Towards a framework for risk identification and mitigation in Agile software re-engineering: A case study

by

Chiara Maria Fasching

A thesis submitted in partial fulfilment of the requirements for the degree of MSc (by research) at the University of Central Lancashire.

October 2023

Abstract

Legacy software is becoming increasingly ubiquitous, and many companies nowadays are facing the challenges associated with this phenomenon. In certain circumstances, re-engineering is the only logical way to deal with legacy software. Such projects, by their very nature, are subject to a wide variety of risk. The first research question of this study was concerned with the identification and investigation of different risks associated with one particular re-engineering project. The potential impact of those risks over a given development phase of the project, along with their actual impact and ultimate mitigation status, have been analysed. The second research question discussed the Scrum practices used on the project and evaluated their usefulness in supporting the management of risk. An interpretive Case Study approach has been followed, with the method of analysis being inductive and reflexive Thematic Analysis. The risks observed were themed around people, process, and technology. While technical and procedural risks are discussed in the literature, the presence of risk in social situations relating to re-engineering appears to have been overlooked. Although these risks do not necessarily have a higher impact, they were found to outnumber those encountered in other aspects of the project by a significant factor. Furthermore, the social risks were often either underestimated or not even recognised. It has also been found that Scrum is an appropriate approach to re-engineering projects. Since many of the re-engineering tasks in the case study were unknown at the beginning, the flexibility brought by Scrum was an important factor in the timely and successful mitigation of emerging risk. My contribution to the field is an initial set of risk categories including their impact on a re-engineering project, which are able to form the basis of further research into different types of re-engineering projects in order to produce a more generalised framework. It is anticipated that the results presented here will help future project teams to prioritise aeras of re-engineering and put adequate risk mitigation into place.

Acknowledgements

First and foremost, I would like to thank my Director of Studies, Dr Peggy Gregory, who picked up on this project on very short notice, but still put all her effort and patience into it. I also want to thank her for her ongoing support and interest for my research.

I would also like to thank my supervisor, Dr Nicholas Mitchell, for his never-ending patience for my stream of endless questions, his wise words and for keeping me motivated through, what it feels like, an endless journey!

Next, I want to thank my second supervisor, Professor Charlie Frowd, for involving me in such an interesting project, which was a once in a lifetime chance. I also want to thank him for his never-ending support.

I want to thank all my supervisors for their, almost parental, love and care for me, which I must say, was exceptional. Thank you so very much. I do not know what I would have done without you all!

I also must express my many thanks to my parents, who keep supporting me, even though I am so far away now. Thank you to my mum for her unconditional love and support, her care packages, and everything you have ever done for me. Thank you to my dad, who got me interested in Software Engineering and made me to this curious woman I am now. Thank you for my stepdad, who always supported and believed in me, and who was always there for me. I also want to thank everyone else of my family. I love and miss you all.

I also want to thank Linda and James, who are always there for me.

Finally, I would like to thank Lola, my loveliest hamster girl, who I would like to dedicate this thesis to. She supported me morally from the beginning to the end of this thesis, kept me sane, and was there when no one else was. I love and miss you so much. I know you are not resting in peace, because you are getting up to mischief in hamster heaven!

Contents

1. Introduction

1.1. The history of programming

The concept of programming – that is to say the idea of setting out a sequence of instructions that could be followed by a machine – has fascinated humanity for centuries. Even before the invention of the modern programmable computer and the accompanying term "software", which was first used in 1958 (Fitzgerald, 2012), mechanical means of controlling the logical behaviour of machines was widespread. Take for example the 18th Jacquard loom, which used punched card, an early form of storing data sets, to enable the automatization of weaving complex patterns into cloth or mechanical interlocking in railway signal boxes setting routes and controlling signals. Even further back in history we find the polymath Al-Jazari, who invented a programmable castle clock in the 13th century, which contained pure hardware and no software (Tatnall & Davey, 2016). We can see that the ideas of "modern" programming, such as sequences and conditions, were developed centuries before the first modern computer.

The first algorithm as we would recognise it was written by the mathematician Ada Lovelace, in the 18th century (O'Regan, 2013). Even though her algorithm remained theoretical, she is considered by many to be the first computer programmer. Fast forward to the 20th century, and Alan Turing proposed the Turing machine, a model of a universal computer, which gave a formal definition to the term computable (Strawn, 2014) and allowed mathematicians to begin exploring what would be theoretically possible to achieve with a programmable machine. Turing is not just the father of computer science, but also made a sizeable contribution to Artificial Intelligence (AI) with the Turing test. Furthermore, his work on codebreaking during the second world war famously led to the cracking of the German Enigma code using electro-mechanical devices (perhaps shortening the war by up to two years).

These experiences pave the way for post-war work at Manchester university, which ultimately led to world's first stored program computer, the Manchester Baby – a device which could be re-programmed without making physical alterations using electric memory, and the dawn of the software age. However, the early years of computer software were still characterised by punched cards – over two hundred years after their first introduction, which were also used for the Apollo Mission that needed software to run on the computers of the landing machines. The first compiler, a program which translates programming languages to binary, was written in 1951 by Grace Hopper (Strawn & Strawn, 2015). Six years later the first major programming language, FOTRAN, designed by IBM for scientific computing, appeared. Including statements such as IF, DO, and GOTO, which were considered a big step forward. Soon more programming languages followed: Cobol, PL/1, Pascal, and C, which influenced many later languages such as Java, C#, C++ and Python, which are the major languages today.

The year when computing truly became mainstream was 1975 (Wirth, 2008), which was triggered by two major events: affordable microcomputers appeared on the market, and a cheap, high-level compiler, at a time when compilers were very expensive, for Pascal. Programming became accessible for the average citizen, and soon software started pervading everyone's life. In the 1980s companies saw the potential of user-friendly software, and soon Apple and Microsoft started producing fast and convenient operating systems. By 2008 over one billion personal computers were sold. By 2016 75% of the world's population owned a smartphone (Sommerville, 2016). Today software engineering covers a broad spectrum of disciplines and shapes our everyday life. From health and science, over social science, management, and manufacturing, to security, and policing, software can no longer be imagined away from our lives (Kazman & Pasquale, 2020).

1.2. The software crisis and the introduction of software eningeering

However, the rocket-like rise of software since the 1950s brought problems within the software development world. Within 10 years computer power increased rapidly and problems to be solved became more complex. The demand for useful and efficient software could not be met and led to a software crisis (Naur & Randell, 1969). Software projects never overcame this crisis and since this period software project delivery delays, budget overruns, and software developer shortages became a persistent problem. In 1991 the increased software demand resulted in a shortage of approximately 1,000,000 software engineers (Johnson, 1996) and in 2021 it was estimated that the shortage had increased to 4,000,000 (Dayaratna, 2021).

By the 1990s there was a proliferation of legacy. Bennett (1995) escribed the difficulties with legacy systems in 1995. The first applications were already 40 years in operation and remedial action had not been taken on the majority of them (Bennett, 1995). Clarity and program structure was sacrificed for program speed, and features were added in an ad-hoc manner. Legacy applications persist to this day, with up to 200 billion lines of legacy code still in use (Fanelli et al., 2016).

Looking at how the software crisis manifested itself, including projects running late and over budget, unmet requirements, low quality software, and never delivered software, most of the problems came down to inappropriate project management (Fitzgerald, 2012). In 1968, these difficulties were openly discussed at a NATO sponsored conference, where it was recognised that the current software development techniques were inadequate and new methodologies needed to be invented which could adapt to the fast-paced change in software development (Wirth, 2008). Retrospectively it was not surprising that poor project management was one of the major causes, as programming was seen as a pure mathematical discipline for a long time. Programmers were good at writing algorithms, but not at working on projects.

1.3. Early software methodologies

During this NATO conference, the term software crisis was first coined, and also the term software engineering, which refers to software development as a highly disciplined approach of writing and maintaining software (Wirth, 2008). Software engineering lifts software development from writing algorithms up to managing everything which is involved in the software life cycle, including the management of software projects. In the early years of software development, “code and fix” was the method employed (Awad, 2005). Due to the difficult nature of this approach, Winston Royce proposed the first methodology, the Waterfall model. Soon more methodologies followed, most of them based on sequential series of steps. These are known today as traditional or heavyweight methodologies. These methodologies were seen as the solution to the software crisis, however, it was realised after a while that even though, they were helpful, heavyweight methodologies only tackled parts of the problem.

To visualise why traditional methodologies did not solve the problem completely, the “Iron Triangle” can be used. The triangle describes the three basic factors which influence software quality. These three factors are cost, features, and time. To ensure the success of the project, no more than two of these factors can be fixed without sacrificing quality (Lehman & Sharma, 2011). In traditional methods, the requirements are usually the fixed scope. This often leads to late or overpriced software, with features which might not even be relevant to the customer anymore.

1.4. How Agile revolutionised software engineering

However, around the mid-1990s a new methodology was mentioned in multiple publications describing new software development approaches such as Extreme Programming and Scrum (Fowler et al., 2001). In 2001 the Agile Manifesto was published, describing Agile as a “lighter approach to building software”. The manifesto states the values Agile represents, which are:

- Individuals and interactions over processes and tools.

- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan

Since then, Agile has increased in popularity. A survey conducted at Microsoft in 2007 pointed out that already 30% of the participants used Agile as a software development methodology (Begel & Nagappan, 2007). In 2011 the number increased to 66% highlighted by a different survey conducted by (Azizyan et al., 2011). A third survey conducted by the Xebia's Agile Survey in 2012 revealed that even 80% of the respondents used Agile for software development (Matharu et al., 2015). The most recent numbers show an increase of Agile adoptions in software development teams from 37% in 2020 to 86% in 2021 (*15th State of Agile Report*, 2021).

The reason why Agile became so popular is that it increases the success of software development projects. A report published by The Standish Group in 2011 concludes that Agile is three times more successful than traditional software development approaches (Matharu et al., 2015). This success can be described with the Iron Triangle again. Instead of requirements being fixed at the beginning, in Agile the cost and the time are fixed, while the features are estimated and more flexible. This enables the software developers to prioritise the features of the application according to the business needs, which leads to on-time delivered quality software with the biggest value for the money.

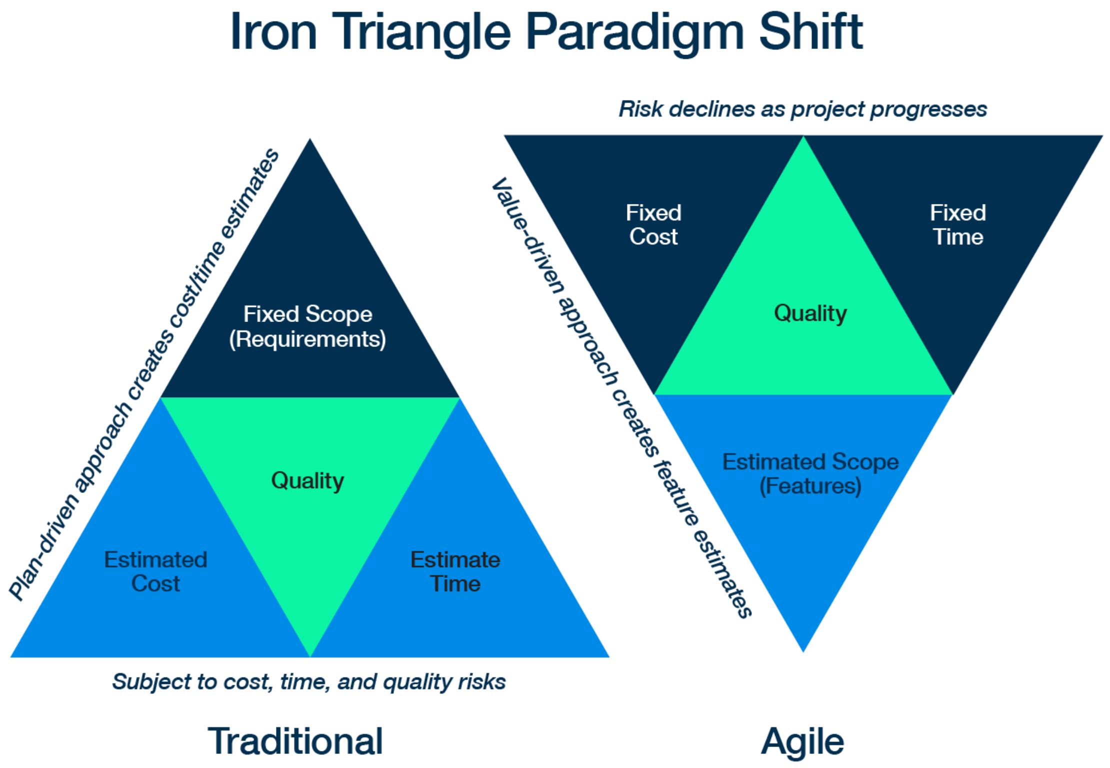
Iron Triangle Paradigm Shift
Risk declines as project progresses
Plan-driven approach creates cost/time estimates
Fixed Scope (Requirements)
Quality
Estimated Cost
Estimate Time
Subject to cost, time, and quality risks
Traditional
Value-driven approach creates feature estimates
Fixed Cost
Fixed Time
Quality
Estimated Scope (Features)
Agile

Figure 1: Iron Triangle Paradigm Shift. Photo: CleanPNG/Cowpaw

The flexibility of requirements, and with that the re-evaluation of priorities during the process, makes Agile a superior approach for re-engineering legacy software. In 2014 Masood & Ali (2014) describe Agile, more specifically Scrum, as a handy approach in Brownfield (re-engineering) projects. Holvitie et al. (2018) also point out how Agile can be used to reduce technical debt. Legacy applications often contain a large amount of technical debt (Birchall, 2016). Further review of Agile used in re-engineering projects will follow in the literature review chapter.

1.5. Introduction to this research

The significance of this research stems from a lack of research papers about agile in re-engineering. Some grey literature sources even suggest that Agile is an inappropriate approach (Diana, 2010). Moreover, I have discovered that re-engineering risks are an uncharted area as the risks mentioned in the literature are barely overlapping, which will be discussed further in the following chapter. My proposed framework will be novel as no re-engineering framework to this extent exists. Rajavat & Tokekar (2011) proposed an agile re-engineering framework, but it takes decision concerning the success of a re-engineering project and does not analyse the whole re-engineering process nor the challenges and risks involved.

The two research questions which will be answered in this thesis are:

- RQ1: What types of risks are encountered in a software re-engineering project and how are they mitigated?
- RQ2: How helpful are Scrum practices to support a software re-engineering process?

To answer these questions, a case study of a re-engineering project using an Agile approach was conducted. The data collected during the case study were the risks encountered and how, if so, they were mitigated. Using Thematic Analysis the data was analysed to find patterns between the type, severity and time of the risk encountered, and if, risks were mitigated and how long it took. Moreover, a retrospective evaluation was undertaken of which aspects of Scrum supported the re-engineering. Both of these analyses were used to build an initial framework to describe risk identification and mitigation during a re-engineering process using an Agile approach. An overview of the research process is shown in Figure 2 below.

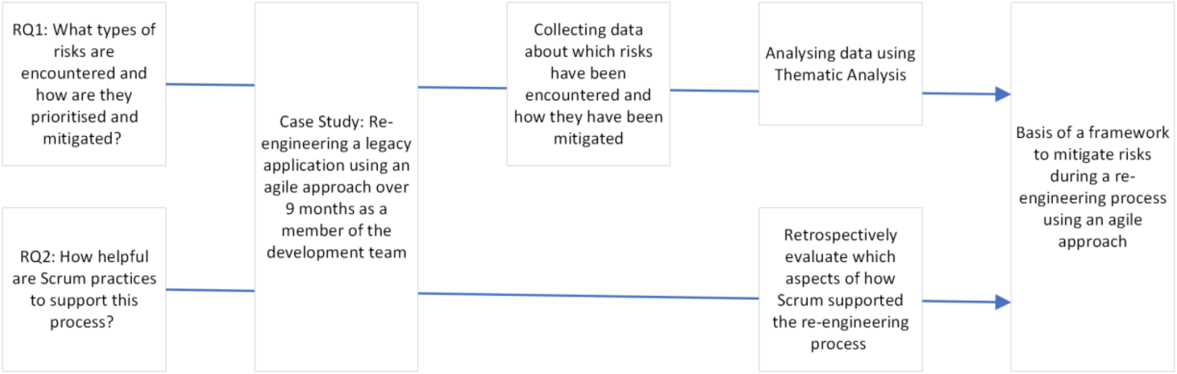
RQ1: What types of risks are encountered and how are they prioritised and mitigated?
RQ2: How helpful are Scrum practices to support this process?
Case Study: Re-engineering a legacy application using an agile approach over 9 months as a member of the development team
Collecting data about which risks have been encountered and how they have been mitigated
Analysing data using Thematic Analysis
Retrospectively evaluate which aspects of how Scrum supported the re-engineering process
Basis of a framework to mitigate risks during a re-engineering process using an agile approach

Figure 2: Study process

The remainder of this thesis is structured as follows. Chapter 3 discusses background information and literature about re-engineering legacy applications and associated risks. It also gives an insight in the existing re-engineering frameworks in the literature. Chapter 4 describes the research method, and the approach used for data analysis. In Chapter 5 a detailed insight into the case study is given. Chapter 6 presents the risk analysis and results, and Chapter 7 presents and analysis of the different facets of Scrum used to support the re-engineering process. In Chapter 8 the findings are discussed and compared to already existing frameworks. Chapter 9 looks into the future and how this study can be used as the basis of a risk framework. Finally, Chapter 10 concludes my research and proposes suggestions to practitioners.

2. Literature Review

2.1. What is legacy software?

In general legacy applications can be described as old, but well-established software systems, which are also essential for business process support (Sommerville, 2000). However, there is not just one definition of legacy applications. Bennett (1995) defines legacy software informally as “large software that we don’t know how to cope with but that are vital to our organisation”. Seacord et al. (2003) state that software applications turn legacy when they begin to resist modification and evolution. Birchall (2016) describes it as often large, inherited software which lacks testing, but is also difficult to test, consists of inflexible code, and accumulated technical debt. Surprisingly, academics and practitioners have different ways of describing legacy systems. Academics tend to look at them as obsolete systems which consist of outdated technology, whereas practitioners look at them as core company processes that need updating (Khadka et al., 2014). A survey, conducted by Khadka et al. (2014), compares 28 practitioners‘ perceptions to those of established academics. The most important, but also obvious, finding is that legacy systems are critical. One of the participants would go even so far to say that a legacy system is business critical by definition. This is because where legacy applications are still in use many years after having been developed, they are reliable systems built with proven technology. The definitions important for this study are those of Sommerville and Bennett.

There are clearly benefits of legacy applications as there are still 180-200 billion lines of legacy code in use (Fanelli et al., 2016). However, both sides, academics and practitioners, agree on the fact that a legacy application is inflexible and expensive to maintain, although practitioners often hesitate to upgrade a system if it is not broken. Fanelli identifies a number of drivers for practitioners to modernise legacy systems. Over 80% of participants reported that needing to be “flexible to change”, and the “high cost of maintenance” were strong or a very strong drivers for modernisation. The next most important factors, indicated by over 70% of respondents were “faster time to market”, and “lack of experts/documentation”. Less important, but still significant, drivers were “creating business opportunities”, and “lack of suppliers”.

2.2. What makes legacy software legacy?

The source of high maintenance is often so-called technical debt. Almost every software developer defines technical debt in a slightly different way, but the meaning stays the same. The term technical debt is often used as a metaphor to describe numerous software quality problems as it can be used to send a strong signal to technical and non-technical audiences. If technical debt is ignored, it may get worse (Ernst et al., 2015). It is agreed that technical debt is tightly connected to software quality. Eberhard Wolff (Wolff & Johann, 2015) proposes two different types of software quality: an external one, which is measured by the customers, and an internal quality, which can only be perceived by programmers. This internal quality can be described as anything that makes extending and maintaining the code easier or more difficult, including tests, architectural styles, and coding issues. Sven Johann adds that internal quality will eventually affect external quality too, which makes it to a companywide problem. Buschmann (2011) describes technical debt as a trade-off between writing clean, but expensive code and writing messy code, which can be delivered cheap and fast. The maintenance cost of the latter is higher once delivered. Buschmann compares technical debt with financial debt by saying “it supports quick development at the cost of compound interest to be paid later”. He adds that over time this technical debt needs to be paid back.

2.3. How do we deal with legacy software?

Sommerville (2000) states that it is necessary for companies to re-engineer legacy applications to keep them in service. The term re-engineering applies to a set of activities and techniques to tidy up the underlying structure of the application code without affecting its functionality. These activities include the analysis, redesign, restructuring, and re-implementing of the software system (Jain &

Chana, 2015). The general aim of such activity is to reduce the ongoing maintenance cost of a system by improving its quality (Singh et al., 2019). Re-engineering is also an umbrella term for specific processes such as forward engineering, reverse engineering, and refactoring (Jain & Chana, 2015). Opdyke (1992) defines refactoring as the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure". Fowler (2018) expands the definition by describing it as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour". Refactoring does not necessarily needs to take place during a re-engineering process, but is a continuous process throughout the development cycle (Sommerville, 2016). Sommerville (2016) describes it as "preventative maintenance" which reduces the problems of introducing future features. Forward engineering is the traditional re-engineering process of moving from high-level abstractions, logical designs to the physical implementation of the system (Rashid et al., 2013). Sommerville (2000) describes the difference between software re-engineering and forward engineering, stating that forward engineering starts with the system specification and involves the design and implementation of a new system whereas re-engineering starts with an existing system. Nonetheless, forward engineering can be employed during re-engineering processes to recover design information from the legacy application and utilise this data to modify or reconstitute the system, thereby improving its overall quality (Pressman, 1997). Reverse engineering is the opposite process – a working legacy system is analysed in order to create an abstract design. This is done to get a fundamental understanding of the structure of the legacy application and how to modify it without having suitable documentation (Ghezzi et al., 2003). Before reverse engineering work can be undertaken, so called "dirty source code" needs to be restructured to turn it into clean and easier to read source code which then can then be abstracted (Pressman, 1997). Extra abstraction is the core activity of reverse engineering, which includes extracting meaningful specification of the performed processing, the user interface, and the database. The final specification is completed after more refining and simplifying work. Modernisation is an umbrella term describing the process needed to be undertaken when maintenance of a system is no more appropriate to meet the new demands and a more thorough rewrite is required (Bakar et al., 2019). All of these processes are used to remove technical debt.

