FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards Agile Requirements Management at Vodafone: a Reverse Engineering Approach

Maria Gonçalves Caldeira

Master in Informatics and Computing Engineering

Supervisor: Professor António Lucas Soares [FEUP]

Second Supervisor: Professor Ademar Aguiar [FEUP]

Second Supervisor: Luís Moreira [Vodafone]

November 2, 2022

Towards Agile Requirements Management at Vodafone: a Reverse Engineering Approach

Maria Gonçalves Caldeira

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Prof. João Carlos Pascoal Faria

External Examiner: Prof. Ângelo Martins

Supervisor: Prof. Ademar Manuel Teixeira de Aguiar

November 2, 2022

Abstract

The need to change an already finished software system is not something new. It has happened since the implementation of the first programs, and it will continue to happen because we are in an ever-changing world, and software systems must be constantly adapted to this-worldliness.

Software Reverse Engineering (SRE) is a process with extreme relevance in the software engineering world. Its ability to extract any information from a software system to understand how it works is used nowadays to improve code implementation, interoperability, cybersecurity, etc. However, it can be challenging and time-consuming because it is mainly based on human comprehension, which may be compromised due to the lack of documentation required.

As it happens in a typical *Software Development Life Cycle* (SDLC), in a *Software Reverse Engineering* process there is also a need for all the requirements to be documented, managed, and tested, as we want to be sure that everything is working as it should. This need to manage the requirements is derived from the possibility of arising new requirements or changes in the already existing ones since a product that goes through a *Reverse Engineering* (RE) process is often a product that needs to evolve, and that need can bring changes in its final functionalities. The lack of proper *Requirements Management*, documentation, and *Software Testing* can affect many roles, reflected in the final product not meeting expectations, possibly resulting in the project's failure.

Vodafone uses a Scrum methodology in some of its projects since they want to deliver the best product possible in the shortest amount of time. However, issues like lack of project documentation, requirements management and specification, and software testing may negatively impact the SDLC of some of their projects, and they want to avoid this as much as possible. In software development teams it is seen as essential that all requirements are specified and validated throughout the development cycle to guarantee the *Product Owner*'s (PO) satisfaction and consequent success of the final product, even while facing a RE process.

This work aims to mitigate some of these issues and to document a plan to improve a *Reverse Engineering* process at Vodafone. This was achieved by refining some *Re-engineering* practices and adapting them to Vodafone's reality, together with thorough management of the requirements using a *BDD approach*. The tools used included *Gitlab* for the requirements management and the usage of the *Gherkin* Specification Language for specifying them, while the *Cucumber* tool executes automated acceptance tests written by Gherkin.

We tested and validated the process and tools by implementing the plan described above in a Vodafone's project to evaluate its benefits and impact on the overall life cycle of software systems development at Vodafone. We concluded that the constructed plan and the usage of Gherkin and Cucumber are relevant and interesting and can benefit some of Vodafone's projects and development teams.

Resumo

A necessidade de mudar um sistema de software já terminado não é algo novo. Já acontece desde a implementação dos primeiros programas, e vai continuar a acontecer porque estamos num mundo em constante mudança e os sistemas de software devem adaptar-se constantemente a esta mundanidade.

Software Reverse Engineering (SRE) é um processo com extrema relevância no mundo da engenharia de software. A sua habilidade de extrair qualquer informação de um sistema de software para perceber como ele funciona é usada hoje em dia para melhorar implementação de código, cibersegurança, etc. Contudo, ele pode ser desafiante e demorado porque se baseia principalmente na compreensão humana, que pode ser comprometida devido à falta de documentação necessária.

Como acontece num típico *Software Development Life Cycle* (SDLC), num processo de *Software Reverse Engineering* também há a necessidade de todos os requisitos serem documentados, geridos e testados, pois queremos ter a certeza de que tudo funciona como deveria. Esta necessidade de gerir os requisitos deriva da possibilidade de surgirem novos requisitos ou mudanças naqueles já existentes, já que um produto que passa por um processo de *Reverse Engineering* (RE) é na maior parte das vezes um produto que precisa de evoluir, e essa necessidade pode trazer mudanças nas suas funcionalidades finais. A falta de uma *Requirements Management* adequada, de documentação e de *Software Testing* pode afetar muitos papéis, refletido no final produto não corresponder às expectativas, resultando possivelmente no fracasso do projeto.

