on software development evolve constantly, it goes without saying that the processes and thus this handbook should be developed and improved further over time. Thus, this research is considered as the first version of the artifact and by all means not final iteration in its development.

The goal of this research was not so much to find completely new ways to manage software projects, but to gather them and introduce them as a way that gives concrete tools for the software project team. Compared to other related work, it gives more practical and detailed answers than general introductions on software project life cycle. Also, it gives methods and real-life justifications for following the advice found in many software project improvement guides and combines good practices usually found from guides for those specific areas like books on testing or requirement management.

An additional note made regarding the research and its resulting artifact, that the project handbook should be useful also when aiming for compliancy with standards like ISO27001. Many of its topics, when documented properly, are relevant to the standard, like project risk assessment, access to source code and other areas discussed in the research and documented in the handbook as commissioner’s workflows.

References

Affairs, A. S. for P. (2013, November 13). Usability testing. Usability.gov. https://www.usability.gov/how-to-and-tools/methods/usability-testing.html

Alexandra. (2023, May). What Is SDLC? Understand the Software Development Life Cycle. Stackify. https://stackify.com/what-is-sdlc/.

Artificial Intelligence Market Size, Share, Growth Report 2030. 2023. Grand View Research. Cited October 2023. https://www.grandviewresearch.com/industry-analysis/artificial-intelligence-ai-market.

Belanger, A. 2023. Will ChatGPT's hallucinations be allowed to ruin your life? Cited October 2023. https://arstechnica.com/tech-policy/2023/10/will-chatgpts-hallucinations-be-allowed-to-ruin-your-life/?utm_source=tldrnewsletter

Bhatti, J., & Hightower, K. (2021). *Docs for Developers: An Engineer's Field Guide to Technical Writing* (1st ed. 2021.). Apress. https://doi.org/10.1007/978-1-4842-7217-6

Coffman, C. 2023. Does the Use of Copyrighted Works to Train AI Qualify as a Fair Use? Cited November 2023. https://copyrightalliance.org/copyrighted-works-training-ai-fair-use/

vom Brocke, J., Hevner, A., & Maedche, A. (2020, September). (PDF) introduction to design science research. Research Gate. https://www.researchgate.net/publication/345430098_Introduction_to_Design_Science_Research

Budinski, K. G. (2001). *Engineers' guide to technical writing*. ASM International.

Carreira, B., & Trudell, B. (2006). *Lean Six Sigma that works: A powerful action plan for dramatically improving quality, increasing speed, and reducing waste*. AMACOM – Book Division of American Management Association.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde de Oliveira Pinto, H., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv*. https://arxiv.org/abs/2107.03374

Cline, A. (2015). *Agile Development in the Real World* (1st ed. 2015.). Apress. https://doi.org/10.1007/978-1-4842-1679-8

Clements, P. (2011). *Documenting software architectures: Views and beyond* (2nd ed.). Addison-Wesley.

Dresch, A., Lacerda, D. P., &; Antunes, J. A. V. (2015). Design science research: A method for science and technology advancement. Janet Finna. https://janet.finna.fi/Record/jamk.993580554806251?sid=3392366920

Fantina, R. (2005). *Practical software process improvement*. Artech House.

Franzen, C. 2023. Meet Nightshade, the new tool allowing artists to ‘poison’ AI models with corrupted training data. Cited November 2023. https://venturebeat.com/ai/meet-nightshade-the-new-tool-allowing-artists-to-poison-ai-models-with-corrupted-training-data/?utm_source=tldrnewsletter.

Full Stack Academy Team. 2023. Addressing Bias in AI. Cited in November 2023. https://www.fullstackacademy.com/blog/addressing-bias-in-ai

Goodfellow I., Papernot N., Huang S., Duan Y., Abbeel, P., Clark J. 2017. Attacking machine learning with adversarial examples. Cited October 2023. https://openai.com/research/attacking-machine-learning-with-adversarial-examples

Gordon, K. (2022, November 6). *Visual Design in UX: Study Guide*. Nielsen Norman Group. https://www.nngroup.com/articles/visual-design-in-ux-study-guide/

Horch, J. (2003). Practical guide to software quality management, second edition. Artech House.

Jain, N. (2011). Software Development Life Cycle: A Detailed Study. International Journal of Advanced Research in Computer Science

Johannesson, P. P. (2014, January 1). An introduction to design science. An Introduction to Design Science | Jyväskylän ammattikorkeakoulu | Janet Finna. https://janet.finna.fi/Record/jamk.993721568006251?sid=33923

Khorikov, V., & Khorikov, V. (2020). *Unit testing: Principles, practices, and patterns* (1st edition.). Manning.

Larrucea, X., O'Connor, R. V., Colomo-Palacios, R., & Laporte, C. Y. (2016). Software Process Improvement in Very Small Organizations. *IEEE software*, *33*(2), 85-89. https://doi.org/10.1109/MS.2016.42

Leon, Alexis. (2015). Software configuration management handbook, third edition.

Liu, J. Y., Chen, V. J., Chan, C., & Lie, T. (2008). The impact of software process standardization on software flexibility and project management performance: Control theory perspective. *Information and software technology*, *50*(9), 889-896. https://doi.org/10.1016/j.infsof.2008.01.002

Oktaba, H., Garcia, F., Piattini, M., Ruiz, F., Pino, F., & Alquicira, C. (2007). Software Process Improvement: The Competisoft Project. *Computer (Long Beach, Calif.)*, *40*(10), 21-28. https://doi.org/10.1109/MC.2007.361

Panetta, K. 2023. Set Up Now for AI to Augment Software Development. Cited November 2023. https://www.gartner.com/en/articles/set-up-now-for-ai-to-augment-software-development

Povarov, N., Penchikala S. 2022. AI for Software Developers: a Future or a New Reality? Cited November 2023. https://www.infoq.com/articles/ai-for-software-developers/

Radigan, D. (n.d.). A guide to optimal branching strategies in git. Atlassian. https://www.atlassian.com/agile/software-development/branching

Reifer, D. J. (2011). *Software maintenance success recipes*. Auerbach Publishers, Incorporated.

Rüping, Andreas. (2003). Agile documentation: a pattern guide to producing lightweight documents for software projects.

Spillner, A., Linz, T., Spillner, A., & Linz, T. (2021). *Software testing foundations* (Fifth revised and updated edition.). Dpunkt.verlag GmbH.

Stewart, L. (2023, September 24). How to conduct effective pilot tests: Tips and tricks. ATLAS.ti. https://atlasti.com/research-hub/pilot-test

Stober, T & Hansmann, U. (2010). Agile software development: best practices for large software development projects.

Sun, Z., Du, X., Song, F., Ni, M., Li, L. 2022. CoProtector: Protect Open-Source Code against Unauthorized Training Usage with Data Poisoning. Cited October 2023. https://xiaoningdu.github.io

Susnjara, S & Smalley, I. (2024). What is continuous deployment?. IBM. https://www.ibm.com/topics/continuous-deployment

Suvorova. (2024). What is Presales in IT and How it Helps to Hire the Right IT Service Company. Exposit. A. https://www.exposit.com/blog/what-is-presales-in-it/

Tozzi, C. (2024). The 7 stages of the SDLC explained. https://www.techtarget.com/searchSoftwareQuality/tip/The-stages-of-the-SDLC-explained

Voil, Nick de (2019). User experience foundations. BCS Learning & Development Lim