2.4. Risks in software re-engineering

The question now to ask is if there are so many strong drivers to modernise legacy system why are re-engineering processes so often abandoned (Fanelli et al., 2016)? The reason for the reluctance to upgrade software can be summarised in one word: risk. Rashid et al. (2013) identify six categories of risks which can occur during a re-engineering process according to their frequency mentioned in papers. These are: user satisfaction, cost, forward engineering, reverse engineering, performance, and maintenance. *User satisfaction* is essential for every company, and hence reliable measurements need to be taken as re-engineering risks introducing unexpected behaviours to the target system, potentially impacting user friendliness. Upgrading legacy software is often carried out in order to create a more *cost*-efficient system, but risks in cost benefit can include high maintenance cost after re-engineering, and poor-quality processes for re-engineering and inconsistency of business plans. The process of moving from high-level abstractions to the physical implementation in *forward engineering* risks difficulties in migrating existing data to the new system or in objects not being able to be integrated to the new system. The counterpart of forward engineering is called *reverse engineering* and its risks include the difficulty of capturing requirements and efficient design from source code, and not capturing all existing business knowledge embedded in source code. One of the key factors of the re-engineering process is to improve the *performance* of the system. If the performance is poorer than at the start, one of the basic goals has failed. Risks here include non-portability of the new system. The last category, improving *maintenance,* is one of the key goals of reengineering. The risks here include re-documentation and data restructuring, as well as recovery of legacy systems. Khadka et al. (2014) also asked participants about the constraints of modernising legacy systems. The most challenging constraints mentioned were time constraint for finishing the work, quickly followed by the availability of funding. Other significant challenges were predicting ROI (Return of Investment), data migration, lack of knowledge, and adequate testing. Additionally, resistance from inside the development team can occur as staff may fear their specialist knowledge will become redundant once the legacy system has been modernised. Practitioners also reported: "… they [staff] see legacy systems as their baby… they might not share their knowledge."

2.5. Risk management in software engineering

This leaves us with the question how can we mitigate these risks? The main challenge for software re-engineering is to make improvements whilst simultaneously mitigating risks and keeping the legacy application up and running. Such a process can be more risky than new development as the operational integrity of the application needs to be maintained throughout (Rashid et al., 2013). Thus, it is crucial that risk assessment, risk analysis and management, and risk mitigation are regularly and systematically undertaken during the re-engineering of a legacy application.

Boehm (1991) divides risk management into two main steps: risk assessment and risk control. Both of these main steps have three subsidiary steps. The first sub-step of risk assessment is the risk identification which produces a list of potential risks for the project. Techniques to identify risks include checklists, examination of decision drivers, or decomposition. The following step, risk analysis, deals with the assessment of the loss probability and magnitude of each identified risk. Performance model, quality-factor analysis, or cost models are typical techniques. The final task of the risk assessment step is the risk prioritisation, which ranks the identified and analysed risks. Preferred techniques include risk exposure analysis, risk reduction leverage, or group-consensus techniques. The first task of the second main step, risk control, is risk-management planning, which helps you to address each risk via e.g.: information buying, risk transfer, or risk avoidance. Common techniques include cost-benefit analysis, checklists of risk-resolution techniques, and standard-risk management outlines. The next step is risk resolution during which risks are being eliminate or mitigated by the use of e.g.: prototypes, benchmarks, or incremental development. The last step is risk monitoring deals with the tracking of the project's progress towards the resolving of the identified and handled risks. It also includes taking any corrective action if necessary. Milestone tracking and a top-ten risk-item list, which is highlighted regularly, are suggested techniques for this step.

To be able to fully implement risk management, Boehm (1991) suggests the use of risk-driven software-process models like the spiral model, which was also first described by Boehm. The spiral model has a cyclic outline, as opposed to the linear one of the waterfall model, in which each cycle consists of four stages (Ghezzi et al., 2003). The first stage sets the objective and evaluates alternatives and identifies any potential constraints. Phase two deals with the identification of risks, by using prototyping or simulations, and activities for mitigation are put into place. Moreover, the alternatives are evaluated. The product is developed and verified in stage three. The strategy the development follows is being dictated by risk analysis. During the final stage, the results of the stages traversed so far are being reviewed and the next iteration, if any, will be planned. The reason why the spiral model is recommended for implementing risk management into a project is because every sequence of life-cycle activities are determined by risk analysis (Boehm, 1991).

2.6. Proposed frameworks for risk management in re-engineering

A few authors have suggested frameworks for risk management during re-engineering. Clemons et al. (1995) suggested a framework for the identification and management of risks. Their approach is to support re-engineering as well as achieving strategic advantage by maintaining consistency between the internal needs of the organisation and the demands of the external environment (i.e., customers and other stakeholders). The authors also criticise that actions to reduce political risks are too conservative as they protect the project manager than supporting the re-engineering process. Furthermore, the authors make some suggestions based on experience with two Fortune 100 companies to reduce political and functionality risks. The avoidance of confrontation could move the espoused strategy closer to the more conservative strategy-in-use which may be fatal. They recommend the use of action science to uncover private assumptions and privately held strategies which could lead to informed discussions of different positions so that political risks can be assessed. However, they warn that in some cases discussions could also threaten the success of the project in form of unskilled interventions, but also forcing participants to abandon their defensive routines and taking the espoused position.

Rajavat and Tokekar (2011) propose a framework for decision driven risk engineering called ReeRisk. This theoretical framework serves to identify and eliminate risks in the early stages of the development cycle. The process starts by analysing the current state of legacy system and defining the requirements for the target system. The legacy system is used to develop a re-engineering strategy through which the target system will be developed. The risk engineering framework provides support for decisions made about the re-engineering process. The framework consists of three domains or aspects, which are system, managerial and technical. Risks are identified and measured for each domain. The system domain is responsible for maintaining the product and service to its customers. This domain holds two types of models: the infrastructure perspective model and the stakeholder perspective model. The managerial domain deals with the economic side of the project, which includes costs of the target system and the impact of market factors. It includes the economic perspective model and the business perspective model. The technical domain is concerned with the software functionality and quality including its technology. The domain consists of the quality perspective model and the functional perspective model.

2.7. Re-engineering approaches and Agile in re-engineering

Necessary for a successful re-engineering process is also the chosen approach. The classic approaches are “The Big Bang” approach, which replaces the whole system all at once, the “Phase-out” approach, which adds new software incrementally, and the “Evolutionary” approach, which is similar to the “Phase-out” approach, but where sections are chosen by their functionality, instead of their structure (Rosenberg & Hyatt, 1996). Fanelli et al. (2016) describe two other major categories, which are the “Database first”, and “Database last” approaches. These two can be categorised into the “Evolutionary” approach. While the “Database first” approach starts by migrating the data to a modern database, “Database last” does it at the end. An approach less associated with software re-engineering is Agile as a lack of academic papers shows. Some people would even argue that “re-engineering in an Agile point of view just can be refactoring” as no user stories are faced (Diana, 2010). However, every project has its stakeholders, and so its requirements from which user stories can be derived. Moreover, Agile, especially Scrum, is designed to deliver incremental additions while software is in use as Scrum emphasises a working product (fully integrated and tested) at the end of every Sprint. Additionally, Agile methodologies are based on the idea that many events cannot be anticipated beforehand, and therefore it is best to plan tasks in a flexible way as the project progresses (Masood & Ali, 2014). As one of the risks of software re-engineering is the unpredictability of tasks and challenges, Agile approaches can be a real benefit to upgrading legacy systems.

Singh et al. (2019) proposed a framework, which is supported by a case study, using Agile methodology as the flexible methodology fits nicely to the requirements of a re-engineering process. Their framework follows the standard pattern of a Scrum lifecycle. It starts of by ensuring the release plan including the iteration plan and the estimation of cost. All requirements are put in the product backlog. The requirements which will be implemented in the upcoming Sprint are put into the Sprint Backlog involving the whole Scrum team and the stakeholders. This also includes the estimation of velocity and the Sprint cost estimation. To estimate the size of user stories, story points are used. Each three-week sprint cycle is designed to accommodate forward engineering, code alteration and reverse engineering. After every sprint, retrospective actions are performed to confirm the correct implementation of the user stories. At the end of the lifecycle the application is integrated into the system by a big bang approach.

2.8. Gaps of previous research

However, all the mentioned frameworks ((Clemons et al., 1995), (Rajavat & Tokekar, 2011), and (Singh et al., 2019)) fail to describe risks that could occur during a re-engineering project or their impact, which would help project teams to prioritise their tasks and put mitigation in place before some risks might occur. Clemons et al. (1995) did describe some risk categories, however comparing them to Fanelli et al. (2016) risk categories, shows that there is barely any overlap between the two, so clearly some risk categories are missing in the framework. Moreover, the paper fails to specify any risks. Rajavat & Tokekar (2011) framework rather focuses on the decisions which need to be made during a re-engineering project instead of the risks which can appear.

Only one of the frameworks, which is the one from Singh et al. (2019), suggests a methodology for the re-engineering process. However, they only tested their framework over a single Sprint while reducing a set of code complexity.

This dissertation is based on an in-depth empirical study of a single re-engineering project involving a legacy application which had to be kept operational throughout. From this study I have developed a risk framework that will help practitioners with the identification and classification of potential risks and ways in which they can be mitigated. I have also assessed how helpful Scrum practices are during such a process.

3. Methods

To answer the research questions about which risks emerged during a software re-engineering project, how they got mitigated, and how Scrum supported this undertaking, an interpretive case study approach was taken (Walsham, 1995). This was based on the initial phase of a re-engineering project (see Case Study).

3.1. The justification of using a case study and the comparison to action research

It might not be obvious in the first place why a case study was chosen over an action research study as I was directly involved in the project. Let us compare these two research approaches first. A case study can be described as a method in which a phenomenon is investigated within its real-world context (Yin, 2018). The researcher wants to get an in-depth understanding of the phenomena (Dobson, 1999). A case study can be approached in two different ways: positivist, and interpretivist. Walsham (1995) describes the positivist approach in contrast to the interpretive one. Positivism is an approach that believes that facts, and only facts, represent science. In contrast, the interpretive approach states that facts and values are intertwined, and both are involved in scientific knowledge. The interpretive researcher could also take the position that "scientific knowledge is ideological and inevitably conducive to particular sets of social ends". Walsham describes interpretive research in this field as an ontology with regard to the human interpretation of computer systems. It is supported by either the theory of 'internal realism', which views reality as an intersubjective construction or 'subjective idealism' where each person has its own idea of reality. In contrast an action research study is an empirical research method which includes the researcher's active involvement in the project (Dresch et al., 2015). Action research uses an iterative pattern following two distinct stages: first diagnosing, which is about understanding the problem, and second a therapeutic stage, which addresses the problem. The focus is on the resolution and better understanding of a problem, but also the "improvement of a practice over a period of time".

The general idea of case studies is that they involve a clear division between the researchers work and the activity that is being researched (Petersen et al., 2014). However, Walsham (1995) states that most researchers influence the work of those people who are being researched, even if the researcher just shares a concept with them. Furthermore, the researcher holds two distinct roles during a case study: the one of being the outside observer, and the one of being the involved researcher. None of these roles can be considered as objective from an interpretive perspective given that the researcher analyses the data in their own subjective way. The involvement still could be considered as a threat to the validity as the researcher could get too involved in the project and lose sight of the reason for the research (Dawson, 2005). On the other hand, this approach has strengths as the researcher is deeply involved at the source of the data and has a good understanding of the issues faced during the software engineering process. A point of criticism, mentioned by Yin (1989), is how a single case study can be generalised. Walsham (1995) introduces four different types of generalisations supported by examples from information system case studies. The first type is the development of a concept as a single concept can be part of a broader network or a fusion of multiple concepts. The next type is the generation of a theory by constructing a theoretical framework. Another suggested way of generalisation is drawing of specific implications in which a tendency can be described. Last but not least, the contribution of rich insight describes the generalisation which readers gain from studying a report and result from an interpretive case study.

There were several reasons why a case study rather than an action research methodology was chosen even though I was directly involved in the re-engineering project. The most important was that no research information or suggestions for changing practice were fed back to the development team during this re-engineering project which is what would have happened if it had been run as an action research study. Also, the re-engineering work started before the research so the actual analysis happened retrospectively. Moreover, I wanted to immerse myself in the re-engineering work completely to gain an in-depth understanding of the phenomena without concurrently learning how to do action research. Finally, the team was very small, had little experience of re-engineering, and an external consulting company were involved so it would have been very difficult to run it as action research project.

3.2. Application of case study in this project

Yin (2003) describes the typical five stages of a case study: design, preparation, data collection, analysis and reporting. The first phase is the design of the case study in which objectives are defined, in this case it was the re-engineering process of a legacy application. The second phase is about the preparation for data collection, which includes the definition of the procedure. The data came from different sources. One of the most important forms of my data were the tasks in form of user stories, which were tracked on a Kanban board. The Kanban board was held online, which made it possible for everyone involved in the development process having access to it. The reason why it was so important for my dataset is that it made it traceable when which tasks had been carried out, when which user stories had been added as challenges occurred and risks needed to be handled or mitigated, or if tasks were delayed. It was not enough to know which tasks had been carried out when, but it was important to find out why. The meeting notes and the reflective field journal were the answers to this question. The meeting notes documented the Stand-Ups, Sprint Reviews, and Customer Demos. The most important decisions were made during these meetings. The field journal consisted of notes that appeared important about the code or decisions that have been made. A smaller, and not so important data source was the code basis of the legacy application. It was held in a repository on GitHub, where multiple people had access to. With a repository it can be ensured, that code changes can be undone, and to get an overview of the evolution of the code. Furthermore, bugs can be reported there. The code was important as it showed how the application was formed during the modernisation process. Collecting the evidence, which was the third phase, was done by going through the meeting notes, and the Kanban board, and looked which risks when occurred, and what or if anything was done to mitigate them. But also, the repository was studied to see what was done with the code and when certain risks were tackled.

3.3. Thematic analysis and its application in this project

The fourth phase was the data analysis, which in this case used a Thematic analysis (TA) approach to see how risks were handled and mitigated. TA can be applied when data needs to be interpreted, coded, and categorised (Alhojailan, 2012), or in other words to identify and interpret recurrent themes or patterns in data (Clarke et al., 2015). This project used an inductive, and reflexive TA. It was inductive since no pre-existing codebook was used, and the analysis was based primarily on the collected data. A pure induction was not possible, as the analysis had already been shaped by assumptions and prior knowledge of the researcher. To get a better understanding the reflexive aspect of TA, we need to understand Braun and Clarke's (2021) view of qualitative research which is also referred as “Big Q”. It rejects the idea of producing a general meaning as it is understood to be always tied to the context in which it appears. Qualitative research focuses on situated meaning, instead of searching for the truth or a theory as the quantitative research does. As the qualitative research accepts the fact that the truth is situated, or even multiple truths exist, the contribution is seen as a part of a rich tapestry, instead of searching for a singular truth and stepping towards to a complete, and perfect understanding. The researcher is seen as an interpreter of meaning and embraces their position by being seen as a subjective storyteller, instead of as an objective observer who is frightened of being biased and invalidating their research. The purpose of the data, which is rather text and meaning instead of numbers, is not to gain a generalised understanding, but an in-depth perception.

Braun and Clarke (2021) describe the analytic process which applies the reflexive TA method to the work in six phases: (1) dataset familiarisation, (2) data coding, (3) initial theme generation, (4) theme development and review, (5) theme refining, defining and naming, and (6) writing up. The familiarisation happens by re-reading the data until the researcher is deeply and intimately familiar with the dataset. The dataset in this case was a list of all the risks found in the notes, the Kanban board, and the repository. During the data coding, code labels are put on segments of the data. The coding is aimed to put a single meaning or concept on these segments. The level of coding can reach from surface meaning (semantic) to an implicit meaning (latent). The code is not just used to generalise the data, but also to capture the analytic take. In this research project, the codes put on the data reflect the risks which occurred during the project. Some of these data segments have the same code as the same risk arose multiple times. The coding is semantic and tries to state the risks explicitly. Some risks (codes) were difficult to identify as they were covered by the problems and work, they created. Moreover, it needs to be mentioned that these codes are my subjective interpretation of what the risks are. In the third step, initial theme generation, the researcher tries to identify any patterns across the dataset. By clustering codes meaningful answers to the research question might appear. The themes will not just appear, rather the researcher constructs them based on their research question, the data, the researcher's knowledge, and insights. Themes are trying to catch a broader, shared meanings. Once potential themes are identified, coded data are collated to candidate theme. In this case, the themes and sub-themes show in which situation the risks appeared. It puts the different risks into categories to see if a pattern can be found between different risk categories, e.g.: Did specific risk categories emerge during a specific time? How severe was the impact of certain risks through the lifetime of the project? Have all risks been mitigated? If not, can a pattern be found in the different types of risks? The candidate themes need to be reviewed in the fourth step to ensure it fits to the data set. The researcher needs to go back to the full dataset, and ask themselves if each theme tells a compelling story or does it highlight the most important patterns? Radical revision is a normal process during this step, which can lead to the collapse of whole themes. The relationship between candidates also needs to be considered. In this project the sub-themes changed a few times until they reflected the different types of risk in the best possible way without having themes with too many or too few risks. A result of this was that some risks (codes) appear in multiple themes as they fit into multiple risk types. The fifth step deals with giving every theme a concise name and a brief synopsis. Analysis still can happen during this step if noticed that a theme needs more development. The name of the themes and sub-themes reflects the situation in which the risk occurred. The last step, writing up, sounds misleading, as the writing should already have started around step 3 in the form of notes. The more notes generated during the steps, the more it can feed into the formal writing process. The aim of this step is to write a persuasive narrative about the researcher's dataset that addresses their research question. The results of the analysis are discussed in chapter 5.

The final phase of the case study deals with the reporting of the data, which talks about the basis of a risk framework and how Scrum/Agile aspects supported the re-engineering project and the mitigation of its risks.

3.4. Researcher's involvement during the re-engineering process

As mentioned in the beginning of this chapter, I, the researcher, was involved in the re-engineering process of the analysed legacy application. The small core team of developers working on a day-to-day basis, also formed part of a larger Scrum team, which involved internal stakeholders as well as external consultants. The full Scrum team is described in more detail later in Chapter 6.2. The core development team, which was the context in which the research was conducted, was formed as follows: I, a female junior developer, completed the main re-engineering work. It might also be important to mention that although a competent C++ developer, I had no previous experience with the MFC library which was a major part of the legacy system. The remainder of the development work was shared with two male senior developers of whom one is the original developer of this application, therefore, the most important contribution made by him was his intimate knowledge of the legacy application. The other senior developer has extensive knowledge of (deprecated) C++, so his technical support was essential for the success of the re-engineering work undertaken.

The external consultant company was not involved any development activity, although they participated in some stand-up meetings and most other scrum events in a support role.

4. Case Study

This case study is about the first phase of the re-engineering process of a legacy application. The two goals for the first phase were to make the application commercially viable again, and to turn it into a multi-developer application, while keeping it functional, both of which were achieved. My contribution, which was mainly from the development perspective, started with phase one during which I was the main software engineer. Some initial re-engineering had been done before my contribution started, which included the upgrade to a modern Integrated development environment (IDE). This work is categorised as Sprint 0 in the following timelines. The software was initially built in 1998, it was audited by an external company January - February 2021, Sprint 0 ran March – August 2021, and Phase one ran September 2021 – August 2022. Phase two began in January 2023.

4.1. The Application

The development of the application started 25 years ago. The system was written in non-standard C++ and was built by a single developer for a research project. Over time it was constantly expanded, not just for additional functionality, but also for research purposes. Due to the absence of an architecture, and the constant extensions, the code quality worsened over time and the code base became very messy as it included numerous redundant elements. All of this contributed to the accumulating technical debt, on top of which there was a constant need for quick bug fixes.

Several versions of this system exist. A demo variant is used for sales and other demonstration purposes. Another version is used for training. Within the main product, a version for more experienced users exists. Two different solutions of the system are provided: a desktop version using MS Windows native user-interface, and a browser-based cloud version using a WordPress front-end.

4.2. The Technical Debt Audit

Before the re-engineering started, the application was audited by an external consultant company. The audit was carried out to evaluate the status of the system and covered a technical review of the software source code and documentation, but it also included suggestions to improve the quality of the code. The audit was released in February 2021. The two dimensions of the audit, which are pertinent for the re-engineering are comments about the software itself, and the process. Comments on the software were categorised into six factors where technical debt were found: coding standards, testing, build and deploy, architecture and system design, collaboration, and technology stack and infrastructure. The important process dimension was the software development process.