A Vodafone usa uma metodologia Scrum em alguns dos seus projetos, já que eles querem entregar o melhor produto possível no mínimo espaço de tempo. Contudo, problemas como a falta de documentação do projeto, de gestão de requisitos e especificação, e de software testing podem impactar negativamente o SDLC de alguns dos seus projetos, e por isso eles querem tentar evitar ao máximo essas situações. Em equipas de desenvolvimento de software, é visto como essencial que todos os requisitos sejam especificados e validados durante todo o ciclo de desenvolvimento para garantir a satisfação do *Product Owner*'s (PO) e consequente sucesso do produto final, mesmo enfrentando em processo de RE.

Este trabalho pretende atenuar alguns destes problemas e documentar um plano para melhorar um processo de *Reverse Engineering* na Vodafone. Isto foi conseguido através da refinação de algumas práticas de *Re-engineering* e adaptando-as à realidade da Vodafone, com a minuciosa gestão dos requisitos usando uma abordagem *BDD*. As ferramentas usadas incluem o *Gitlab* para a gestão dos requisitos e o uso da linguagem de especificação *Gherkin* para os especificar, enquanto a ferramenta do *Cucumber* executa testes de aceitação automatizados escritos pelo *Gherkin*.

Testamos e validamos o processo e as ferramentas ao implementar o plano acima descrito num projeto da Vodafone para avaliar os seus benefícios e o impacto em todo o ciclo de vida de um sistema de software implementado na Vodafone. Concluímos que o plano construído e o uso do *Gherkin* e do *Cucumber* é relevante e interessante e pode ser benéfico para projetos da Vodafone e equipas de desenvolvimento.

Acknowledgements

I want to start by expressing my deep gratitude to Vodafone for embracing this dissertation and providing the needed environment for its realization and to my supervisors, António Lucas Soares and Ademar Aguiar, for guiding and advising me throughout this entire journey.

I also want to express my gratitude to my friends Ana Silva and Margarida Pinho for always standing by my side these last five years and making my college life easier and happier. A big thank you to Sofia Lajes for all the precious moments that we lived together. My college experience wouldn't be the same without you, and you know it. On this note, I couldn't forget to thank my great friend Carlos Duarte for being my greatest support, besides my supervisors, throughout the development of this dissertation. Thank you for all the time you spent hearing my doubts and ideas and for all the motivation you gave me. You're the reason this dissertation exists in the first place, so none of this would be possible without you.

A special thanks to my boyfriend Artur Santos for being my "rock" over these past few years. Thank you for always being there for me in the most challenging times and encouraging me not to give up.

Finally, an immense appreciation and thank you to my family, who always encouraged and supported me, especially my mother Paula Gonçalves, my father Fernando Caldeira, and my brother João Caldeira. One more thank you to my parents for all the patience you had with me throughout the years. Without you and your support, I wouldn't be able to achieve any of this and would definitely not be the accomplished person I am today.

Maria Gonçalves Caldeira

"All you need in this life is ignorance and confidence;
then success is sure."

Mark Twain

Contents

List of Figures

List of Tables

Abbreviations

SDLC	Software Development Life Cycle
RQE	Requirements Engineering
RM	Requirements Management
PO	Product Owner
VV	Validation and Verification
BDD	Behavior-Driven Development
TDD	Test Driven Development
RE	Reverse Engineering
SRE	Software Reverse Engineering
ST	Scrum Team
PB	Product Backlog
AAT	Automated Acceptance Testing
LS	Legacy System
US	User-Story

Chapter 1

Introduction

Contents

This first chapter has the purpose to introduce the general context, motivation and goals of this dissertation. A brief summary of the problem faced is also given. At the end of it, the structure of this document is also specified and detailed.

1.1 Context

The main goal of any software development process is to deliver the highest quality and lowest cost product in the shortest possible amount of time. It is crucial for the final product's success that the best management and maintenance of the development process are ensured, since these aspects will then be reflected in the degree of efficiency of the process. Therefore, it is necessary to ensure that appropriate measures and techniques are used to raise development efficiency to its full potential.