4.3. The Re-engineering Process

The following sub-chapter describes the technical debt identified by the consulting company in their audit, and how it was resolved if it was tackled during the first phase. The sub-chapters are structured as the categories in the audit.

The decisions of which work needed to be done in Phase 1 were based on the two goals for the phase: making the application commercially viable again and turning the system into a multi-developer software.

4.3.1. Software

4.3.1.1. Coding Standard

Technical Debt	Solution
Use of god classes/functions	• Redundant code has been removed • Large files have been broken into smaller ones, each with a single purpose • Deleted unnecessary class
Inconsistent capitalisation of file and folder names	• Files and folders have been renamed to achieve consistency in Camel Case

Hard coded sensitive data	• Passwords in plain text have been removed from the code and encrypted and saved in a file
Absolute file paths	• Absolute file paths turned into relative file paths
Use of magic numbers	Will be done in Phase 2
Large conditional statements	Will be done in Phase 2
Hard coded database names	Will be done in Phase 2
Hard coded user data	Will be done in Phase 2

Table 1: Technical Debt - Coding Standard

Four elements of technical debt were identified as needing to be removed in phase 1. The first was the god classes/functions. A “God Class” is a single class with multiple purposes. One of these classes had 35,000 lines of code. The principle of “God Functions” is the same, but on a function/method level. The course of action to reduce this technical debt was to break up these classes into multiple classes, each with a single purpose. The second technical debt elements that needed removing in Phase 1 was the inconsistent capitalisation of file and folder names. These were renamed to achieve consistency in camel case. The third element of technical debt, which posed a security gap, was hard coded sensitive data in form of passwords. To close this gap, passwords in plain text were removed, encrypted and saved in a file. The final element of technical debt to be removed in Phase 1, which was necessary to turn the system into a multi-developer application, was absolute file paths. These were turned into relative file paths. The timeline of the re-engineering work regarding the coding standard can be seen in Figure 3.

The remaining technical debt will be removed in Phase 2, as it was not necessary for either turning it into a multiple-developer application, nor for re-commercialisation.

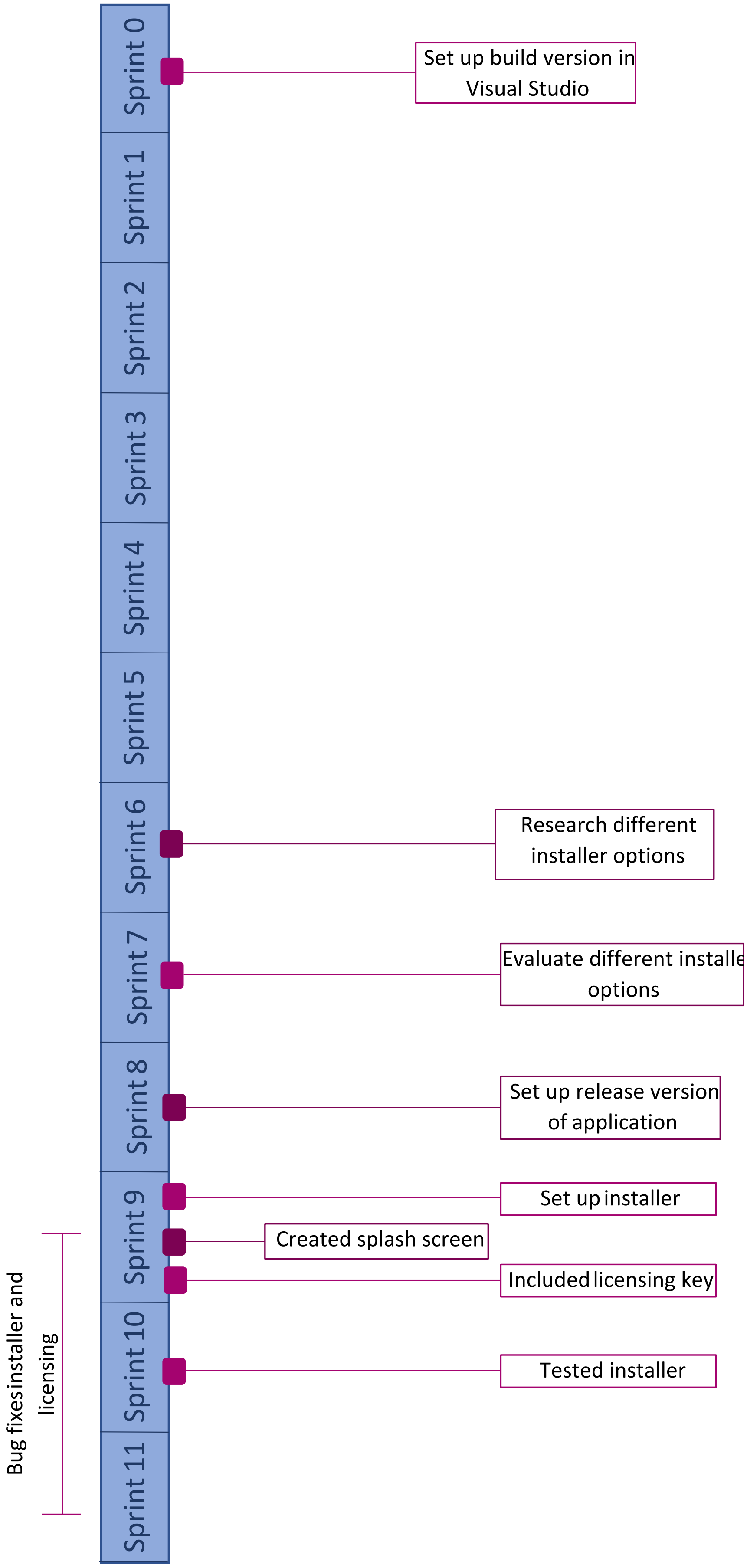
Sprint 0
Sprint 1
Sprint 2
Sprint 3
Sprint 4
Sprint 5
Sprint 6
Sprint 7
Sprint 8
Sprint 9
Sprint 10
Sprint 11
Set up build version in Visual Studio
Research different installer options
Evaluate different installe options
Set up release version of application
Set up installer
Created splash screen
Included licensing key
Tested installer
Bug fixes installer and licensing

Figure 3: Timeline of re-engineering work regarding the aspect of coding standard

4.3.1.2. Testing

Technical Debt	Solution
Manual user acceptance tests	• Detailed, documented user acceptance have been created • Automatic user acceptance tests have been created
No unit tests	Will be done in Phase 2
No integration tests	Will be done in Phase 2

Table 2: Technical Debt – Testing

The technical debt which needed to be removed in Phase 1 regarding the testing was that the manual user acceptance tests needed to be replaced with automated ones, in order to speed up testing. For this purpose, detailed, documented user acceptance tests were created, and a menu item to run the automated test was added. The timeline of the re-engineering work regarding the testing of the application can be seen in Figure 4.

Unit and integration tests are important indeed, but they were not essential for the goal of Phase 1, and therefore they will be added in Phase 2.

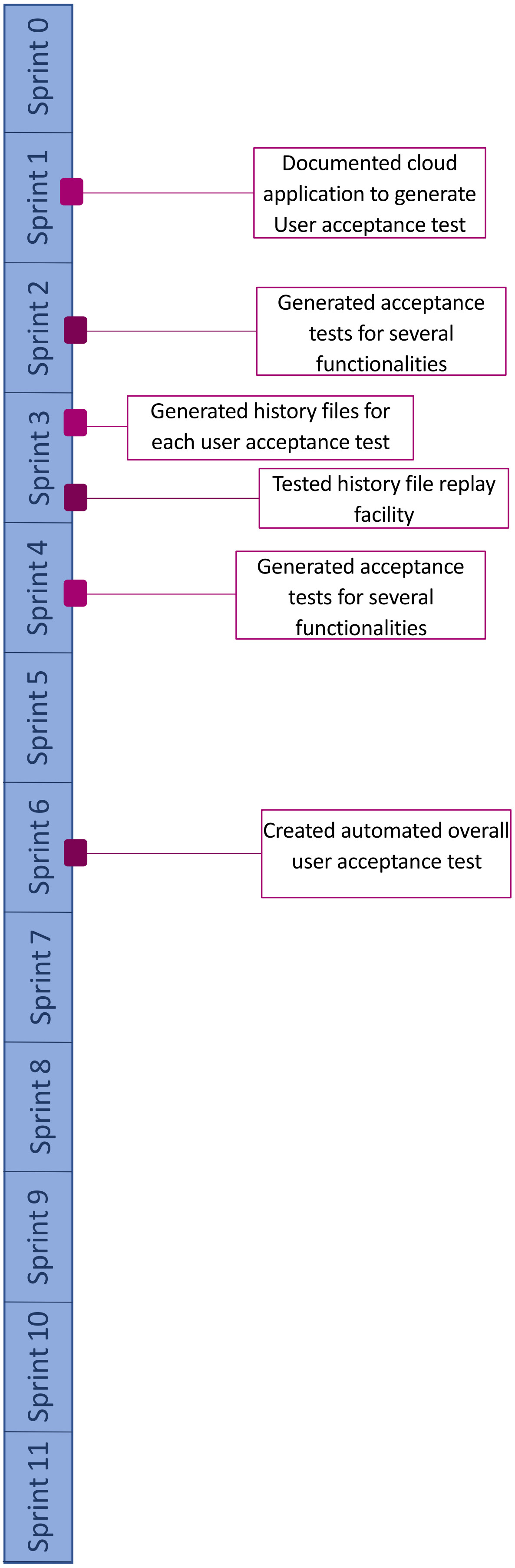
Sprint 0
Sprint 1
Sprint 2
Sprint 3
Sprint 4
Sprint 5
Sprint 6
Sprint 7
Sprint 8
Sprint 9
Sprint 10
Sprint 11
Documented cloud application to generate User acceptance test
Generated acceptance tests for several functionalities
Generated history files for each user acceptance test
Tested history file replay facility
Generated acceptance tests for several functionalities
Created automated overall user acceptance test

Figure 4: Timeline of re-engineering work regarding the aspect of testing

4.3.1.3. Build and Deploy

Technical Debt	Solution
No automated build tools	• Visual Studio is used as the build tool
No deployment tool	• Set up release version of application • Installer has been created as deployment tool • Added security key to installer

Table 3: Technical Debt - Build and Deploy

An important factor of a modern application is to have automated build tools, which saves the project settings and build instructions along with the code. Visual Studio was chosen and used as the build tool to remove this technical debt. Second, a major criterion for releasing an application to market is a deployment tool. The first step to achieve this was to set up a release version of the application, then to create an installer as a deployment tool. The installer also included a new splash screen, the C++ Redistributables – to enable offline installations -, and a prompt to ask the user for a license key. The timeline of the re-engineering work regarding building and deploying the application can be seen in Figure 5.

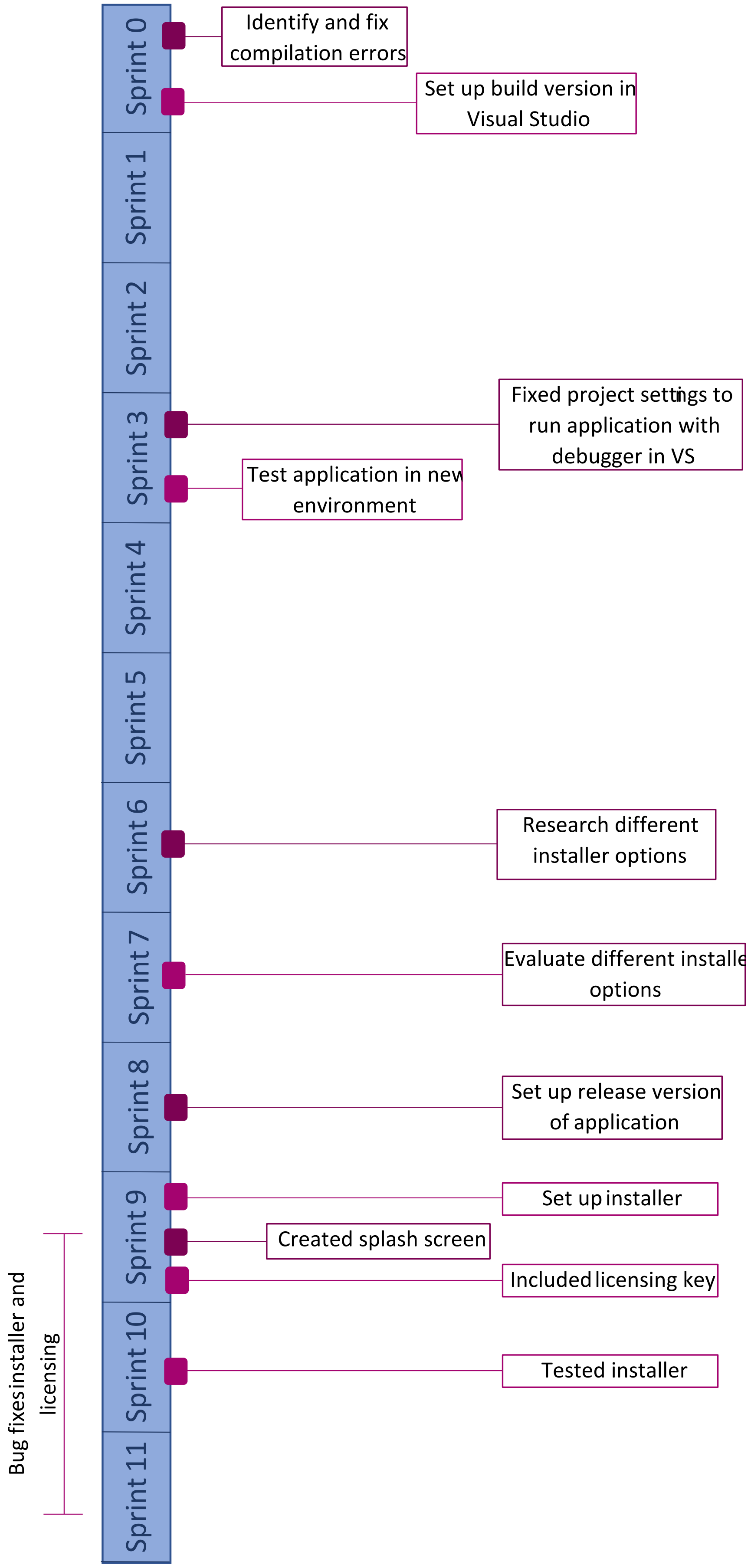
Sprint 0
Sprint 1
Sprint 2
Sprint 3
Sprint 4
Sprint 5
Sprint 6
Sprint 7
Sprint 8
Sprint 9
Sprint 10
Sprint 11
Identify and fix compilation errors
Set up build version in Visual Studio
Fixed project settings to run application with debugger in VS
Test application in new environment
Research different installer options
Evaluate different installe options
Set up release version of application
Set up installer
Created splash screen
Included licensing key
Tested installer
Bug fixes installer and licensing

Figure 5: Timeline of re-engineering work regarding the aspect of building and deployment

4.3.1.4. Architecture and System Design

Technical Debt	Solution
No architectural style	Will be done in Phase 2
Single directory for source code	- Files and folders have been structured based on functionality, and named based on naming conventions
Same development approach for cloud and desktop version	Will be done in Phase 2

Table 4: Technical Debt - Architecture and System Design

The architecture and system design technical debt that needed removing were related to the goal of moving towards a multi-developer system. The source code was bundled in a single directory, and no project structure was given. The files and folders were structured based on their functionality, and the folders named based on naming conventions. This makes it easier for new developers to understand the structure of the project and can also be considered as the first step towards creating an architecture. The timeline of the re-engineering work regarding the architecture and design can be seen in Figure 6.

An MVC (Model-View-Controller) architecture will be implemented in Phase 2, and the application will be broken up in three distinct parts: a standalone C++ library, a desktop application, and a client-server application.

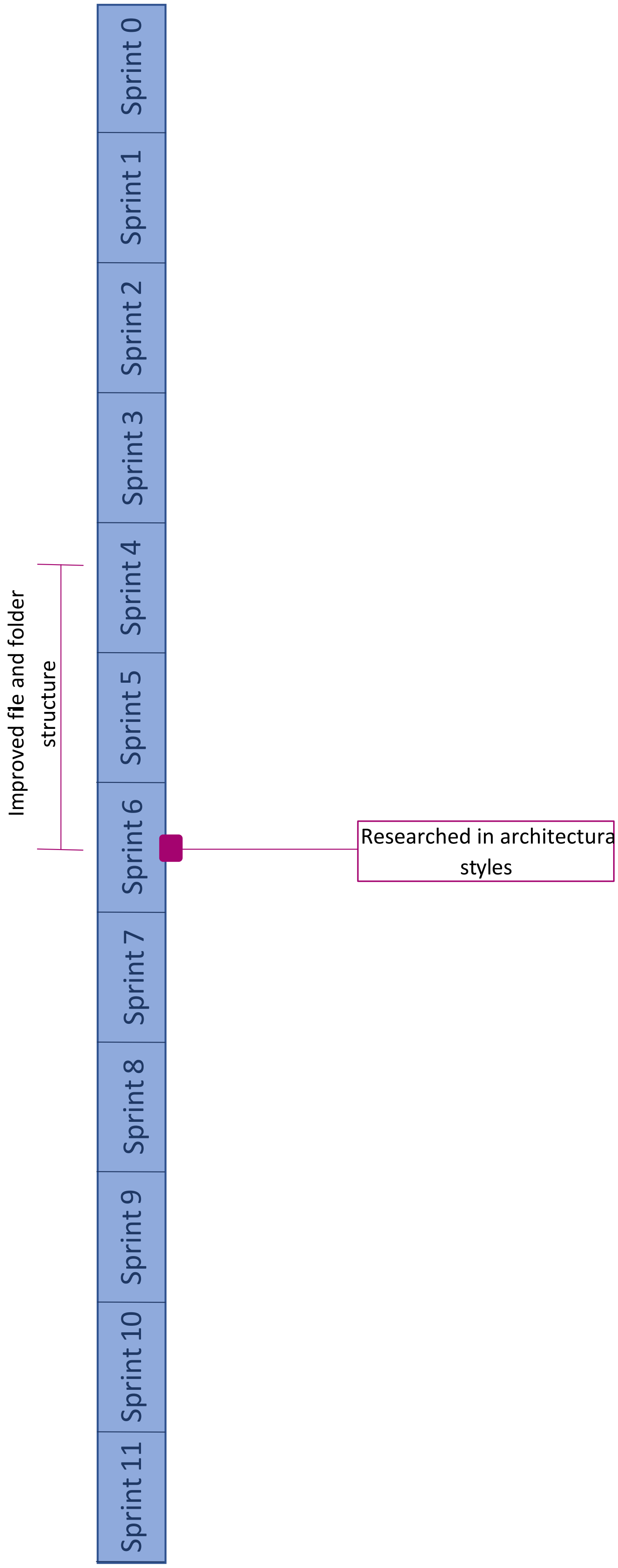

Figure 6: Timeline of re-engineering work regarding the aspect of architecture and system design

4.3.1.5. Collaboration

Technical Debt	Solution
Only ad hoc collaborations with another team within the university	• A second software developer has been employed and trained
Some use of communication tools	• GitHub has been set up for version control, distribution, and issue tracking

Table 5: Technical Debt – Collaboration

The system was mainly developed by a single developer with some ad hoc collaboration with another team within the university. In order to help with the re-engineering work, I was employed and trained as a second developer. The training included running the system as a user two dozen times, along with creating a detailed set of user acceptance tests. A major step towards a multi-developer application was to set up a communication tool. GitHub was identified and set up as a tool for version control distribution and issue tracking. The timeline of the re-engineering work regarding the improvement of the ability to collaborate with other developers can be seen in Figure 7.