At Vodafone, some of the development teams use an Agile development framework, called Scrum, to help manage their teams and improve the efficiency of the development process. Scrum is already well-known worldwide, and multiple development teams use it. Its flexible nature and easy implementation make Scrum the most used agile framework globally [56].

In Scrum, the *Product Owner* (PO) is the person responsible for maximizing the value of the product that will be developed by the *Development Team*. He is responsible for specifying and

communicating the product requirements to the *Scrum Team* and managing them in the *Product Backlog* (PB) [5]. A final product's success depends on the *Product Owner* satisfaction with it. Logically, if the requirements he imposed aren't met, he won't be satisfied, which might reflect in the failure of the final product.

Sometimes, some systems need to go through a *Reverse Engineering* (RE) process. A RE process is required when a software system is outdated and needs to be maintained. Its goal is to find a less expensive way to create a similar actualized product that fulfills the current needs of its users, and it does that by analyzing and understanding what already exists in the system and how it works.

While facing a project that will need to go through a *Reverse Engineering* process, Vodafone's teams will need more than just an Agile framework, like Scrum, to help them achieve the level of efficiency expected since the RE process can be really difficult and time-consuming [33]. Some *Re-engineering* practices will have to be adopted to ensure the quality required for software development at Vodafone, as well as some tactics and tools to specify and manage some requirements (old or new) that may arise in the process [25].

A *Software Re-engineering* process consists in three main phases: the *Software Reverse Engineering* (SRE) phase, where all existent artifacts are gathered (code, documentation, requirements, etc); the changes phase; and the Forward Engineering phase, where the new code implementations/changes are made. In *Software Reverse Engineering*, like in a typical *Software Development Life Cycle* (SDLC), the specification of project requirements is essential to guarantee the final product's success. In a SRE process, the development team needs to know the requirements that already exist so that they know which functionalities have already been implemented and the new requirements that may arise following the need for product evolution.

Not only are requirements essential, but also their management since lousy management of the requirements during any development process can be fatal to the final product's success. In a *Software Reverse Engineering* process, in the event of absolute scarcity of documented requirements, it's crucial to manage in the best way possible the requirements that will be gathered up by the study and analysis of other project artifacts, as well as new ones that may arise.

Vodafone's development teams have the perception that some product failures worldwide are due to the final product not meeting the requirements proposed by the *Product Owner* [53]. In generic terms, this may happen because, although people usually give more importance to designing and development, the *Requirements Engineering* phase of a *Software Development Life Cycle* (SDLC) is one of the most critical part of software development [53].

This product failure problem is emphasized when dealing with a *Reverse Engineering* project where documentation, including documented requirements, is scarce. There is a need to understand the best way for a Vodafone's team involved in a RE process to gather the project requirements and manage them. In addition, there is also a need to understand if those requirements are already implemented or not. That can only be achieved by testing the code.

Vodafone wants to avoid those product failure situations to the fullest since they understand the inconveniences this could bring to the company in terms of costs.

Furthermore, software developers teams also typically include junior developers, therefore any method that facilitates/accelerates their ramp-up to become closer to the Product owner's goals and true requirements is a valid target to chase. It is necessary to educate them so that the project's efficiency is not impaired, and the best way to do so is to use tools that will make their learning more accessible.

In this way, it is necessary to arrange a plan to improve the efficiency of a Reverse Engineering process, while refining some re-engineering practices and adapting them to Vodafone's reality. In this dissertation, a *Behavior Driven Development* (BDD) approach using Gherkin, Cucumber, and Gitlab's help, is proposed and experimented.

1.2 Motivation And Goals

As said in the previous section, increasing the efficiency of the overall development process should always be a priority for any software development company. Improving the development efficiency will not only reduce the time required for a product to be complete, but it will also reduce the costs that the company has to spend on its production.

Despite being vital for a company, improving efficiency won't directly translate into the final product's success. For its success, the satisfaction of the *Product Owner* and the Stakeholders has to be guaranteed. Stakeholders and *Product Owner's* satisfaction mainly derives from the final product meeting or not the expected results.