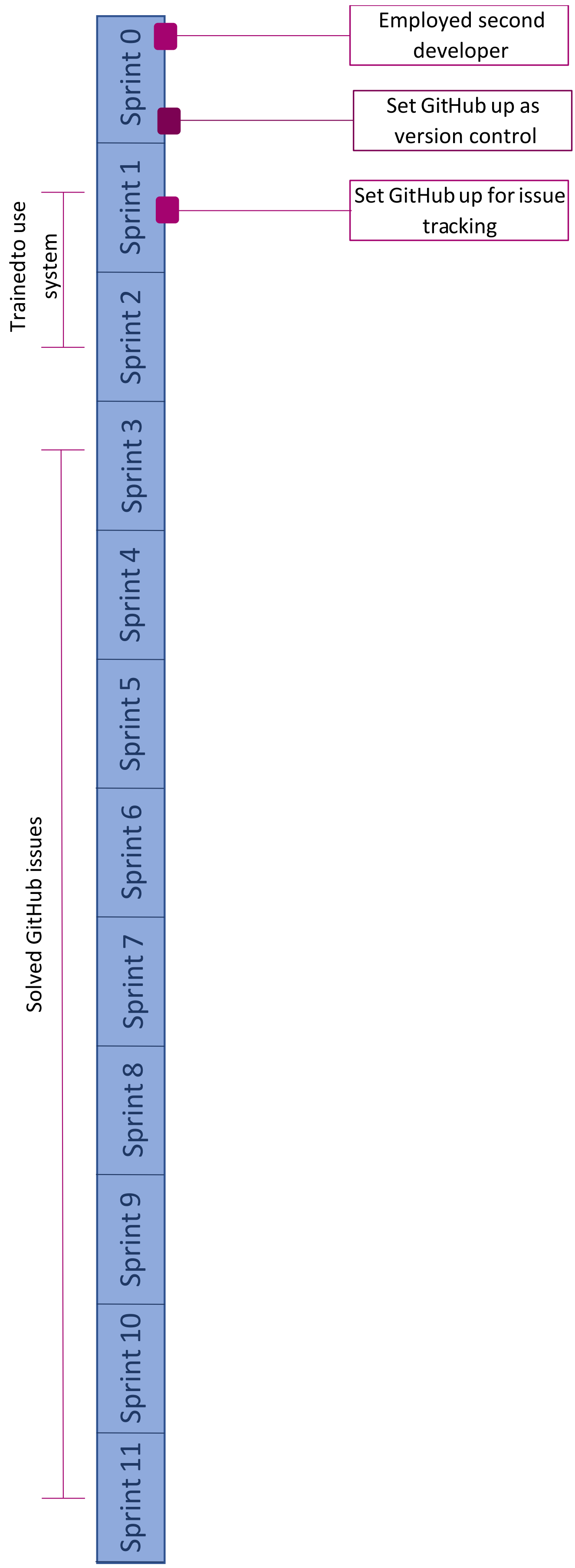

Figure 7: Timeline of re-engineering work regarding the aspects of collaborative working

4.3.2. Process

4.3.2.1. Technology Stack and Infrastructure

Technical Debt	Solution
Use of deprecated version of C++	- Upgraded to latest C++ ISO and MFC standards - Identified and fixed compilation errors - Ensured full functionality of system
Use of non-supported IDE	- Migrated system to modern IDE (Visual Studio 2019)
Use of external libraries, which are outdated or alarming	- Created an inventory of the third party libraries - Assessed external libraries - Put a façade on deprecated/unsafe libraries to enable fast exchange
No use of declarative UI	Will be done in Phase 2

Table 6: Technical Debt - Technology Stack and Infrastructure

As the technology of the system had not been updated since it was first created, a significant proportion of it was deprecated. The first thing to be done in Phase 0, before my involvement, was the migration of the system from Visual C++ 6 to a modern IDE, in this case Visual Studio 2019. During this process, the older version of C++ was upgraded to the latest ISO and MFC standards, and compilation errors were identified and fixed. When my involvement started, the first thing to do was to check the full functionality of the system after the migration. The bespoke user acceptance tests were the outcome of this procedure. A full inventory of the external libraries was created to ascertain which are currently supported, and which are a security risk. A façade was put on the deprecated and/or unsafe libraries to enable fast exchange.

It was also planned to re-compile Matlab files to DLLs with the newest compiler. However, the most recent compiler only compiles Matlab files to 64-bit, which is not compatible with the 32-bit system. The alternative would have been to use an older version of the Matlab compiler, which is still significantly more modern than the one the original DLLs were compiled with, but the newly complied DLLs were too different to make them compatible with the application. After it was checked that no security threats arose from the old DLLs, it was decided to leave the old DLLs and wait to compile 64-bit DLLs until Phase 2. The timeline of the re-engineering work regarding the technology stack and infrastructure can be seen in Figure 8.

An upgrade to the User Interface (UI) will be made in Phase 2 in form of a declarative UI. Furthermore, the application will be upgraded to a 64-bit system.

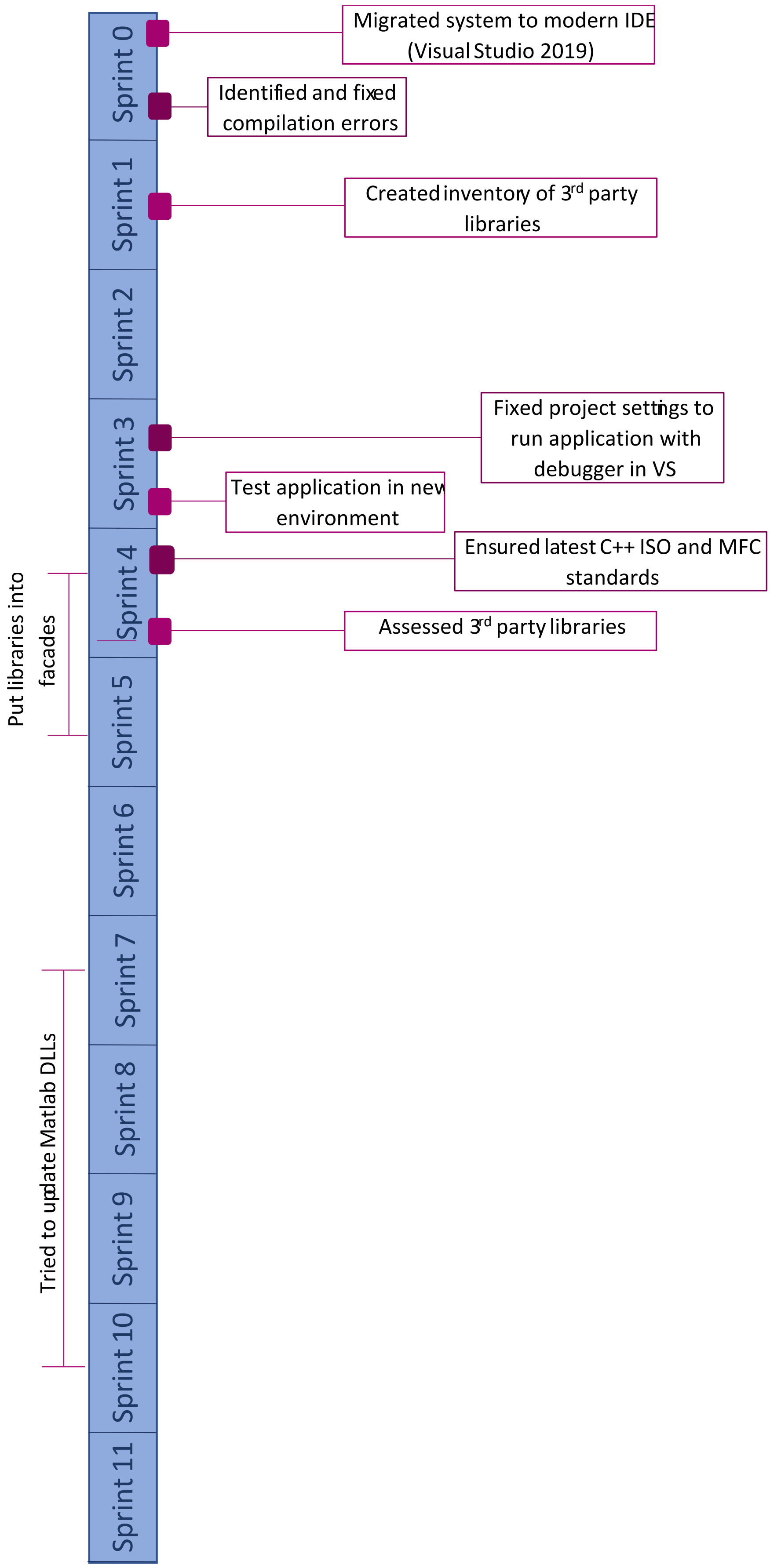
Sprint 0
Sprint 1
Sprint 2
Sprint 3
Sprint 4
Sprint 5
Sprint 6
Sprint 7
Sprint 8
Sprint 9
Sprint 10
Sprint 11
Migrated system to modern IDE (Visual Studio 2019)
Identified and fixed compilation errors
Created inventory of 3rd party libraries
Fixed project settings to run application with debugger in VS
Test application in new environment
Ensured latest C++ ISO and MFC standards
Assessed 3rd party libraries
Put libraries into facades
Tried to update Matlab DLLs

Figure 8: Timeline of re-engineering work regarding the aspect pf technology stack and infrastructure

4.3.2.2. Software Development Process

Technical Debt	Solution
No use of modern development process	• Implemented a Scrum process

Table 7: Technical Debt - Software Development Process

The audit noted that the implemented development approach was consistent, but that it was not a modern approach. It was decided to change the approach to a modern development process: Scrum. The way this Agile approach was applied to the re-engineering process was by working in Sprints, having Stand-Ups twice a week, and Sprint Reviews, Sprint Retrospective and Sprint Planning every three weeks, and undertaking a Customer Demo with the Stakeholders every six weeks. The timeline of the re-engineering work regarding the software development process can be seen in Figure 10.

Since using Scrum as the methodology of a re-engineering process is part of the research question, the process is explained in more detail later.

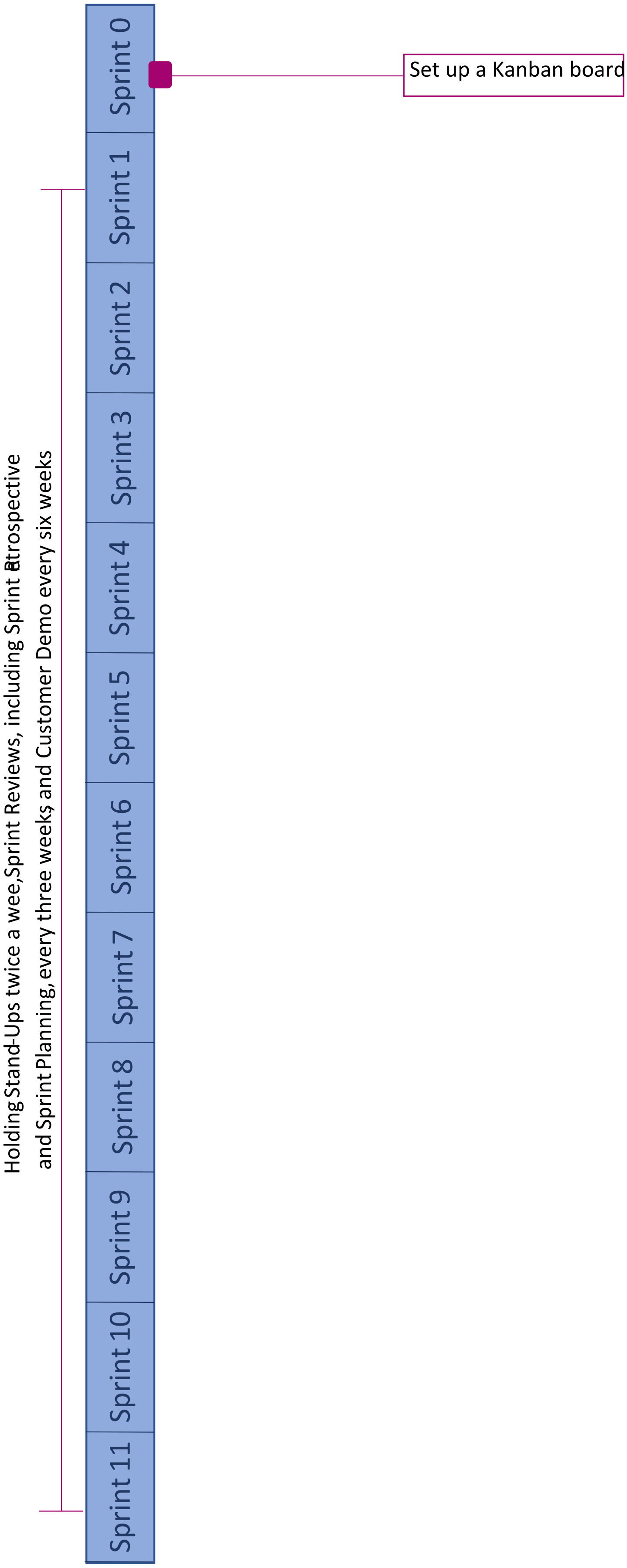
Sprint 0
Sprint 1
Sprint 2
Sprint 3
Sprint 4
Sprint 5
Sprint 6
Sprint 7
Sprint 8
Sprint 9
Sprint 10
Sprint 11
Set up a Kanban board
Holding Stand-Ups twice a wee, Sprint Reviews, including Sprint Retrospective
and Sprint Planning, every three weeks, and Customer Demo every six weeks

Figure 9: Timeline of re-engineering work regarding the aspect of software development process

5. Towards a Risk Framework

The aim of this study was to build the basis of a risk framework to support future re-engineering projects within Agile (Scrum) environments. The primary data consisted of a list of risk descriptions. Each risk description was augmented with a qualitative assessment of other attributes of the risk a) the potential impact of the risk on the project viability, b) the actual impact of the risk, and c) the mitigation status, which is the status of the risk after the first phase of the re-engineering project ended. The risk descriptions were analysed using inductive Thematic Analysis (TA) (Clarke et al., 2015), and the other attributes were used to aid a more comprehensive analysis of how the risks affected the re-engineering work.

5.1. The Analysis of Risks

5.1.1. The Dataset

The dataset consisted of a list of different risks that had occurred during the re-engineering process. The risks were collected post hoc from various sources. They were collected from the Kanban board by looking for tasks undertaken to mitigate risks. Other sources were meeting notes as well as the field journal which held information about decisions, including the mitigation of risks. Also, the source code gave good insights into certain risks and how they were tackled. Each of these risks was summarised in a single sentence.

5.1.2. Thematic Analysis: Coding, Sub-Themes and Themes

Following Braun & Clarke's TA approach (Braun & Clarke, 2021) the analysis started with coding. The codes slowly developed after reading through the data multiple times. Some of these codes appeared multiple times as the same type of risk had occurred multiple times during the project in slightly different situations. Other risk descriptions were allocated multiple codes as multiple potential risks occurred in this situation. During this process some code names were revised as sometimes the name initially given reflected the work it created or other problems rather than describing the risk itself. This was done because codes should only reflect a single meaning. The challenge was to not get into too much detail as the codes were used to capture a broader, more general picture, while still being explicit. This process resulted in 44 different codes. Codes could be linked to multiple risk descriptions, which means a risk occurred multiple times, but some codes were only linked to one risk description. A list of all codes with a description can be found in Appendix 1: Themes, sub-themes, and codes with explanation. To group together multiple codes and identify potential patterns, sub-themes and themes were created. The sub-themes categorise and group together similar codes to see if a pattern in the different codes can be found, but also to give these codes a more general meaning. Some codes appeared in multiple sub-themes as they fitted in multiple categories. The themes, in which the sub-themes were grouped into, reflect the different parts of the project. I was able to identify three distinct parts which form my themes: Technology, Process and People. The Technology theme is rather obvious as it contains all the risks related to the technical side of programming, and different technologies. Process consists of all the risks which were related to managing the re-engineering work. People holds all the risks which are caused by human failure. These themes hold different sub-themes are as follows:

Themes	Sub-themes	Codes
Technology	legacy technology	discontinued technology support
		struggle to integrate technologies
		legacy code
		no replacement for technologies
		lack of old technology for developing old software
		fundamental change of technology
		lack of knowledge about old technology

	insufficient technology	lack of testing environment
		missing version controlling
		inability to roll back code changes
		code over-heap for IDEs and support tools
	legacy application	lack of legacy application documentation
		lack of readability of code
		limited number of people with knowledge about legacy application
		lack of commentary in code
		fix is worse than problem
		lack of knowledge about legacy application code
Process	**testing**	missing test documentation
		lack of time for testing
		lack of automated testing
		lack of testing environment
		inconsistent testing
	time constraint	lack of time for testing
		limited licensing time
		lack of time for meetings
	lack of documentation	missing test documentation
		lack of commentary in code
		lack of legacy application documentation
		lack of re-engineering documentation
People	**lack of knowledge**	lack of knowledge about legacy application code
		lack of state-of-the-art technology knowledge
		limited number of people with knowledge about legacy application
		lack of knowledge about old technology
		lack of commentary in code
	process engagement	confused stakeholders
		not clarifying importance of task
		inconsistent issue tracking
		holding meetings incorrectly
		lack of time for meetings
		no clear goal
		inconsistent testing
		inconsistent bug tracking
		inconsistent use of version control
		unwillingness to change
	methodology engagement	misevaluation of story points
		too big tasks
		neglecting methodology
		unclear definition of done
		wrong prioritisation
		poorly written user stories
	social	unwillingness to change
		lack of face-to-face working

		lack of support
		lack of a team
		confused team members
		confused stakeholders
		not clarifying importance of task
		insecurity of team members

Table 8: All Codes, Sub-Themes, and Themes

It might not be obvious why the sub-theme **Process engagement** is located in the theme ***People*** and not in ***Process***. However, these risks were not caused by the process itself, but rather by the way people executed it. Taking the code “inconsistent use of version control” as an example, it can clearly be said that the use of version control is part of the process engagement, but failing to commit code regularly is a human issue. The sub-theme **Legacy technology** includes all the risks which are caused by old, out-dated technologies used in the application. **Insufficient technology** describes all the risks which limited or even derailed the execution of re-engineering tasks due to the restriction or lack of certain technologies. Every risk which dealt with the legacy application itself falls under the sub-category **Legacy application**. This also includes everything about the code, not to be confused with the programming language which falls under **Legacy technology**. The sub-theme **Testing** covers all risks which are caused by anything related to testing or the lack of it. **Time constraints** include the risks which arose by a lack of time. Risks which arose by missing documentation are covered in the **Lack of documentation** sub-theme. The **Lack of knowledge** sub-theme entails all risks which occurred because of ignorance. **Process engagement** covers all risks which fell under the housekeeping side of the project such as testing and holding meetings. This should not be confused with **Methodology engagement**, which is an own sub-theme, holding all risks related to Scrum in this project. The last sub-theme is **Social**, which includes all risks which arose from interpersonal relationships and the group dynamic of the team.

5.1.3. The Weighted Risk Codes

Two weightings were added to each risk code to indicate a) the potential impact it could have had, and b) the actual impact it had on the project. The potential impact stated here was not assessed before the start of the project, but retrospectively, at the same time as the rest of the analysis was conducted. The potential impact was reconstructed as accurately as possible by making reference to notes and the Sprint board. In this project the potential impact does not indicate the potential worst-case scenario, but rather a realistic assessment of the impact the risk code of the imagined point of view before the project started, which was challenging. The realistic assessment fits better for this analysis as it shows how the team perceived the risk. The actual impact describes how big the effect of the risk was as it happened.

The base risk graph can be found in chapter 5.2.1. Each risk has the value 1 before the impact for a weighting was added. Each risk can have different types impacts on the project. Stress, time, or money are the most frequent ones which occurred in this project. All these factors constitute a certain thread to the project. These weightings are relative, so the framework can be fitted to any re-engineering project. The weighting is divided into four levels: none, low, medium, and high. The explanation of each weighting for the project is:

- None: The risk did not occur, or it was decided to deal with the mitigation during a later point of time; hence no work or stress was caused.
- Low: The risk has been noticed but is rather a nuisance and led to some inconveniences.
- Medium: The risk caused moderate inconveniences.
- High: The risk was a potential showstopper. The work and/or stress it generated is vast.

Each weighting was assigned a logarithmic value in order to make the higher-impact risks stand out in the resultant graph. Risks with the weighting “none” received the value 0, with the weighting “low” the value 1, with the weighting “medium” 2, and with the weighting “high” they obtained the value 4. The reason for having only four weightings was to simplify the process of assigning them to risk codes. Consideration was given to using additional weightings, as some higher-rated risks posed a greater threat than others with the same weighting. However, given the size of this study, four weightings were considered sufficient.

A list of each code with its weighting can be found in Appendix 2: Codes with their expected and actual impact.

A third factor was also added to describe the status of mitigation of each risk code after the end of the first phase. The different statuses are:

- Not occurred: The risk did not occur at all, hence there was no mitigation to be done.
- Mitigated: The risk was mitigated successfully.
- Partially mitigated: The risk was mitigated, but not satisfactory or the solution was rather a temporary one.
- Not mitigated: The risk was not mitigated at all.
- Deferred: The risk was not mitigated; however, it will be tackled in the future.

A list of each code with its mitigation status can be found in Appendix 3: Codes with their mitigation status.

5.2. Results

5.2.1. The Calculation

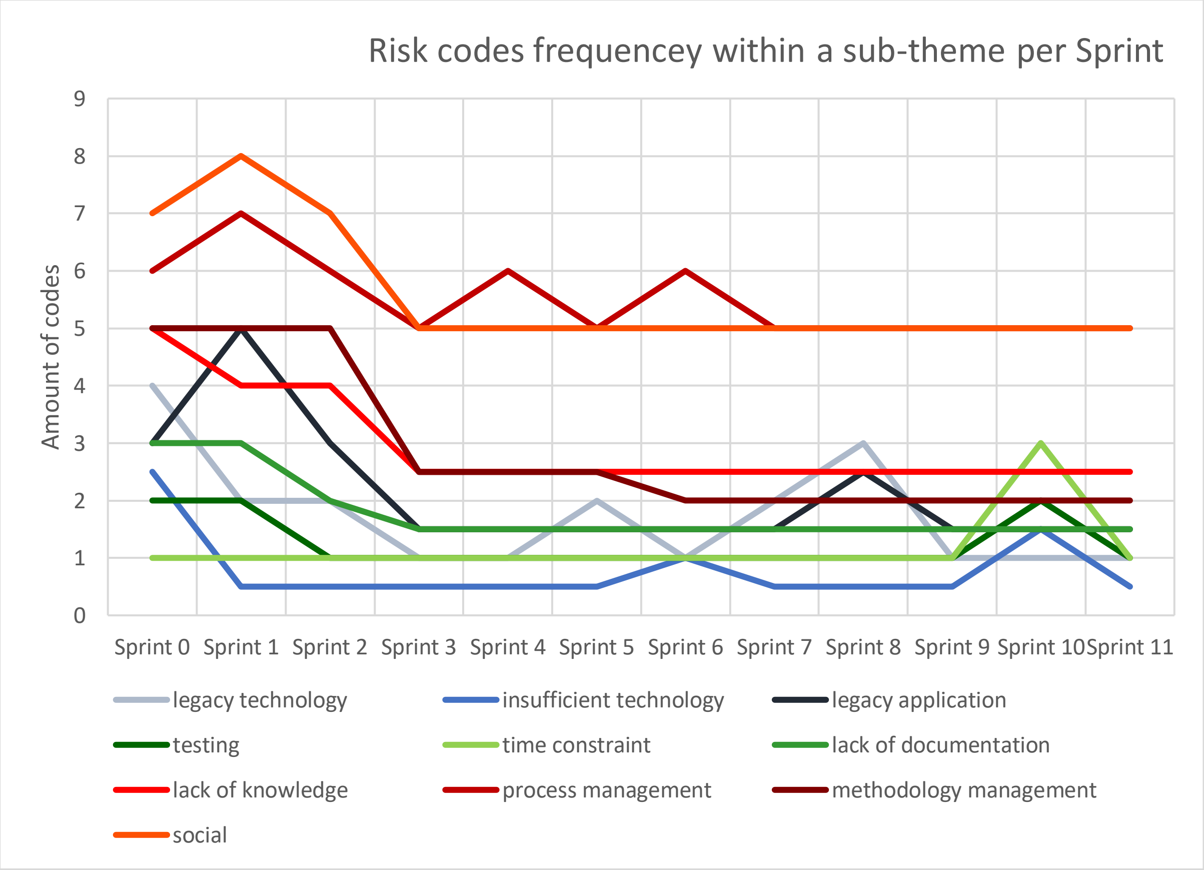

Figure 10: The risk codes frequency within a sub-theme per Sprint (without weightings)

The basis of the calculations was a table matrix with the Sprints on the x-axis and the risk code frequency within a sub-theme (sub-themes) on the y-axis. Each risk code had the value one, so no weightings were applied so far.

A risk code could appear in multiple Sprints or over a certain period like the beginning (Sprint 0-2), the middle (Sprint 4-8), or the end (Sprint 9-11). Another time span used to declare when a risk happened was the first half (Sprint 0-5), or the second half (Sprint 6-11). If a risk code only occurred partially during a timespan, the weighting of this code will be divided by 2.

To make it a bit clearer, I give a detailed explanation below of the risk codes in the **Legacy application** sub-theme (see Table 9). *Lack of legacy application documentation* appeared only in Sprint 1, which means its value was only shown in Sprint 1 on the graph. *Lack of readability of code* was a longer occurring risk. The occurrence description reads 'during the whole project, but especially in the beginning', which implies the risk was omnipresent during the whole project, but only relevant in the beginning. Therefore, the value of the risk code counts medium, with the value two in this case, in the first three sprints, but only half for the rest of the project. *Lack of knowledge about legacy application code* occurred in the middle of Phase 1, so the risk code value was added from Sprint 4-8.

Legacy application	Lack of legacy application documentation	Sprint 1
	Lack of readability of code	during the whole project, but especially in the beginning
	Limited number of people with knowledge about legacy application	during the whole project, but especially in the beginning
	Lack of commentary in code	during the whole project, but especially in the beginning
	Fix is worse than problem	Sprint 8
	Lack of knowledge about legacy application code	in the middle

Table 9: Risk code appearances in the sub-theme Legacy application

For the sake of clarity, all preparatory work for the re-engineering implementation phase was put into Sprint 0. The risks that appeared during the preparation were as much as a risk for the success of completing the re-engineering work as the risks that appeared during the implementation phase.

Even though the weightings were not included, (Figure 10) shows certain trends within the sub-themes. In the beginning the IDE and the compiler were changed, which raised the number of risk codes withing the **Legacy technology** and **Insufficient technology** sub-themes. It can also clearly be seen that it took some time to get used to Scrum (looking at the sub-theme Methodology engagement). Later, more process work was done, such as writing tests and in the later Sprints at the end, more technical work was done again, but also the time pressure increased towards the end. It is striking how important the ***People*** sub-themes **Social**, and **Process engagement** are, but this will be discussed further in the following chapters.

5.2.2. Weighted Impact of Risks per Sprint

To generate the weighted potential impact of the risk types, the frequency was multiplied by the n weighting mentioned in 5.1.3.

5.2.2.1. Weighted Impact of Technology Risks per Sprint

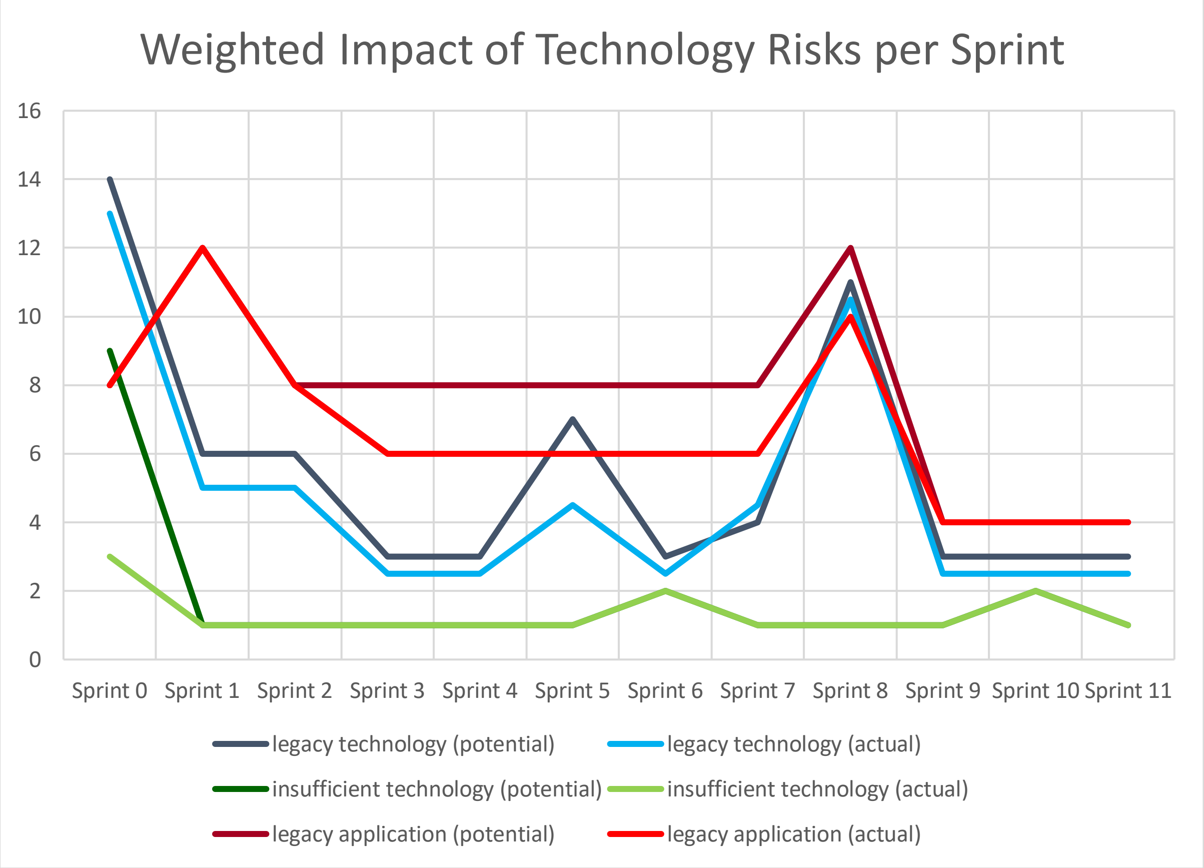

Figure 11: Weighted impact of technology risks per Sprint

The first striking thing comparing the weighted impact with the base graph is that even though **Legacy technology** has fewer risks than some other categories, its risks had the biggest impact at the beginning of project. The starting phase was crucial because the technology within the IDE was discontinued so there was a risk the software would not work on a newer OS, so it needed to be transferred into a new IDE. This risk included the difficulty of integrating the code and that it might not work as intended because changes were necessary to meet new language standards. Furthermore, there was a risk that such outdated code might be difficult to understand. As well as this risk code – *Legacy code* - peaked in the beginning, and continued through the rest of the project, the risk code *Lack of knowledge about old technology* did that too as it took some time to get used to the legacy technology used in the application. The risk code never disappeared as it is impossible to learn everything about a certain technology in such a short time. A medium-impact rated risk arose in Sprint 5 when an outdated technology could not be replaced – risk code *No replacement for technologies* – as certain deprecated libraries could not simply be replaced.

This was followed by *Lack of old technology,* which posed a high-impact risk, for developing old software in Sprint 7 as a 32-bit environment was required. The risk sub-theme of **Legacy technology** peaked again in Sprint 8 when the risk of *fundamental change of technology* triggered the risk of struggle to integrate technology. This was triggered by the DLL problem mentioned in 4.3.2.1.
It is noticeable that the graph for the expected impact and the actual impact follow the same patterns. This may reflect the fact that the expected and actual impact were assessed at the same time, but it still gives a realistic insight. However, the actual impact was almost always less than the predicted one. For example, *No replacement for technologies,* was rated as a high risk, but the actual impact was medium as it was decided to not replace them within this phase, but rather put a façade on each library to make the replacement easier later on, which still caused work, but far less than expected.

Another risk was L*ack of knowledge about old technology*. It was expected that team members might have problems using old technologies which are used within the legacy application, but because of the good support from other team members, the lack of knowledge was able to be compensated quickly. Even though most risks had less impact than expected, it can be seen that in Sprint 7, the lines crossed, indicating that there was a risk, which had more impact than expected. This was L*ack of old technology for developing old software* as instead of just getting an old laptop, a Virtual Machine needed to be set up, which caused some work.

Most of the risks regarding **insufficient technology** were eliminated within Sprint 0 as the risks were triggered by the lack of version control, which was added right in the beginning through the use of GitHub. The medium-impact risk code a *code overflow for IDEs and supporting* tools was omnipresent, which peaked in Sprint 6, as a static analysis tool could not comprehend some code files as they were around 35,000 lines long. Another risk with rather small impact risk occurred towards the end caused by a lack of a testing environment as a clean machine without pre-installed C++ re-distributable was needed. Both graphs, the expected and the actual impact, overlap for most of the sprints, except for Sprint 0. This was due to the risks *missing version control* and i*nability to roll back code changes* as work on the code was done before a version control was added. It was expected it would cause a lot of extra work as the code could not easily be shared among team members and changes could not be rolled back. However, none of that happened.

Risks related to the **Legacy application** also posed a big threat at the beginning of project. In this case, the reason was the initial ignorance for the legacy application as most team members have not worked with/on it before. As the team got to know the code better, many risks decreased after the first few Sprints, however, they still caused issues through the whole project. The peak in Sprint 8 was caused by a potential fix, which was a workaround to integrate modern technology into the legacy application, which would have been worse than the initial bug as it would have introduced new technical debt. This risk is not bound to a point of time and could have happened at any time of the re-engineering activity. The graphs in this risk category overlap again, apart from the difference in the middle phase caused by one risk, *lack of knowledge about the legacy code*. It was expected that more coding would occur during Phase 1, hence the risk was expected to cause a high impact. However, it turned out that there was not enough time to get more coding done, so the impact was less severe. It is expected that this risk will need to be addressed in Phase 2.

5.2.2.2. Weighted Impact of Process Risks per Sprint

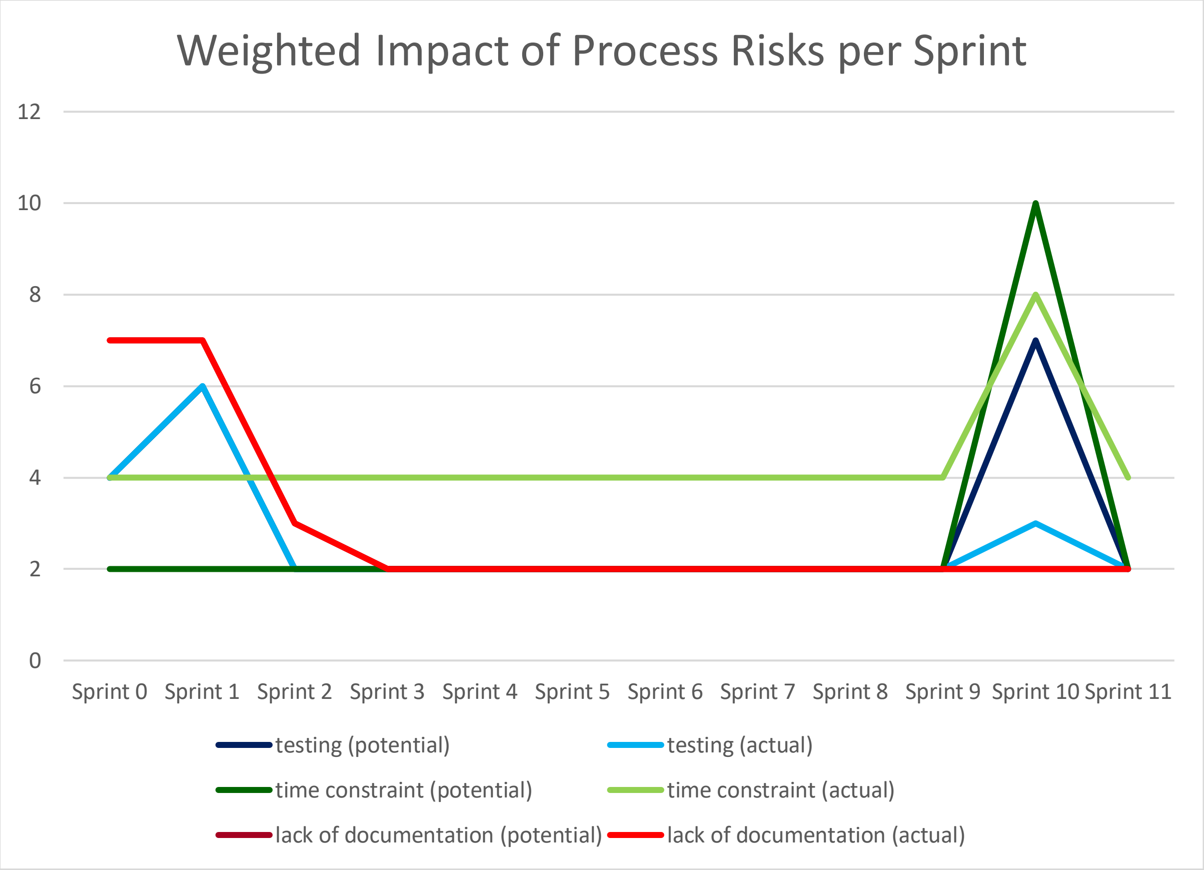

Figure 12: Weighted Impact of Process Risks per Sprint

Testing posed a subtle risk most of the time. After the initial risks, which were *missing test documentation* and a *lack of automated tests*, were overcome, the only continuous risks were the medium-rated risk of *inconsistent testing*. However, in Sprint 10 the risks drastically increased as it could have been fatal if the application had been shipped to the customers without thorough testing. For this undertaking a proper testing environment was needed, which it's lack posed another high-important risk. The only time the potential and the actual impact graphs did not overlap was in Sprint 10. It was not the case that the risk – *lack of time for testing* - caused less work or stress, but rather that it did not occur at all, as the planned big test in the end was deferred.

The risk sub-theme of *time constraints* only consists only of three different risks, of which only one posed a constant threat. This risk – a lack of time for meetings – was marked with a high importance as the communication and the exchange of ideas during such a project is crucial. This communication relied on meetings as the team was physical dispersed, which increased the risk. The peak in Sprint 10 was caused by the other two risks – a lack of time for testing and limited licensing time. The licensing time only bore a low risk as it could be renewed, in case it was needed, contrary to a lack of time for testing. This risk category is one of the few where the actual impact was most of the time higher than the expected one. This was a result of the risk *lack of time for meetings*, which was even more stressful than expected. The stress was caused by not being able to have a meeting when help was needed or to discuss ideas. The peak in Sprint 10 was caused by the risk of having a lack of time for testing as belongs into this risk category too.

The lack of documentation bore the biggest risk in the beginning as it would have been needed to understand the legacy applications and its functionality. This especially includes the test documentation as without it, the application could not be properly tested, and the documentation about the legacy application itself for the reason mentioned above. Both graphs overlap through the whole project, which does not prove the estimation would have been right, but still gives a realistic indication.

5.2.2.3. Weighted Impact of Social Risks per Sprint

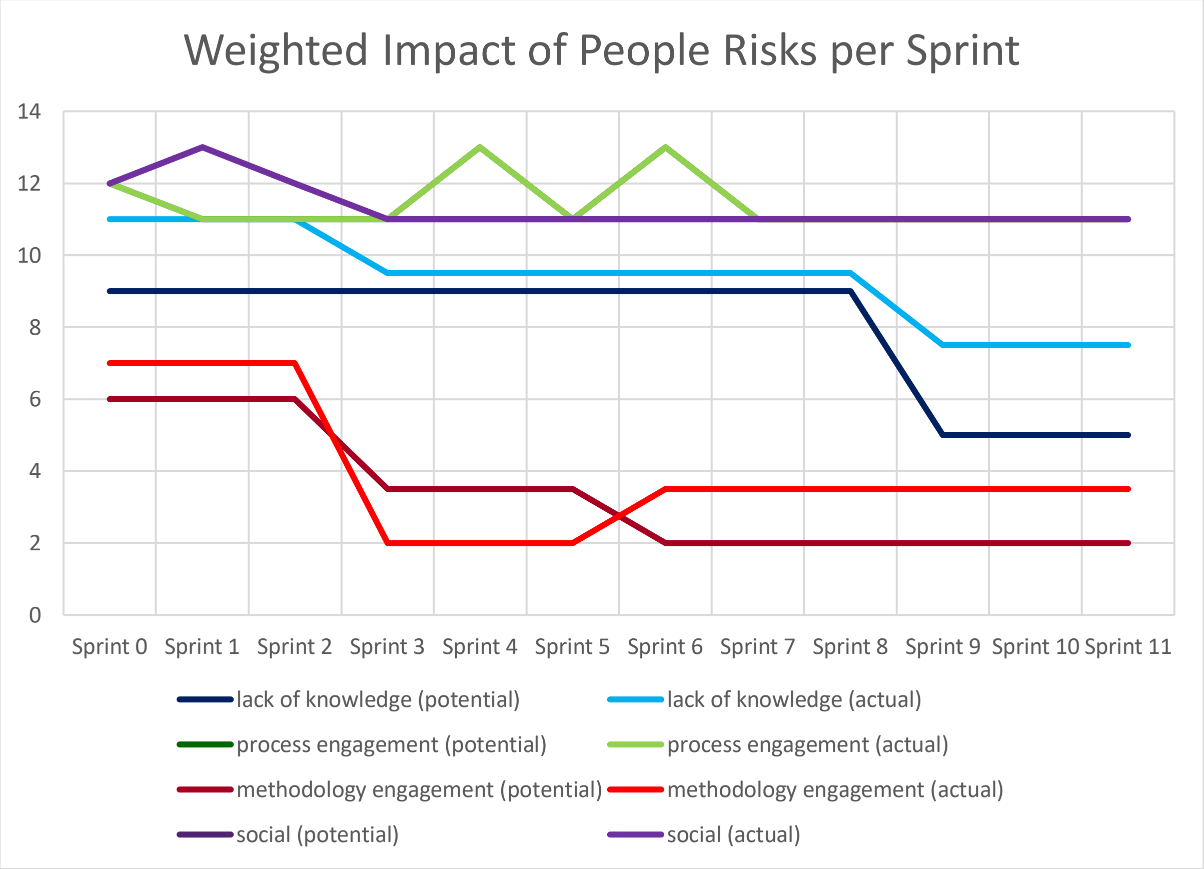

Figure 13: Weighted impact of people risks per Sprint

Even though it is not clearly pictured on the graph, most risk codes of **Lack of knowledge** were relevant in the beginning phase of the project. This can be explained by the steep learning curve of getting to know the legacy application. The risks were still present throughout the project, but they posed a much smaller threat. The graph stays significantly high as during the middle phase, the risk *lack of knowledge about legacy code*, which has already been mentioned in a different risk category, occurred as during this time development work was done. A risk which caused a high impact through the whole project was a *lack of state-of-the-art technology knowledge*. This risk was expected to have a low impact. However, a misjudgement about DLLs caused which created a substantial amount of work as it was more difficult to integrate re-compiled ones them than expected. Two risks which already have been mentioned had less impact than expected which are *lack of knowledge about old technology* and *lack of knowledge about legacy code*.

Process engagement risks were constant threats through the whole project. However, this sub-theme mostly consisted of low and medium risks, apart from the risk called *unwillingness to change*. Most risk codes posed a constant thread (*confused stakeholders, inconsistent issue tracking, holding meetings incorrectly, lack of time for meetings, no clear goal, inconsistent bug tracking*) as the events triggering them happened regularly. However, three risk codes only appeared once during the project, which namely are: *not clarifying importance of task, inconsistent testing, inconsistent use of version controlling*. Realistically, these risks potentially pose a constant thread in different projects, though, as they are triggered by tasks which can be called project chores (e.g.: testing, using of version controlling, etc.). The only risk which turned out to be a high impact risk was *Lack of time for meetings* as it turned out to be more difficult to organise meetings among all team members. The misestimation is covered up in the graph by a risk code which was initially expected to have a high-impact, but ended up only being a medium-impact one, namely *Unwillingness to change*. It had less impact as it was easier to change some team members' minds than expected, however it still delayed some tasks.