It is possible to have an efficient development process that, in the end, doesn't meet the stakeholders' expectations and vice-versa [30]. Still, the ideal is to always have both in order to achieve success with the final product, in terms of product quality and costs for the company.

This dissertation aims to address these two essential points: increasing efficiency while also increasing the odds of success of the final product. To this end, the goal is to develop a plan for that in the case of a *Software Reverse Engineering* process.

This plan addresses the possibility of the non-existence of documented requirements, as well as the possibility of new changes in the current requirements and the emergence of new ones following the evolution of the product. Furthermore, it also focuses on *Software Testing* as a way of validating if the requirements stipulated by the PO are being met, even if the old ones suffer changes and new ones emerge.

Software Testing is an integral and mandatory part of the entire development process at Vodafone. However, there are teams actively looking for solutions that help them improve the efficiency and effectiveness of these tests, with test automation being one of these vectors. The plan developed in this dissertation will also try to introduce new automated test methods to the development teams.

Through an approach with the usage of an almost-natural language framework like Gherkin for requirements specification, a tool like Cucumber for testing and a version control system (Gitlab) to help with *requirements management*, a documented plan for a Reverse Engineering process and subsequent requirements management will be implemented.

In addition, the last goal of this dissertation is to bring new automated *Software Testing* methods to Vodafone's development teams.

1.3 Problem Introduction

At Vodafone, some software development teams use Scrum. Scrum is focused on the collaboration between team members and on improving the development process efficiency [5].

However, a framework like Scrum is insufficient to ensure the proper functioning of a *Software Reverse Engineering* process, especially when there are little to no documented artifacts besides the implemented code.

At a big company like Vodafone, most products will face a need for evolution, as the client's needs change over time. *Reverse Engineering* processes are essential for Vodafone products to be able to keep up with current and future customer needs.

Since these processes are essential, there is a need to improve them as much as possible. A *Software Reverse Engineering* process can't take too long to be finished because customer needs change daily, so there is really no space for delays. This becomes an issue when a product needs to go through a RE process, and there is not much documentation to start with, not even specified requirements.

The time that a development team loses in studying, analyzing, and understanding what already exists is time that they are wasting on delivering the product suited to the customer's new needs.

In a competitive market as the one in which Vodafone falls, Vodafone cannot afford to deliver new or updated products after its competitors, so making sure that any development process runs as fast and smoothly as possible is crucial, as well as making sure they are delivering what the customers need at the moment.

1.4 Project

As already mentioned, this dissertation is being done in the context of a business environment at Vodafone. The team and project I will be working on will also host another dissertation.

That other dissertation and mine will complement each other since combining the two will contribute to an efficiency increase throughout every stages of a development process, using tools like Gherkin and Cucumber.

While my dissertation focuses on ensuring that the Product Owner is satisfied with the final product, the other is all about bringing the management and traceability visualization of the requirements to the development environment for the developers to see.

The focus of my dissertation is more on the Product Owner side, and the direction of the other work is more on the Developers' side. As we work with a Scrum framework, these different focuses will ensure that all of the different phases of a Scrum plan/sprint will be covered.

Together, the two dissertations will aim to evaluate the advantages of the tools used to manage, trace and validate constantly changing requirements, such as the ones in a RE process, in a large-scale company such as Vodafone.

1.5 Dissertation Structure

The first chapter of this dissertation (Chapter 1) tries to introduce and transmit the reason for its realization. Software Reverse Engineering processes are increasingly relevant nowadays, so Vodafone would benefit from implementing a plan that will benefit requirements specification and management while creating an automated test environment.

In the next chapter, the State of the Art (Chapter 2) forms a compilation of the research of the main concepts argued in this dissertation.

The next chapter, the Problem chapter (3), presents a more detailed description of what has already been addressed in Section 1.1 and Section 1.3, the context and the existent problem. While the problem is deeply analyzed, some assumptions made and the methodology are also specified.

In Chapter 4, the developed solution is detailed. All the different steps of the implemented RE plan are specified and explained, as well as the context of the project that had to go through a RE process.

In Chapter 5, the solution is validated. It specified how the validation process was conducted and analyzed the results.

Finally, the Conclusions chapter (6) summarizes all the work developed and its main conclusions, as well as possible future work.