The drop in the **Methodology engagement** line indicates the learning process of the team using Scrum as a methodology. These initial risks consisted of an unclear definition of done, too big tasks, and poor prioritisation. Risks which were mostly solved early on, but still appeared through the project included the misevaluation of story points and neglecting the methodology. The rise which can be seen in the second half is poorly written user stories. All of the risks are rated to have either a low or medium impact. This is the only risk category in which expected and actual impact pattern are not even similar. Most of them had had even a higher impact than expected. This may have several causes: the lack of experience with Scrum of some team members or the, the fact that only facets of the methodology were used instead of everything, the small size of the team, or even plain ignorance of the impact of the risks on the project. The only risk with a smaller than expected impact was *wrong prioritisation* as it did not occur at all. This is since the next tasks were always obvious as either risks needed to be mitigated or there was a clear plan how to get the application back to market.

Another constant problem was posed by **Social** threats. Even though most of the risks were marked with a low or medium importance, the graph shows that the overall risk category can be considered as rather serious. The only hazard which was tagged with a high importance was *lack of a team*. Such enormous undertaking as a re-engineering project represents requires a lot of teamwork, especially as only some of the team members might have knowledge about the application. Not having enough team members, or a lack of a team spirit could be fatal for the success of the project. Even though, the expected and actual impact seem to overlap, a few risks were still misjudged. A risk which has already been mentioned which was poorly estimated was *unwillingness to change*. Another risk which causes less impact than expected was *lack of support*. The risk was rated medium as other members of the team were not working full-time on the project, so it was expected that getting support would be a struggle, however, there was barely a problem to receive support when needed. On the other hand, the risk *lack of team* had a higher than anticipated impact. It was expected that it will cause stress and more work to have so few people working on the project, however, it turned out to be even more stressful as it is not only the missing work force, but also the lack of having colleagues to share ideas with or discuss problems. A surprisingly severe risk was *lack of face-to-face working* as it should not be a problem to hold meetings online, especially in technology work. However, there were situations when it would have been easier to be in the same room to discuss problems or to look at code. Connection dropouts were also part of this problem.

5.2.2.4. Overall Comparison of Weighted Impact of Codes within a Theme

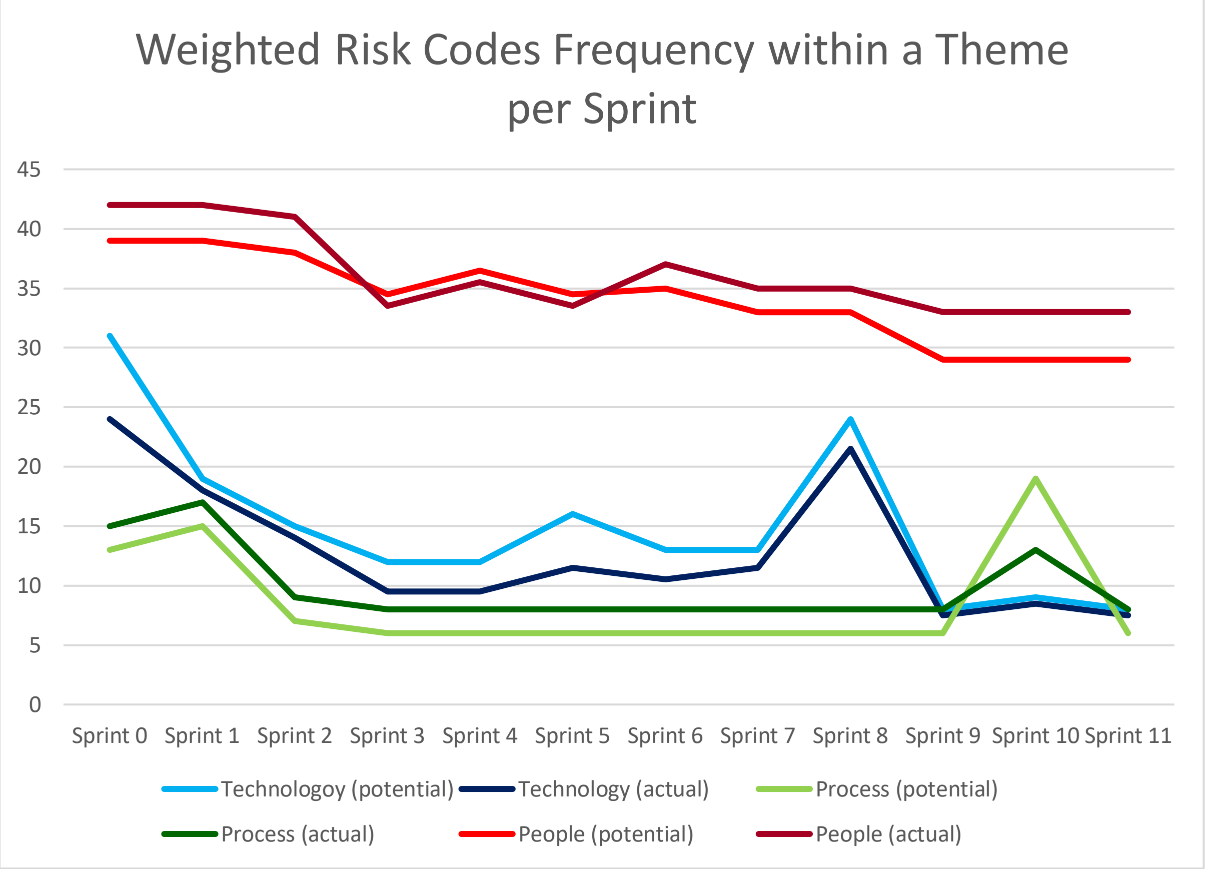

Figure 14: Weighted risk codes frequency within a theme per Sprint

It is remarkable how much more significant the impact of the risk codes in the theme "People", seen in red shadings, were compared to the other two themes. The impact was high through the whole project with only a little dip at the end. The even higher impact at the beginning can be explained by the risks related to team members having a lack of knowledge, which got resolved by getting to know the technologies, and also the methodology risks were resolved during the course of the project as the team members got used to Scrum. Most risk codes in this theme were marked of having a low or medium impact, however, the substantial amount of risk codes in this theme, made it to a considerable threat for the success of this project.

Another striking detail in this diagram is that the theme "Technology" peaked in Sprint 8. This was due to the trouble and eventually the failure of integrating 64-bit DLLs. Different risk codes occurred during this time and each of them was marked having a high impact.

It is also noticeable that the impact of the theme "Process" suddenly increased in Sprint 10 due to not having the application tested thoroughly at the end of Phase 1. Moreover, due to running out of time at the end of Phase 1, some tasks could not be executed and therefore risk codes related to having a lack of time emerged.

5.2.3. Mitigation Status of Risk Categories after Phase 1

This section describes the different mitigation statuses of the risk codes within a sub-theme after the first phase. A risk can hold one of the following statuses: not occurred, mitigated, partially mitigated, not mitigated, or deferred.

5.2.3.1. Standardised Mitigation Status of the Risks after Phase 1

The data has been standardised to show a relative view of the mitigation. Without this view it might seem that one risk category mitigated more risks just because it has a bigger amount of risk, whereas there was a bigger success in the mitigation of risks in a different category.

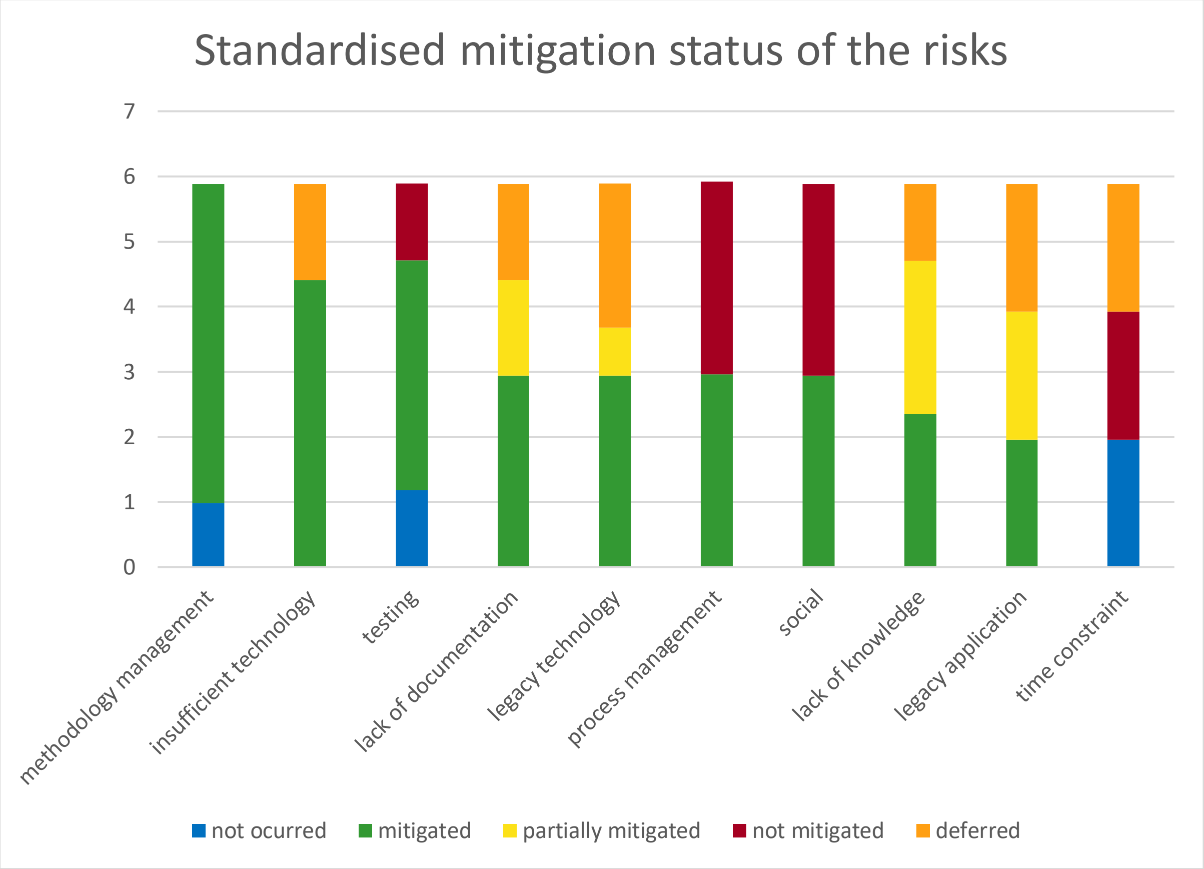

Figure 15: Standardised mitigation status of risk categories after Phase 1

The sub-theme which was clearly most successful in the mitigation of risks is **Methodology engagement**. This was because risks got mitigated by getting used to the methodology. However, the learning curve of using Scrum was very steep.

Another successful sub-theme in mitigating risks was **Insufficient technology** as the tasks behind the risks were crucial for the project and needed to be done e.g.: adding version controlling. A risk – code over-heap for IDEs and support tool - was deferred as it was not necessary to be mitigated in Phase 1. It only prevented the completion of some tasks to "pimp up" the code.

The mitigated risks in the category T**esting** were all the risks related to everyday testing e.g.: missing test documentation or a lack of automated testing. However, the problem in this category was the inconsistent testing. It did not get mitigated as the application was not always tested after a task was finished. The risk code *Not enough time for testing* did not occur as it was decided to do the full system pre-release testing before the release at another time.

Similar to the previous category, the risk codes mitigated in **Lack of documentation** are risks regarding documentation which was necessary for the success of the project, such as documentation about the legacy application itself. The commenting of the code was deferred as it was not necessary in the first phase, and it would have taken up too much time. Furthermore, many parts of the application will be re-written in the future, hence it would not made sense to put comments into the code. The risk code *A lack of re-engineering documentation* was only partially mitigated as no formal documentation exists, but rather slides from customer demos. However, a full test documentation was created during the first phase.

Half of the mitigated risks in the category **Legacy technology** were mitigated after the legacy code was transferred into a new IDE right in the beginning which was a crucial task. The other two risk codes were crucial too, but easier to mitigate. The first one was *A lack of knowledge about legacy technology* and the second was *A lack of old technology for developing old software*. Two out of three deferred risk codes are linked together as the first one is caused by a fundamental change of technology leading to the second risk code which is the *Struggle of integrate technology*. The major cause of these risk codes was the struggle to integrate re-compiled DLLs. This still needs to be done, however, it would have taken too long, so it was decided to execute this task in the next phase as it was not crucial. Another deferred risk code was *No replacement for technologies*. Some libraries needed to be replaced as they are deprecated, however, it is unknown if a comparable replacement can be found. It would have taken too much time to find new libraries and integrate them into the application. *Legacy code* was partially mitigated as the code was successfully integrated into a new IDE, however, it still remains to be fully understood by all team members.

The risk codes mitigated in **Process engagement** were mostly trivial (low impact) risks e.g.: not *clarifying importance of task*, apart *from unwillingness to change*, which addresses the team members. The mitigation of this risk code was less important for Phase 1, but rather for the future of this application. The non-mitigated risk of this category were threats which were partially not mitigated because of lack of time, or it simply was just forgotten to be done by the team members, such as *inconsistent testing,* and *inconsistent use of version controlling*. Both risks are rated to have a medium impact on the project. It can be said that the mitigation would have made the team members' life easier. One risk – *lack of time for meetings* – seemed impossible to be mitigated as it was dependent on the availability of other team members. As it was rated to have a high impact, the non-mitigation of this risks caused enormous damage in form of stress.

A similar pattern applies to the sub-theme **Social**. Most of the risk codes mitigated were marked as having a low impact, apart from apart from *Unwillingness to change*, which has already been mentioned. The other risk codes have been mitigated by clear communication and confirmation. Communication still failed sometimes as the risk code *Confused team members* could not be mitigated. The mitigation of the other risks was rather difficult as it was beyond the team's power. A *Lack of support* or *Lack of face-to-face meetings* could not be mitigated because other team members worked only part-time on this project. And without having more team members, a *Lack of a team* could not be resolved either.

The mitigated risk codes in **Lack of knowledge** are related to the knowledge of both old and state of the art technology. The risks were mitigated during the process of the project as knowledge necessary to do tasks was gained. Both of the partially mitigated risk codes are related to each other. The first one, *Lack of knowledge about legacy code*, has been mitigated –, and through this it helped to mitigate the other risk – *Limited number of people with knowledge about the legacy application* – as other team members gained partial knowledge about it. It has been mentioned in a previous paragraph why the lack of commentary in code was deferred.

The risks in the sub-theme **Legacy applications** which were mitigated appeared mainly in the crucial phase right in the beginning when the legacy code had to be transferred into the new IDE. In contrast, the deferred risk codes turned out to be more complicated than expected and the work behind the risk did not need to be done during this first phase e.g.: improving the readability of code. Both partially mitigated - *lack of knowledge about legacy code* has been mitigated *and limited number of people with knowledge about the legacy application* - risks were discussed in the previous paragraph.

Time constraint was the only category with no mitigated risk. The risk code *Lack of time for testing* did not occur, and the risk *Lack of time for meetings* were discussed in previous paragraphs The risk code *A lack of licensing time* was deferred as the team almost run out of time to compile new DLLs, however, the DLLs could not be properly tested before the licensing time ended, so it is unknown if the risk is mitigated or not.

5.3. Summary

The analysis produced a set of 44 different risk codes which are grouped together into sub-themes and themes. The sub-themes and themes reflect in which part of the project the risks were found. After the risks were identified, two types of impact weighting – the potential, and the actual - were added to each risk code and the mitigation status. The impact weighting adds a deeper meaning of how much extra stress, money, or time the risk caused. The mitigation status shows how the risk was left after the end of the first re-engineering phase.

The most remarkable result this analysis produced is that ***Social*** risk codes had a much bigger impact compared to the two other themes. Even though, most risks were rated as having a low or medium impact, the substantial amount of the risks made it a considerable threat. Moreover, most of the risk codes were a constant threat through the whole first phase. The risk codes which were most significant in the beginning were “lack of knowledge for technologies” and the risks around the methodology, which were resolved by getting used to Scrum. However, the methodology risks rose again in the second half of the first phase as Scrum became neglected after getting comfortable with it. Another risk code, which was also considered as very impactful was “lack of a team” as re-engineering requires teamwork, especially when only one person had all the knowledge about the application. Some of the risks were also caused by omissions, e.g.: inconsistent testing. The difference between the potential and actual impact comes from the underestimation of social risks.

The most significant risk code in the beginning in the theme ***Technology*** related to transferring the code to a new IDE. Also the risk of the deprecated technology ceasing to be supported and the risk of not using version control were very impactful. A peak of impact was caused in Sprint 8 by a technology which could not be integrated into the existing system, which caused several risks to appear. The difference between the predicted and the actual impact in this theme was caused by the fact that less coding than expected was done in the first phase, so the risk codes around this topic were less impactful. Also, the integration into a new IDE went more smoothly than expected and the lack of version controlling before it was implemented did not cause any major problems either.

The last theme, which is ***Process***, had the most impactful risk codes in the beginning and around Sprint 10. The risk codes in the beginning were related to the missing code and application documentation. The increased risk impact in Sprint 10 was caused by the lack of time for a big test before releasing the application to customers, however, this did not have a big impact in the end as testing was deferred to the next phase.

6. Scrum in Re-engineering

As discussed in the literature review, Scrum has been used as part of a re-engineering framework (Singh et al., 2019). However, I would go even further and state that in this case study Scrum was not only useful for process engagement it was also crucial for mitigating risks.. As one of the risks of software re-engineering is the unpredictability of tasks and challenges, an Agile approach can be a real benefit.

Agile was a response to the document-heavy, big planning upfront, contract-based, bureaucratic, and rigid classic methodologies, such as Waterfall, V-Model, and Spiral Model. It was popularised in the early 2000s following the publication of the Agile Manifesto (Fowler et al., 2001) when software developers noticed that projects need to be flexible and responsive to customers' needs. Agile rather prioritises business value through the software development itself instead of its process, documentation, and design (Schwaber & Sutherland, 2011).

Several methods and frameworks were invented based on Agile values and principles. The most popular Agile methods and frameworks are Kanban, Scrum, which was used for this project, and Scrumban (*State of Agile Report*, 2022).

6.1. Scrum Overview

Scrum is a light-weight framework which offers a customised way of working on different projects with a variety of requirements without having the need to follow a specific procedure (Srivastava et al., 2017) Due to its flexible requirements and adaptive solutions, the project can adapt to customer needs even after the contract has been set out.

Scrum is iterative. The development work is structured in cycles called Sprints. The Sprints should last at least two weeks, but no more than four weeks, and are continuous. These iterations are strictly timeboxed – they end on a specific date, no matter if the planned work has been finished or not and are never extended.

To estimate the velocity of a task (a user story), it is common in Scrum to use Story Points. Story Points are relative compared to traditional time measurements. They express an estimated of the overall work done to fully implement product backlog. The advantage of story points over absolute time measurements are that dates don't account for the errands which need to be run outside of the project such as answering emails or attending meetings.

At the beginning of each sprint, the teams put items (customer requirements), which are written as user stories, from a prioritised list (product backlog) onto the sprint backlog. They commit to finish the tasks by the end of the sprint. During the sprint these items are not allowed to change, and neither is it allowed to add items to the sprint backlog. Every workday, the team gathers briefly (around 15 mins) to inspect the progress towards the Sprint Goal, and adapt the Sprint Backlog as necessary, to adjust the upcoming work. At the end of every sprint the done work is inspected at the Sprint Review. The team is presenting their work to the stakeholders and the progress towards the Product Goal. During this meeting, the product may also be adjusted to incorporate feedback. Scrum emphasising on having a working product at the end of every Sprint. In software development terms this means integrated code, which is fully tested and potentially ready to be shipped. The last event at every Sprint is the Sprint Retrospective, which's purpose is to improve quality and effectiveness. The team discusses if the Sprint went well, what could have done better, and what problems were encountered. This process repeats until the Product Goal is achieved.

A Scrum Team consists of three roles: the Scum Master, the Product Owner, and the Developers (also referred to as The Team). The Developers are committed to execute the tasks on the Backlog and building the product. The Team is self-organised and cross-functional – it includes all the expertise needed to deliver the product. The skills required can be very broad, however the Developers are always accountable for (Schwaber & Sutherland, 2011):

- Creating a plan for the Sprint, the Sprint Backlog
- Instilling quality by adhering to a Definition of Done
- Adapting their plan each day toward the Sprint Goal
- Holding each other accountable as professionals

The Scrum Master ensures that Scrum is practiced as described in the Scrum guide. They are helping the team (and the organisation) understanding Scrum theory and improving its practice.

The Product Owner represents the customers (in some cases they are the customer) within the team. They are responsible for maximising the ROI (Return of Investment) and the Product Backlog including its prioritisation.

The number of members of the Scrum Team should not exceed 10. Smaller teams have been proven to work better as the communication and productivity are increased (Schwaber & Sutherland, 2011). If a team is getting too large, they might not share the same Product Goal, Product Backlog, or Product Owner anymore, therefore it should be considered to reorganise them into multiple cohesive Scrum Teams.

Scrum should rather be seen as a philosophy, or structure supporting a project to achieve its goal and create a value (Schwaber & Sutherland, 2011). It is not a process with strict rules, and only uses a framework to define the parts required to practice Scrum. This makes Scrum extremely flexible and fit for all kinds of teamwork, which is one of the reasons why it became so popular.

6.2. Scrum in this Project

The Scrum Team of this project consisted of five people, of whom only one was working full-time on this project. The Developers were a full-time software developer (myself), and a part-time one. The Product Owner was the researcher who had initially written the software, but who was now not in a position to do further development or modernisation on the product. The role of the Scrum Master was shared by two people from a consultant company hired to support the project.