Chapter 2

State of the art

Contents

This chapter will give an overview of the relevant background for this dissertation and related work, starting by addressing the concept of a *Software Reverse Engineering* process 2.1.

2.1 Software Reverse Engineering Process

Re-engineering and *Reverse Engineering* (RE) are two concepts that most of the time are confused or are taken to be the same thing [22].

While *Reverse Engineering* is the process of fully understanding and analyzing in detail what is happening and what exists in a legacy system, creating representations of it at a higher level of

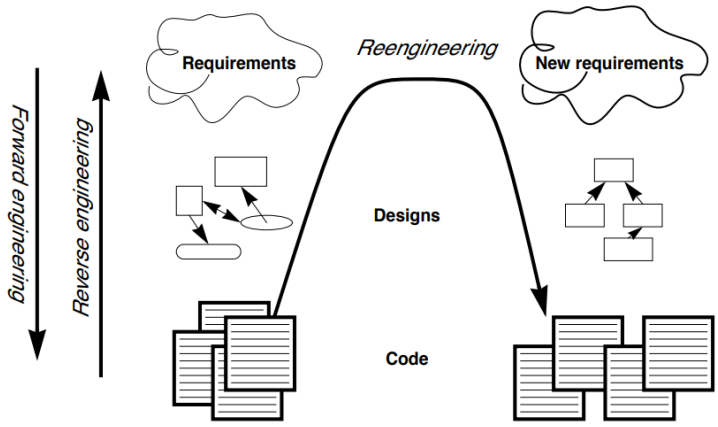

Figure 2.1: Forward Engineering, Reverse Engineering and Re-engineering [22]

abstraction, *Re-engineering* is the process that re-structures a system for it to be able to further extent development-wise [22].

Associated to these processes, there's also the definition of *Forward Engineering*. *Forward Engineering* is the traditional way of moving forward in a development process, moving from high-level abstractions to a concrete implementation of features [22].

In figure 2.1, an illustration of the relationship between these three concepts can be seen. *Re-engineering* is a combination of the *Reverse Engineering*, alterations phase, and *Forward Engineering* process, and *Reverse Engineering* is the very first part of the *Re-engineering* process.

In *Software Reverse Engineering* (SRE), there is a need to recapture most things of a legacy system, such as the requirements, the design, structure, and content [46]. With the help of the development artifacts and their analyzes, SRE aims to recover lost information while defining higher abstractions of the product, which will later facilitate its reuse [46].

A SRE process usually begins with extracting the requirements and design from the existing development artifacts. This procedure is illustrated in figure 2.2.

To extract the requirements and design, it's used a technique called *slicing*, or *code isolation*. *Slicing* is a way of locating the minimum amount of data and code related to a particular feature of the product [46]. While this concept is theoretically really simple, it is much more complicated and time-consuming than it seems, even with the help of tools that try to facilitate it.

To mitigate this hardships, there are currently three approaches to *slicing*: *conditional-based*, *forward*, and *backward* [46]. All these approaches describe different ways of analyzing the source code to understand what exists in the system. However, because of the process's high complexity, these approaches are ineffective in facilitating it.

2.2 Software Re-Engineering

While talking about *Reverse Engineering*, approaching the *Re-engineering* concept is a must.

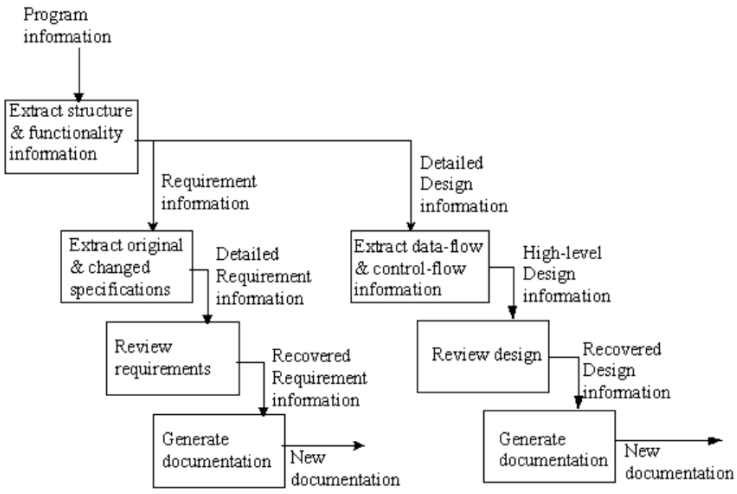

Figure 2.2: Software Reverse Engineering procedure from [46]

Software Re-engineering is the process of analyzing, examining, and studying an already existing software system and subsequent updating of that system so that it gains a new form [46]. It's a process that takes on existing valuable legacy software and redoes it while adapting it to current technologies [46], contributing to the improvement of the maintainability of a software system.

A *legacy software* is seen as a kind of software developed using programming languages or old methods that are already outdated [22]. However, it doesn't necessarily mean that the software is old. The *software engineering* world is constantly finding ways of upgrading development tools and methods, so it's very easy for a software system to get quickly outdated and become a legacy software [22].

Even though they are outdated, *legacy software systems* are valuable [22]. This means that it would be a waste not to use them anymore. The goal of *Software Re-engineering* is precisely to make sure that it's possible to update legacy software systems at an affordable cost [22].

The difficulty of a *Re-engineering* process relies on the lack of existing documented artifacts. A *software artifact* helps document and describe software functionalities, as well as its architecture and design. The implemented code, requirements, diagrams, and other software documentation are some examples of software artifacts. Often, those artifacts are no longer available at the start of a *Re-engineering* process or are simply outdated [46].

The first step to recognize that a software system has to undergo a re-engineering process is to identify that there's a legacy issue [22]. In Table 2.1, a brief summary of the warning signs that a product probably needs to go through a *Re-engineering* process and respective brief descriptions are shown. If any of these signs are detected in a legacy system, the system needs to go through a *Re-engineering* process to be functional again.

While the *Re-engineering* process helps with the maintainability of a system, it doesn't come without risks. Some of those risks include lack and inadequate management of the Re-engineering

process, high manual costs, lack of quality assurance and metrics program, too much documentation produced, among others [46].

It is important to make an evaluation and estimation of those potential risks before the start of any Re-engineering process. That way, it's possible to determine which Re-engineering approach is better to use.

Warning Signs	Description
Obsolete or no documentation	Sign of problems from the moment the original developers leave the project
Missing tests	Sign the system will evolve with high risk or cost
Original developers or users have left	Deteriorates in a poor documented software, unless it is already well documented with good test coverage
Inside knowledge about system has disappeared	Documentation is out of sync with the existing code. Nobody knows anymore what is really happening
Limited understanding of the entire system	Almost nobody has a good overview of what is going on
Too long to turn things over to production	If you don't know how to deal with the difficulties in the development process, it will only get worse
Too much time to make simple changes	When a system is so complex that now is even difficult to make small changes
Need for constant bug fixes	Every-time a new bug gets fixed a new one appears
Maintenance Dependencies	Every-time a new bug gets fixed in a certain *place*, a new one appears *somewhere else*
Big build times	Long recompilation times slow down the ability to make changes
Difficulties separating products	Architecture no longer right for the job, if there's a difficulty tailoring releases for each customer
Duplicated code	If not eliminated by refactoring maintenance becomes a nightmare
Code Smells	Sign that a system has been repeatedly expanded and adapted without having been re-engineered

Table 2.1: *Re-engineering warning signs* [22]

2.2.1 Approaches

In [46] and in [40], the authors discuss three different approaches to Re-engineering: the *Big Bang Approach*, the *Incremental Approach*, and the *Evolutionary Approach*.

The *Big Bang approach* consists in replacing the entire system at one time. This is a very high-risk approach since the old system has to work in parallel with the new system to assure that everything is working, and it can be too much resource consuming and time-consuming for larger systems [46, 40].

In the *Incremental approach*, parts of the system are re-engineered one by one and added, in an incremental way, as new versions until all the needs are met. It is lower in risks than the *Big Bang approach*, but the fact that it takes too much time until the entire process is complete can be seen as a disadvantage [46, 40].

Finally, the *Evolutionary Approach* suggests a similar method to the *Incremental approach*, but in this case, the parts of the "old" system that are added as a new version to the "new" system are chosen based on functionalities and not on structure [46, 40].