Stand-up meetings were held twice a week instead of daily. Because of the small number of developers and the reduced amount of time spent on the project two meetings a week were sufficient.

The sprints were held in a three-week cycle, which is within the recommended duration of two to four weeks. The Sprint Review and Sprint Retrospective happened on the Friday of the third week. Every other Sprint (every six weeks) the Customer Demo took place within the Sprint Review, where the customers got to see the progress made during the last couple of Sprints and had the chance to ask questions or give feedback. The Sprint Planning for the upcoming Sprint took place after the Sprint Retrospective. During the Sprint Planning the User Stories which were intended to be worked on in the next Sprint were given a velocity. Instead of using Story Points right away, the popular T-Shirt sizing method was used to estimate the size of a task. This method uses the US way of indicating T-Shirts and Story Points are given to each size. This way takes the advantages of the relative time estimations of story points even further. Before User Stories which fit within the set time limit for the upcoming Sprint were added to Sprint Backlog, they were prioritised.

Because of the nature of this project – making functional changes to an operational system - there was a potential for risks to have a severe impact. This affected the prioritisation of the User Stories during the Sprint Planning as the risks and problems that had emerged during the previous sprint were discussed in the Sprint Reviews, hence tasks to mitigate risks might have been added to the Product Backlog with a higher prioritisation than a normal re-engineering task. Tasks were prioritised in three categories: low, medium, and high.

A tool which is often used in Scrum is a Kanban board. It was used to visualise the work. The identified user stories were collected in the Product Backlog and the relevant ones were moved into the Sprint Backlog during Sprint Planning. During the Sprint, the items were moved into the In Progress column. Before a user story could be moved into the Done column, it was reviewed by being aligned with the Definition of Done and evidence for the Customer Demo was added.

6.3. How Agile and Scrum Practices Supported this Project

As Scrum is a part of Agile, Scrum follows Agile practices as well as its own guidelines. As mentioned, Scrum was chosen for this re-engineering project because its flexibility is a good response to the unpredictability of challenges and risks.

6.3.1. Agile Principles

In this section the Agile Principles listed in the Agile Manifesto (Fowler et al., 2001) are discussed in the light of the experience in this project in terms of how they supported the re-engineering process. This is a reflective analysis from a subjective point of view. The principles most relevant for this particular project were: customer satisfaction, embracing change, incremental and iterative delivery, customer involvement, face-to-face conversation, delivering working software, and team reflection.

According to the Agile Manifesto, customer satisfaction has the highest priority in Agile software development. In this project, at the end of every other Sprint, the functionality of the software was confirmed to the stakeholders during a customer Demonstration.

One of the most challenging parts of a re-engineering project is the unpredictability of tasks and challenges. Agile embraces change instead of resisting it. When it comes to risk management and mitigation, amendments need to be executed as soon as possible. This means that an agile approach can be used to support risk mitigation.

The incremental and iterative delivery approach in agile projects product allows regular feedback to the development team. In this case, the delivery happened every six weeks, and was discussed during the customer demonstration, where the stakeholders could check the functionality of the application, and confirm that no major features had been broken during the re-engineering process. During these meetings stakeholders also had the chance to express concerns, mention criticisms and give feedback.

Customers feedback is also related to another Agile principle, the importance of customer involvement. Frequent feedback from customers allowed fast correction of mistakes. The goal of this project was not to re-engineer the whole application at once, but rather to make it fit for sale again within 8 months. This meant that not everything highlighted for re-engineering could have been done. Close cooperation with the customers was needed to know what was important for the re-sale. Additionally, this collaboration was bidirectional, so when some things did not work out as intended, alternative ways were suggested to the customers by the development team. This collaborative approach helps.

As the team was physically dispersed, face-to-face conversations, which the agile manifesto states are the most efficient and effective method of conveying information, were not possible. It can be said that this quite probably impaired the project, and some problems would have been solved faster if this Agile principle had been implemented.

Agile emphasises delivering a fully functional application at the end of every Sprint, which played into our hands as during the re-engineering process as the application was still being used and needed to be kept up and running. It also ensured that the software was always ready to be demonstrated to the stakeholders.

To reflect regularly how to become more effective and how to adjust such a behaviour, Sprint Retrospectives were used. This will be discussed in more detail in the next chapter.

6.3.2. Scrum Practices

In this chapter, I will consider the common Scrum practices used in this project and discuss how they supported the re-engineering process. To give an even better insight, I will use an example from the project to explain how each practice influenced the project. This example is from the time when the re-compilation from Matlab files to 64-bit DLLs failed, triggering two risks: fundamental change of technology and struggle to integrate technology. See 4.3.2.1 for more details about this event. More examples will be given as needed. This chapter is a reflective analysis from a subjective point of view.

The iterative pattern in the form of Sprints supported this re-engineering project by always needing to have a deliverable product, which means that at the end of every Sprint, the application needed to work bug free. It was a requirement of this project that the application was kept up and running, so that other software engineers could still add functionality to it.

At the beginning of the project, only a few tasks were planned out properly. Most of the other tasks were identified during the process of the project. Re-engineering projects can rarely be properly planned as many aspects of the work to be undertaken are unknown initially. Tasks to be done kept emerging during Sprints and were flexibly added to the backlog for the following Sprint during the Sprint Review. Even when tasks had been planned, they often turned out to be far more complicated than expected. I found that Scrum gave us the agility to handle such events. Remembering the example given in the beginning of this chapter, it was first planned to just re-compile the DLLs into 64-bit, however, it turned out that this task was more complicated as 64-bit DLLs cannot be integrated into a 32-bit application, which this legacy application is. Therefore, it was planned to compile older versions of the DLLs in 32-bit, which are still more modern than the old ones. However, it turned out that the interfaces of the modern DLLs are incompatible with the application.

Moreover, the flexible planning given by the Sprint Review helped a lot with the mitigation of the risks too, as countermeasures could be done right away in the next Sprint. In the previous example, we tried to mitigate the two risks of fundamentally changing the technology and the struggle to integrate different technologies. However, flexible planning helped to handle more risks. Another example would be that at a certain point in the project the manual user acceptance testing were taking up too much time, so a user story was added to the upcoming Sprint to write an automatic user acceptance test. This practice is also called Backlog Refinement, which was later on added to the Scrum Guide (Schwaber & Sutherland, 2011).

Meetings like the Sprint Review or the Daily Stand-up (Daily Scrum) were the perfect opportunity to discuss ideas or raise concerns. Especially the Daily Stand-up helped discuss concerns about tasks. If a problem could not be resolved during the Sprint or more actions were needed, the discussions were recessed to the Sprint Review. These techniques also supported me to stay on track with the work as I did not work full-time on the project, so time management was a challenge sometimes. The Stand-Up itself did not help to uncover risks – this rather happened during the execution of tasks - most of the time, however it helped to understand the extent of the impact of risks. Given the example with the DLLs, it was only during a meeting that we realised the problem was so big it could not be solved during that phase of the project as it would have caused more technical debt than it fixed, which is the against every intention of the project.

In addition to discussing completed work in Sprint Reviews, we also discussed the risks which had occurred during the previous Sprint. If they had been resolved, no further discussion was needed, however, if this was not the case, they were analysed further and if needed additional User Stories were added to the Backlog to mitigate them. When we found out that the DLLs could not just be re-compiled into 64-bit ones and integrated into the project, it was decided during a Sprint Review that a test harness for 32-bit DLLs from a modern compiler should be created to find out if this alternative would work. These meetings also helped to deal with risks quicker. The social risks could often be mitigated during meetings. Many risks related to having a lack of knowledge about something and were resolved by getting support from other team members during meetings. Even though these meetings were an extremely important for the project, it was often a struggle to organise them. Most team members only worked part-time on the project and many of them also worked remotely, so it was difficult to find times when everyone was available.

As risks were discussed during Stand-ups, there was a struggle to keep these short. A Daily Scrum should not take longer than 15 minutes, however most of these took the team around 30 mins as the discussions went beyond a quick overview about what we had done and what we planned to do. We discussed risks and their mitigation, upcoming tasks and how to execute them, and the future of the project. Even though, the Scrum Guide suggests that this kind of meeting should be kept short, I believe that longer Stand-Ups helped a lot with the progress of the project. As mentioned in a previous chapter, one of the risks was "a lack of time for meetings", having already a meeting planned and all (many) team members together, it was good to make use of that time. A better way of handling this situation would have been to set the first 5 minutes of the meeting out for the Stand-up and then use the rest of the time for any further discussions.

Having discussed social risks in previous chapters and how much of an impact they had, this highlights the importance of the Sprint Retrospective, in which the good and bad aspects of the last Sprint were discussed. Many social risks were uncovered during these meetings as difficulties were reconsidered. Risks like "a lack of time for meetings", and "lack of face-to-face working" were identified during Sprint Retrospectives.

During the first Customer Demo, when the team presented the progress to the Stakeholders, the team was asked by one of the Stakeholders how they knew when a task was done. As there was no Definition of Done, no answer could be given. After this, the team started adding to every User Story a Definition of Done as it did not only show the stakeholders that a task had been done properly, but also helped the software developers to know when a task was finished. After this incident, the team also started collecting evidence for every done task to document the progress, but also to be able to prove it to the stakeholders as they wanted to see progress.

As with many software development projects, time estimation for tasks was particularly difficult for this re-engineering project. Time Boxing did not work out very during this project as tasks often took longer than they should have. Once a task was started, it often turned out to be a rabbit hole full of little sub tasks. Time Boxing requires to stop after a set amount of time, which was not possible as some tasks needed to be done in this sprint. However, this task extension made the stakeholders suspicious why less work had been completed than expected during Sprints. To aid transparency the team started visualising their work using a burndown chart showing when a task took longer or when work has been added. It needs to be mentioned though, that even though individual tasks were not time boxed, the Sprint itself was.

As mentioned in a previous chapter, Story Points were equivalent to hours. For every hour available in each sprint, user stories with the appropriate amount of story points were put into it. However, using real time measurements instead of relative estimations made the evaluation of velocity difficult as the team did not compare the size of jobs, but rather estimated how long it will take. In the beginning, the estimations of the size of the user stories were rather experimental. These estimations were mostly inaccurate but mostly within a 10% margin, however for repeating tasks such as preparations for Sprint Reviews, the estimations became more accurate.

A good example of finding it hard to estimate task size is the task allocation for Sprint 3. It was assumed that after the transfer of the application to a modern IDE many run time errors would occur. At least a whole sprint was allocated for fixing these bugs and making the application work again. However, this was not necessary. The application ran smoothly with only a minor bug. To use the gifted time, the team allocated new tasks into the current Sprint, which is against the Scrum guide as tasks should only be added to the Sprint Backlog during the Sprint Planning. However, as mentioned, time was of the essence in this project.

The prioritisation of tasks was most of the time not an object of discussion as the approach of the project was rather reactive. Most of the time it was obvious which tasks needed to be done next, especially when it was a risk to mitigated. Taking the DLL example, once the team realised that the 64-bit DLLs could not be integrated in the system, it was clear that the next task would be to find a way to compile 32-bit DLLs.

Overall, the team became proficient in using Scrum. Velocity estimations became more precise, different types of meetings were used in the appropriate situation, and Sprint Retrospectives were held after each Sprint. However, I noticed that the team had a tendency to become complacent over the time as they became comfortable using Scrum. User Stories became less and less detailed and informal to the point when the team did not know anymore what the task were required to complete the user story. Definitions of Done were also missing for some User Stories.

6.3.3. Summary

The table below summarises all the risks which were mitigated using Scrum techniques during the re-engineering process of this application. It needs to be highlighted that the risks in the table are not the risk codes mentioned in chapter 5, but rather an abstract description of the major risks. This has been done to avoid repetition as similar risks were mitigated in the same way, except when stated.

Risks	Mitigation strategies using Scrum
The precise nature of re-engineering activities was not known in advance.	Continual re-prioritisation of tasks, and planning in detail for the short-term. (Sprint planning / Backlog grooming)
Previously unidentified risks becoming apparent as the project progressed.	Revising and adapting ways of working within the team to address risks as they arise (Sprint Review)
The causes of struggles were difficult to identify.	Discussing issues during Sprint Retrospectives helped to uncover the actual risk that caused them
Being stuck	Highlighting potential sticking points early (Daily Stand-up) so that technical mitigations can be put in place (also feeding into Sprint Review and Planning)
Lack of knowledge	Highlighting the issue during Daily Stand-ups to receive help from team members (also feeding into Sprint Review and Planning).
Having to struggle with time management because of working part-time on the project	Discussing done work and planning work for the next few days during Daily Stand-ups
Difficulties explaining to stakeholders what was done in previous Sprints and how to showcase it	Adding a Definition of Done to every user story and collecting evidence for every done task to document progress

Table 10: Summary of risks mitigated by using Scrum practices

7. Discussion

The two research questions which were answered by this study are:

- RQ1: What types of risks are encountered in a software re-engineering project and how are they mitigated?
- RQ2: How helpful are Scrum practices to support a software re-engineering process?

7.1. The identified risks compared to previous papers

This study identified three different risk themes, namely technology, process, and people during the first phase of a re-engineering project. Each of these risk themes is made up of three or four different sub-themes. The potential and actual impact, as well as the success of the mitigation, of the risks of each sub-theme was identified and analysed. The risk categories were coded without using a pre-existing codebook and the analysis was based on the collected data. This makes it interesting to look at the risk categories of previous papers and compare them with my results. Rashid et al. (2013) listed in their paper risks which have frequently been mentioned in different papers. The risk categories he mentioned in his paper are *User satisfaction*, *Cost*, *Forward Engineering*, *Reverse Engineering*, *Performance*, and *Maintenance*. Clemons et al. (1995) also lists different risk categories they found in their previous studies: *Financial Risks*, *Technical Risks*, *Project Risk*, *Functionality Risk*, and *Political Risk*. Surprisingly, there is very little overlap between the risk categories mentioned in these papers and my findings. The findings of Rashid et al. (2013) do not show any similarities to mine, although it could be argued that the risk codes, *a lack of time for testing*, which appeared in this case study, could be considered as a *User Satisfaction* risk. Users can be affected by bugs if an application has not been tested thoroughly. Clemons et al. (1995) paper contains two risk categories which overlap with mine, Technical Risk and Project Risk. My analysis describes a whole theme dedicated to technological risks and the description of Project Risk in Clemons et al. (1995) paper can be compared to some risks mentioned in the ***Social*** theme. Interestingly, Rashid et al. (2013) and Clemons et al. (1995) papers also only have one overlapping category, Financial Risks. The variety of risks in different papers shows us that the risks of different re-engineering projects can vary a lot. The novelty of this study stems from the fact that social risks were taken into consideration, as they were found to have a major impact, and are not just being mentioned as a side issue.

Social risks seem to be totally ignored in some papers, e.g.: (Rashid et al., 2013). Other papers mention certain team or social risks, such as (Khadka et al., 2014). They describe the reluctance of software developers to modernise legacy applications as they often conceive them as their “baby”, or they fear redundancy following the modernisation process, so they refuse to share their knowledge. Further social risks addressed in the paper are the non-understanding of managers for the need of modernisation, and reluctance of providing a sufficient budget for it. However, they do not mention any risks which could occur within the team or even related to a single person. This proves that social risks are often overlooked or forgotten about. Even though they might not be directly related to the project - in form of the actual software development work – and do not seem to be obvious, they are as critical, or even more, than other risks.

I can imagine that some risks which only appeared at some point of time in this project are constant threads in other ones, such as *inconsistent testing*. It also always needs to be kept in mind that there are risks which are not time bound but just seemed to happen on a specific time in this project, such as the risk code *fix is worse than problem*. This is totally individual. Comparing the risk categories from Rashid et al. (2013) and Clemons et al. (1995) to mine, the reasons for the differences can be singled out. Financial Risks did not appear in my analysis as the budget for the project had already been approved when my contribution started. Forward and Reverse Engineering, and Performance risks did not appear as the re-engineering process was not advanced enough in the First Phase to be concerned about such risks. Maintenance risks were not of importance in the First Phase either as the re-engineering process will continue in the next phase, so no maintenance work could be observed. As the original system designer was part of the re-engineering team, functionality risks, such as the system not meeting present or future needs, never threatened the project. Because of the high importance of the re-engineering project for the organisation, political conflicts did not endanger the success of the project.

Even though, the risks of my findings are barely overlapping with risks identified in previous studies ((Rashid et al. (2013) & Clemons et al. (1995)), my findings provide a better general understanding of the risks which can appear during a re-engineering project. Project teams will be able to look the risks and their impact and be able to prioritise the area of re-engineering, and also know what kind of mitigations can be put into place.

7.2. Agile in re-engineering

Scrum was not just helpful by embracing the uncertainty, but also by handling the prioritisation of the tasks, which was crucial for the management and mitigation of the risks. As explained by means of the Iron Triangle in chapter 1, the time and money were a fixed factor in Scrum, as well as in this project, so the flexible scope was the features, which were constantly re-prioritised to fit to the customer value and eventually bring the application back on the market. The Agile characteristic of regular meetings not only helped uncovering risks, but also supported staying on track and solving problems with tasks. Another Agile/Scrum practice essential for uncovering risks, especially social ones, was Sprint Retrospectives as it was discussed what went well or poorly in the previous Sprint during these meetings. Timeboxing is one of the Scrum practices, which did not work well for this re-engineering project as tasks often took longer than expected, which left stakeholders suspicious.

Suggesting an Agile environment for re-engineering work is the second reason why this study is novel as Agile was mentioned in regards of re-engineering before, however, never to such an extent. Holvitie et al. (2018) penned one of the few papers mentioning Agile and re-engineering related matters, even though it only talks about technical debt, in one breath. By taking a survey among practitioners, it was found that Agile practices are perceived to have an effect – either positive or more diverged - on managing technical debts. Dealing with technical debt was a big part of the re-engineering project this study is about. Taking a closer look into the results of their survey, it can be seen that some results are overlapping with mine e.g.: most practitioners perceived iteration reviews/retrospectives and adhering to coding standards having a positive effect managing technical debt. However, practices I would have viewed as useful, such as on-site customers, were mostly perceived as neutral, and core practices of Agile, like iterations, backlogs, and daily meetings, were rated having a positive impact by only 50-60% of the participants. It needs to be considered that the differences between my results and Holvitie et al.'s may emerge from the fact that the survey was conducted with software engineers who don't particularly work on legacy applications. Only 40% of the participants had very well or well knowledge about technical debt.

For the sake of completeness, I want to mention that Singh et al. (2019) published a paper describing a re-engineering project using Scrum, claiming that taking an Agile approach reduced maintenance cost and improved the maintainability. However, the project narrowly focused on reducing a set of code complexity metrics over a single Sprint in a non-industry context, where the only stakeholders were themselves. Another paper lacking credibility was composed by Masood & Ali (2014). They state that planning the next re-engineering Sprint based on the previous one and breaking down tasks into User Stories was very helpful. However, there is no background information about the interviews this data is based on.

7.3. Overall impact

The overall outcome of this research project is a combination of an initial set of risks including their impact on a re-engineering project, which are going to form the basis of further research into different types of re-engineering project to make a more generalised framework. Project teams can orient themselves according to the framework and prioritise the areas of re-engineering and what kind of mitigation to put into place. Also, the proposed use of an Agile environment supports project teams by giving them a suggested way of how to deal with the factor of uncertainty during a re-engineering project. By this both of the research questions have been addressed and answered.

8. Limitations and Future Work

The aim of this study is to contribute to the development of a framework for risk identification and mitigation in software re-engineering. The limitations are discussed here.

This study is based on a single case study, it is therefore difficult to generalise the risks found here with those that might occur in other re-engineering projects. Also, no assumptions about the probability of the risks can be made as multiple re-engineering processes need to be observed for that. Some risk categories are missing from this study, because it was based on one phase of a re-engineering project. For example, Rashid et al. (2013) found that the maintenance of a system is a risk category, which I would consider as important. However, as the case study ended before the maintenance process started, these risks were not identified. Another risk category missing from this analysis is financial risk as the budget for this project had already been approved when my participation started. This risk was mentioned in two separate papers about re-engineering (Rashid et al., 2013) and (Clemons et al., 1995), thus they can be considered as important.

There are also limits with how the findings were presented. The graphs in chapter 5 presenting the potential and expected impact (e.g.: Figure 11: Weighted impact of technology risks per Sprint) do not show when each risk became apparent. It might be visible when a risk from a certain sub-theme emerged or was mitigated, however if another risk from a sub-theme was mitigated or emerged at the same time, it could get compensated. Moreover, it does not state which risk code it was.

Another major drawback of this study is the that the potential impact was not assessed at the beginning of the project but was done afterwards with the rest of the analysis. This made it impossible to prove how well the team estimated risks. For my future case studies, risks will need to be assessed beforehand. Moreover, my study would have stood to benefit from more notes. Even though, the data from the Scrum board was sufficient to track back to when certain risks were triggered, it would have been easier and faster to analyse if I had written up the initial process with more detailed, and earlier notes.

Finally, there was a strong subjective element to this work as I was directly involved in this project as a software engineering, which could be considered as a limitation. The weightings for each impact, the potential and actual, were subjective as I was the only person deciding them. This could have been improved by doing interviews with each person involved in this project. However, in my opinion, the limitations of a researcher involved in the project stated in chapter 3, e.g.: influencing the work researched, and getting too involved in the project, did not restrict the quality of my research.