The *Re-engineering* process faces challenges that can be mitigated with the help of some tools, but without a well-defined core, the usage of those tools is as good as nothing.

2.2.2 Phases

When it comes to the core process of *Re-engineering*, there is one that every development team should follow, according to the authors of [46]. The process can be divided into five different phases, starting in the *Team Formation* phase. This team will be formed by people with a reasonable amount of knowledge in management tactics for Re-engineering processes, as well as understanding more than the basics to define the best Re-engineering plan possible for that specific situation [46].

The Re-engineering team will then move on to the *Project Feasibility Analysis* phase. Here, the team will evaluate the needs and goals that the already existing products meet and make analyses of the costs [46].

The next step is the *Analysis and Planning* phase. In this phase, the re-engineering team analyzes the legacy system, specifies the new system features, and creates a validation method to validate everything [46].

The last two phases are the *Implementation* phase, followed by the *Testing and Transition*. In the *Implementation*, the new changes and new features are developed, and in the *Testing and Transition* they are validated to ensure that nothing went wrong in the *Implementation* phase [46].

In [52], it is proposed a *re-engineering life cycle* consisting in six phases. Those phases can be visualized in figure 2.3.

That life cycle disregards the first step defended in [46], the *Team Formation* phase. Instead, it states that a Re-engineering life cycle should start directly on the *Requirements Analysis Phase*. This phase and the two that follow, *Model and Source Code Analysis*, are equivalent to the *Project Feasibility Analysis* and *Analysis and Planning* phases argued in [46]. However, while in [46] the analysis of the artifacts should be done simultaneously, in [52], the requirements analysis must be realized first, followed by the model and code analysis, respectively. The last steps of both models are also equivalent.

While the division of the two models is slightly different from each other, they operate in the exact same way. This allows us to validate the crucial steps of a *Re-engineering* process: identifying and analyzing in detail what already exists, implementing the new changes next, and validating the system in the end.

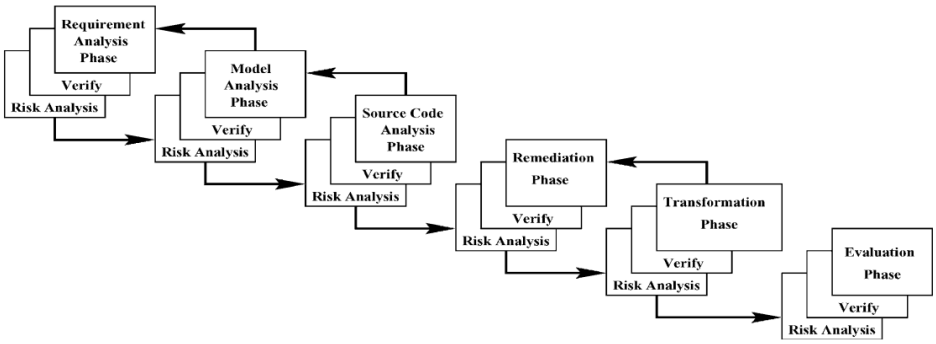

Figure 2.3: Re-engineering Life Cycle from [52]

2.3 Software Requirements

A *Software Requirement* is a description/specification of what should be implemented in a software development process [55]. They incorporate both the customer's vision and more technical aspects proposed by the development team [55].

There are multiple types of requirements. Based on [55], in Table 2.2 those various types are listed.

Software Requirements are specified in the *Requirements Engineering* process. *Requirements Engineering* is the process of defining and maintaining requirements, while also documenting them. These requirements that need to be defined and maintained are basically needs, of functionality and features, that a software development product has to fulfill [39].

It is one of the most complicated processes when it comes to software development. As stated by Fred Brooks [17], the most challenging part of starting to build a software system is deciding what we want and what has to be built. The difficulty of this is due to the significant struggle of establishing all the technical requirements that will be needed for the final product to work correctly. If the proper requirements aren't well specified at the beginning of the development process, it will be much more difficult and costly to rectify them later on.