For future work I am planning to continue this research in for my doctoral thesis. I want to consider the two missing risk categories mentioned above – maintenance and finance - so the framework covers the whole re-engineering cycle, by carrying out multiple long-term case studies in different companies, as well as continuing the research with the current this project. Maintenance and financial risks could arise in future phases of this project by having a lack of budget for future maintenance or improvements and having a lack of time or staff to do maintenance work or quick fixes which would cause new technical debt.

Undertaking multiple case studies would also allow me to calculate a sensible probability to each risk code. I think the framework would also stand to benefit from a case study on a re-engineering project which does not use Scrum (or other Agile frameworks), but rather a traditional methodology, to compare how much Agile essentially supports such an undertaking. With case studies using Agile, the analysis about how Scrum supported the project could be more detailed as this one was rather restricted due to the limited number of team members. The adapted Scrum practices in this project also made it very individual. Attaining insights to a re-engineering project with a standard Scrum team – as described in the Scrum Guide (Schwaber & Sutherland, 2011) - would certainly provide useful insights. Case studies from different re-engineering projects will also yield more objective results. The final framework will support re-engineering teams to identify potential risks, their impact and probability in different stages and situations of the process. It will also focus on the prioritisation and mitigation of identified risks. The framework will be novel as it suggests using an Agile environment for re-engineering work.

I also want to mention what the future of the re-engineered application investigated in this study will look like. As Phase one addressed bringing the product back to market, Phase two will be an overall re-engineering process, which includes taking the core application and turning it into a 32-bit system using an MVC architecture. It will also deal with the replacement of several libraries, the upgrading of the user interface, the improvement of the cloud version, and more things which will be mentioned in chapter 4.4.

9. Conclusion

Legacy software is becoming increasingly ubiquitous, and most companies nowadays need to deal with the challenges associated with this phenomenon. On many occasions re-engineering is the only logical way to deal with such software. However, such projects are prone to many kinds of risk. This thesis has identified the different risks encountered in one particular re-engineering project and analysed their potential and actual impact on the project, as well as their mitigation status at the end of a given phase of development. Moreover, it has evaluated how helpful Scrum practices were to support this project.

The first main observation is that substantial risks were faced in several aspects of the re-engineering project, which were themed around people, process and technology. Interestingly, while much has been written about the technical aspects of re-engineering, the presence of risk in social situations relating to re-engineering appears to have been overlooked in the literature. Risks related to human interactions were not found to have a bigger impact than risks from other areas, however it was surprising to discover that the number of those risks was much higher than those found in other aspects of the project. Furthermore, the social risks were often either underestimated or not even recognised.

While the limited duration of the case study might have led to the omission of some risks related to maintenance and cost, it successfully identified risks related to personal relationship and the re-engineering process itself, which were not reported in previously published papers. Having a small Scrum team with just one developer and only using some facets of Scrum limits the generalisability of the results, however it is clear from the findings that Agile brings in the flexibility needed to succeed with such an project, since many of the tasks which constitute a re-engineering project are often unknown at the beginning. This flexibility was also key for mitigating risks quickly.

Further research is required to better understand the implications of Agile on a re-engineering project, specifically case studies with a standard-sized Scrum team using all facets of Scrum. Moreover, the study of a complete re-engineering cycle is needed to identify all risks emerging during such an undertaking.

Based on these conclusions, organisations need to become more aware of people-related risks to ensure the success of their re-engineering work. As many kinds of risk could arise from anything people related, there is no definite way to mitigate these risks, however, I discovered that miscommunication is often the underlying cause and improved communication often mitigated the risk.

References

15th State of Agile Report. (2021).

Alhojailan, M. I. (2012). Thematic analysis: A critical review of its process and evaluation. *West East Journal of Social Sciences*, *1*(1), 39–47.

Awad, M. A. (2005). A comparison between agile and traditional software development methodologies. *University of Western Australia*, *30*, 1–69.

Azizyan, G., Magarian, M. K., & Kajko-Mattsson, M. (2011). Survey of Agile Tool Usage and Needs. *AGILE*, 29–38.

Bakar, H. K. M. A., Razali, R., & Jambari, D. I. (2019). Implementation Phases in Modernisation of Legacy Systems. *2019 6th International Conference on Research and Innovation in Information Systems (ICRIIS)*, 1–6. https://doi.org/10.1109/ICRIIS48246.2019.9073628

Begel, A., & Nagappan, N. (2007). Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 255–264. https://doi.org/10.1109/ESEM.2007.12

Bennett, K. (1995). Legacy systems: coping with success. *IEEE Software*, *12*(1), 19–23. https://doi.org/10.1109/52.363157

Birchall, C. (2016). *Re-engineering legacy software*. Simon and Schuster.

Boehm, B. W. (1991). Software risk management: principles and practices. *IEEE Software*, *8*(1), 32–41. https://doi.org/10.1109/52.62930

Braun, V., & Clarke, V. (2021). *Thematic Analysis: A Practical Guide* (Vol. 1).

Buschmann, F. (2011). To Pay or Not to Pay Technical Debt. *IEEE Software*, *28*(6), 29–31. https://doi.org/10.1109/MS.2011.150

Clarke, V., Braun, V., & Hayfield, N. (2015). Thematic analysis. *Qualitative Psychology: A Practical Guide to Research Methods*, *222*(2015), 248.

Clemons, E. K., Row, M. C., & Thatcher, M. E. (1995). An integrative framework for identifying and managing risks associated with large scale reengineering efforts. *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, *4*, 960–969 vol.4. https://doi.org/10.1109/HICSS.1995.375651

Dawson, C. W. (2005). *Projects in computing and information systems: a student's guide*. Pearson Education.

Dayaratna, A. (2021). *Quantifying the Worldwide Shortage of Full-Time Developers*.

Diana, R. (2010, July 21). *Re-Engineering In Agile Development Can Just Be Refactoring*. Agile Zone. https://dzone.com/articles/re-engineering-agile

Dobson, P. (1999). *Approaches to Theory Use in Interpretive Case Studies–a Critical Realist Perspective*.

Dresch, A., Lacerda, D. P., & Miguel, P. A. C. (2015). A distinctive analysis of case study, action research and design science research. *Revista Brasileira de Gestão de Negócios*, *17*, 1116–1133.

Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., & Gorton, I. (2015). Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 50–60. https://doi.org/10.1145/2786805.2786848

Fanelli, T. C., Simons, S. C., & Banerjee, S. (2016). A Systematic Framework for Modernizing Legacy Application Systems. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 1*, 678–682. https://doi.org/10.1109/SANER.2016.40

Fitzgerald, B. (2012). Software Crisis 2.0. *Computer*, *45*(4), 89–91. https://doi.org/10.1109/MC.2012.147

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Fowler, M., Highsmith, J., & others. (2001). The agile manifesto. *Software Development*, *9*(8), 28–35.

Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2003). *Fundamentals of Software Engineering* (Second).

Holvitie, J., Licorish, S. A., Spínola, R. O., Hyrynsalmi, S., MacDonell, S. G., Mendes, T. S., Buchan, J., & Leppänen, V. (2018). Technical debt and agile software development practices and processes: An industry practitioner survey. *Information and Software Technology, 96*, 141–160. https://doi.org/https://doi.org/10.1016/j.infsof.2017.11.015

Jain, S., & Chana, I. (2015). Modernization of Legacy Systems: A Generalised Roadmap. *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015*, 62–67. https://doi.org/10.1145/2818567.2818579

Johnson, D. M. (1996). The Systems Engineer and the Software Crisis. *SIGSOFT Softw. Eng. Notes*, *21*(2), 64–73. https://doi.org/10.1145/227531.227542

Kazman, R., & Pasquale, L. (2020). Software Engineering in Society. *IEEE Software*, *37*(1), 7–9. https://doi.org/10.1109/MS.2019.2949322

Khadka, R., Batlajery, B. v, Saeidi, A. M., Jansen, S., & Hage, J. (2014). How Do Professionals Perceive Legacy Systems and Software Modernization? *Proceedings of the 36th International Conference on Software Engineering*, 36–47. https://doi.org/10.1145/2568225.2568318

Lehman, T. J., & Sharma, A. (2011). Software Development as a Service: Agile Experiences. *2011 Annual SRII Global Conference*, 749–758. https://doi.org/10.1109/SRII.2011.82

Masood, A. Bin, & Ali, M. A. (2014). *Applying Agile Requirements Engineering Approach for Re-engineering & Changes in existing Brownfield Adaptive Systems*.

Matharu, G. S., Mishra, A., Singh, H., & Upadhyay, P. (2015). Empirical Study of Agile Software Development Methodologies: A Comparative Analysis. *SIGSOFT Softw. Eng. Notes*, *40*(1), 1–6. https://doi.org/10.1145/2693208.2693233

Naur, P., & Randell, B. (1969). *Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th-11th october 1968.*

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign.

O'Regan, G. (2013). Ada Lovelace. In *Giants of Computing* (pp. 179–181). Springer.

Petersen, K., Gencel, C., Asghari, N., Baca, D., & Betz, S. (2014). Action research as a model for industry-academia collaboration in the software engineering context. *Proceedings of the 2014 International Workshop on Long-Term Industrial Collaboration on Software Engineering*, 55–62.

Pressman, R. S. (1997). *Software Engineering: A Practitioner's Approach* (Fourth).

Rajavat, A., & Tokekar, V. (2011). ReeRisk–A Decisional Risk Engineering Framework for Legacy System Rejuvenation through Reengineering. *International Conference on Advances in Communication, Network, and Computing*, 152–158.

Rashid, N., Salam, M., Sani, R. K. S., & Alam, F. (2013). Analysis of risks in re-engineering software systems. *International Journal of Computer Applications*, *73*(11).

Rosenberg, L. H., & Hyatt, L. E. (1996). Software re-engineering. *Software Assurance Technology Center*, 2–3.

Schwaber, K., & Sutherland, J. (2011). The scrum guide. *Scrum Alliance*, *21*(19), 1.

Seacord, R. C., Plakosh, D., & Lewis, G. A. (2003). *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional.

Singh, J., Singh, K., Sahib, F., Singh, J., & Rajpura, P. (2019). Reengineering framework to enhance the performance of existing software. *System, 1139*, 120–123.

Sommerville, I. (2000). *Software Re-engineering*. https://ifs.host.cs.st-andrews.ac.uk/Resources/Notes/Evolution/SWReeng.pdf

Sommerville, I. (2016). *Software engineering*. Pearson Education.

State of Agile Report. (2022).

Strawn, G. (2014). Alan Turing. *IT Professional*, *16*(1), 5–7. https://doi.org/10.1109/MITP.2014.2

Strawn, G., & Strawn, C. (2015). Grace Hopper: Compilers and Cobol. *IT Professional*, *17*(1), 62–64. https://doi.org/10.1109/MITP.2015.6

Tatnall, A., & Davey, B. (2016). Towards machine independence: From mechanically programmed devices to the internet of things. *IFIP International Conference on the History of Computing*, 87–100.

Walsham, G. (1995). Interpretive case studies in IS research: nature and method. *European Journal of Information Systems*, *4*(2), 74–81. https://doi.org/10.1057/ejis.1995.9

Wirth, N. (2008). A Brief History of Software Engineering. *IEEE Annals of the History of Computing*, *30*(3), 32–39. https://doi.org/10.1109/MAHC.2008.33

Wolff, E., & Johann, S. (2015). Technical Debt. *IEEE Software, 32*(04), 94-c3. https://doi.org/10.1109/MS.2015.95

Yin, R. K. (1989). *Case Study Research: Design and Methods*. Sage.

Yin, R. K. (2003). Case study research design and methods third edition. *Applied Social Research Methods Series*, *5*.

Yin, R. K. (2018). *Case study research and applications*. Sage.

Appendices

Appendix 1: Themes, sub-themes, and codes with explanation

Themes	Sub-themes	Codes	Code explanation
Technology	legacy technology	discontinued technology support	Technology (e.g. IDEs) might not be provided with updates and support anymore. Chances that it might not work in newer OS versions.
		struggle to integrate technologies	It might be difficult to integrate technologies because of the deprecated technology in the legacy application (e.g.: integrating legacy application into new IDE, using new technology in legacy application)
		legacy code	Legacy code might be difficult to understand and integrate.
		no replacement for technologies	Deprecated technologies might not be able be replaced (e.g. no similar technology to be replaced with)
		lack of old technology for developing old software	Lack of technology for developing software which needs outdated development environment (e.g.: 32-bit OS).
		fundamental change of technology	The same technology changed fundamentally too much over time, that it might not be able to simply be updated.
		lack of knowledge about old technology	Team members might have a lack of knowledge about old technology.
	insufficient technology	lack of testing environment	Technology for testing the application might not be available (e.g.: "clean" machines without development technology).
		missing version controlling	Version controlling might not be used for the legacy application.
		inability to roll back code changes	Code changes cannot be rolled back.

		code overheap for IDEs and support tools	Some code files might be too long for the IDE or support tools to process.
	legacy application	lack of legacy application documentation	The functionality of the application might not be well-documented.
		lack of readability of code	The code might not be readable due to a lack of naming conventions or an old coding style.
		limited number of people with knowledge about legacy application	Only a limited number of people might have knowledge about the legacy application.
		lack of commentary in code	The code might be difficult to understand due to a lack of comments in it.
		fix is worse than problem	A fix used during the re-engineering process might actually introduce more technical debt (e.g.: upgrading the technology requires a workaround).
		lack of knowledge about legacy application code	A lack of knowledge about the legacy application and its functionality.
Process	**testing**	missing test documentation	Missing test documentation might cause difficulties to the app thoroughly as team members might not know how certain test results should look like.
		lack of time for testing	Not enough time for testing might lead to miss bugs.
		lack of automated testing	Manual testing might take up a lot of time.
		lack of testing environment	Technology for testing the application might not be available (e.g.: "clean" machines without development technology).
		inconsistent testing	Insufficient testing after applying changes might cause that bugs are getting missed.
	time constraint	lack of time for testing	Not enough time for testing might lead to missing bugs.
		limited licensing time	Limited time with a licensed technology might cause time pressure.
		lack of time for meetings	A lack of time for meetings due to team member's limited time.

	lack of documentation	missing test documentation	Missing test documentation might cause difficulties to test the app thoroughly as team members might now know how certain test results should look like.
		lack of commentary in code	The code might be difficult to understand due to a lack of comments in it.
		lack of legacy application documentation	The functionality of the application might not be well-documented.
		lack of re-engineering documentation	The re-engineering process might not get documented properly. It might lead to confusion afterwards (e.g.: about decision making).
People	**lack of knowledge**	lack of knowledge about legacy application code	New developers lack knowledge about the legacy application and its functionality.
		lack of state-of-the-art technology knowledge	A lack of knowledge within the team about modern technology.
		limited number of people with knowledge about legacy application	Only a limited number of people might have knowledge about the legacy application.
		lack of knowledge about old technology	Team members might have a lack of knowledge about old technology.
		lack of commentary in code	The code might be difficult to understand due to a lack of comments in it.
	process engagement	confused stakeholders	Stakeholder might get confused about the process presented during the Sprint Review (e.g.: decision making). This could come from a too complex explanation or an unclear definition of done.
		not clarifying importance of task	Team members might become unmotivated if the importance of the task hasn't been clarified.
		inconsistent issue tracking	Issues might not be tracked when they arise.
		holding meetings incorrectly	Meetings might not be held correctly (e.g.: not sticking to the agenda, mixing different kinds of meetings).
		lack of time for meetings	A lack of time for meetings due to team member's limited time.

		no clear goal	The goal of the project might not be clear (e.g.: different parties might have different expectations; the goal is not set out properly)
		inconsistent testing	Insufficient testing after applying changes might cause that bugs are getting missed.
		inconsistent bug tracking	Bugs might not be tracked when they arise.
		inconsistent use of version controlling	Code changes might not be committed regularly.
		unwillingness to change	Unwillingness to change might prevent the progress of the project.
	methodology engagement	misevaluation of story points	The time it takes to execute a task might be over- or underestimated.
		too big tasks	Tasks might not be broken up enough.
		neglecting methodology	Team members might not stick to all processes of the used methodology.
		unclear definition of done	The definition of when a task is finished might not be clear.
		wrong prioritisation	Tasks might be prioritised wrong.
		poorly written user stories	The definition of a task might be unclear.
	social	unwillingness to change	Unwillingness to change might prevent the progress of the project.
		lack of face-to-face working	Some team members might want to work more in person with their colleagues.
		lack of support	A lack of support among team members.
		lack of a team	Not enough people working on the project.
		confused team members	Unclear communication might lead to confusion (e.g.: explanations).

		confused stakeholders	Stakeholder might get confused about the process presented during the Sprint Review (e.g.: decision making). This could come from a too complex explanation or an unclear definition of done.
		not clarifying importance of task	Team members might become unmotivated if the importance of the task hasn't been clarified.
		insecurity of team members	Team members might be insecure (e.g.: prevents them from asking questions).

Appendix 2: Codes with their expected and actual impact

Themes	Sub-themes	Codes	Expected Impact	Actual impact
Technology	legacy technology	discontinued technology support	high	high
		struggle to integrate technologies	high	high
		legacy code	high	high
		no replacement for technologies	high	medium
		lack of old technology for developing old software	low	medium
		fundamental change of technology	high	high

		lack of knowledge about old technology	medium	low
	insufficient technology	lack of testing environment	low	low
		missing version controlling	high	low
		inability to roll back code changes	high	low
		code overheap for IDEs and support tools	medium	medium
	legacy application	lack of legacy application documentation	high	high
		lack of readability of code	medium	medium
		limited number of people with knowledge about legacy application	high	high
		lack of commentary in code	medium	medium
		fix is worse than problem	high	high
		lack of knowledge about legacy application code	high	medium
Process	**testing**	missing test documentation	high	high
		lack of time for testing	high	none
		lack of automated testing	medium	medium
		lack of testing environment	low	low

		inconsistent testing	medium	medium
	time constraint	lack of time for testing	high	none
		limited licensing time	high	high
		lack of time for meetings	medium	high
	lack of documentation	missing test documentation	high	high
		lack of commentary in code	medium	medium
		lack of legacy application documentation	high	high
		lack of re-engineering documentation	low	low
People	lack of knowledge	lack of knowledge about legacy application code	high	medium
		lack of state-of-the-art technology knowledge	low	high
		limited number of people with knowledge about legacy application	high	high
		lack of knowledge about old technology	medium	low
		lack of commentary in code	medium	medium
	process engagement	confused stakeholders	low	low

		not clarifying importance of task	low	low
		inconsistent issue tracking	low	low
		holding meetings incorrectly	low	low
		lack of time for meetings	medium	high
		no clear goal	low	low
		inconsistent testing	medium	medium
		inconsistent bug tracking	low	low
		inconsistent use of version controlling	medium	medium
		unwillingness to change	high	medium
	methodology engagement	misevaluation of story points	low	medium
		too big tasks	low	medium
		neglecting methodology	low	low
		unclear definition of done	low	medium
		wrong prioritisation	medium	none
		poorly written user stories	low	medium
	social	unwillingness to change	high	medium
		lack of face-to-face working	low	medium

		lack of support	medium	low
		lack of a team	medium	high
		confused team members	low	low
		confused stakeholders	low	low
		not clarifying importance of task	low	low
		insecurity of team members	low	low

Appendix 3: Codes with their mitigation status

Themes	Sub-themes	Codes	Mitigation Status
Technology	legacy technology	discontinued technology support	mitigated
		struggle to integrate technologies	mitigated, deferred
		legacy code	partially mitigated
		no replacement for technologies	deferred
		lack of old technology for developing old software	mitigated

		fundamental change of technology	deferred
		lack of knowledge about old technology	mitigated
	insufficient technology	lack of testing environment	mitigated
		missing version controlling	mitigated
		inability to roll back code changes	mitigated
		code overheap for IDEs and support tools	deferred
	legacy application	lack of legacy application documentation	mitigated
		lack of readability of code	deferred
		limited number of people with knowledge about legacy application	partially mitigated
		lack of commentary in code	deferred
		fix is worse than problem	mitigated
		lack of knowledge about legacy application code	partially mitigated
Process	**testing**	missing test documentation	mitigated
		lack of time for testing	not occurred
		lack of automated testing	mitigated

		lack of testing environment	mitigated
		inconsistent testing	not mitigated
	time constraint	lack of time for testing	not occurred
		limited licensing time	deferred
		lack of time for meetings	not mitigated
	lack of documentation	missing test documentation	mitigated
		lack of commentary in code	deferred
		lack of legacy application documentation	mitigated
		lack of re-engineering documentation	partially mitigated
People	**lack of knowledge**	lack of knowledge about legacy application code	partially mitigated
		lack of state-of-the-art technology knowledge	mitigated
		limited number of people with knowledge about legacy application	partially mitigated
		lack of knowledge about old technology	mitigated
		lack of commentary in code	deferred

process engagement	confused stakeholders	mitigated
	not clarifying importance of task	mitigated
	inconsistent issue tracking	not mitigated
	holding meetings incorrectly	mitigated
	lack of time for meetings	not mitigated
	no clear goal	mitigated
	inconsistent testing	not mitigated
	inconsistent bug tracking	not mitigated
	inconsistent use of version controlling	not mitigated
	unwillingness to change	mitigated
methodology engagement	misevaluation of story points	mitigated
	too big tasks	mitigated
	neglecting methodology	mitigated
	unclear definition of done	mitigated
	wrong prioritisation	not occurred

		poorly written user stories	mitigated
	social	unwillingness to change	mitigated
		lack of face-to-face working	not mitigated
		lack of support	not mitigated
		lack of a team	not mitigated
		confused team members	not mitigated
		confused stakeholders	mitigated
		not clarifying importance of task	mitigated
		insecurity of team members	mitigated