That is why it is so important to collect all the correct information from the stakeholders right from the beginning. Not only that, but it is also necessary to collect them in the right way. Knowing what questions to ask the stakeholders is crucial because sometimes they may not be aware of specific requirements that can be useful to them. Understanding the best way to approach and communicate with the stakeholders is also essential. The way that a requirements engineer interacts with the stakeholders should make them as comfortable as possible to put forward ideas, ask

Types	Brief Description
Business Requirement	High-level business objective
Business rule	Definition or constraint to some business aspects. The origin of multiple software requirements
Constraint	An imposed restriction to the product development
External Interface Requirement	Software system and user connection
Feature	System functionalities that the user wants implemented. Described by functional requirements
Functional Requirement	System's behaviour, under specific conditions, description
Nonfunctional Requirements	Description of system's characteristic or constraint
System Requirement	Requirement for products that have multiple subsystems
User Requirement	Desired tasks that the users must be able to perform in a system

Table 2.2: *Requirement types* [55]

questions and give opinions that allow the engineer to collect the best and most detailed information possible.

This phase of the *Requirements Engineering* process, where the main goal is to gather the requirements from the stakeholders, is called the Elicitation phase. There are numerous possible ways to elicit requirements, such as *Stakeholder Analysis*, *Brainstorming*, *Interviews*, *Prototyping*, etc. Studies [21] have shown that the *Interviews* method is the most used and is also the most effective elicitation method. In this technique, the interviewer (the requirements engineer/business analyst) asks questions to the stakeholders to obtain information about what they visualize and need in the product that will be developed. For this method to work, the involvement of all participants is fundamental, and that is why it is necessary to establish a good relationship and communication with all elements.

Requirements elicitation, as already seen, is significantly based on the relationship between the requirements engineer and the stakeholders. This relationship, as well as all the requirements engineering process, lasts throughout the entire development process. For the satisfaction of the stakeholders with the final product, it is necessary to include them, as much as possible, through the development of the product, since they can change their mind along the way and decide that what they wanted, in the beginning, is not exactly what they want at the moment.

For this, it is also vital that they feel comfortable expressing what they want. If they change their ideas and expectations for the final product, it is also necessary that the development team knows how to manage and trace these requirements changes.

2.4 Software Requirements Tools

Software Requirements Management is critical for the success of a software development final product and its lifecycle. *Requirements management* and traceability assist many essential activities for software engineering, such as validating the requirements [48]. To do that, it is necessary to use proper tools and techniques that allow a good management of the requirements and guarantee their traceability.

At the moment, there are already several tools available to manage and trace requirements [48]. One of them is *DOORS* (Dynamic Object-Oriented Requirements System) [11]. *DOORS* is one of the requirements management tools used nowadays, and it has already proved to help reduce costs and increase efficiency. It is an application that runs on Windows and Linux systems and has multiple features that help specify and manage requirements. Its capability of providing an easy link between the different requirements, and respective test cases and test plans, facilitates the communication between them and their verification. This ease of use makes *DOORS* one of the best tools to help with requirements management in case of changes in the requirements.

Developed by the same company that acquired *DOORS*, *Rational RequisitePro* is also a requirement management tool. Similar to *DOORS*, this tool also stands out for its ease of use while promoting communication and collaboration between all software development team members. It also offers the power of a database and a word processor, thanks to the optional dynamic link to Microsoft Word, which adds more organization and management power to its capabilities.

While *DOORS* and *Rational RequisitePro* are more focused on managing the requirements, tools like *RETRO* and *TRAM* are more concentrated on traceability. *RETRO* uses IR (Information Retrieval) methods to trace requirements [29], while *TRAM* uses document templates based on an informational model that captures requirements, architecture, and traceability [27]. Another requirements traceability tool is *DesignTrack*. It is a prototype application that was developed to "investigate requirement traceability capability within computer-aided architectural design" [44].

Cradle is also a good contender on the tools more focused on *requirements management*. One of its advantages is that it guarantees the integration of all the software lifecycle, whether the project is small or of a large scale. Besides that, it also considers information security and strategies [1]. *Jama Software* is another great tool for requirements management. One of the key features of this software is live traceability [4]. All the requirements and the relationships between them can be easily navigated in order to understand the impact that changes can have on the development process. It also prioritizes collaboration between team members, providing an environment of real-time collaboration.

Visure is another management tool that can integrate other tools that a team already uses, such as *MS Word*, *Excel*, *Jira*, *DOORS*, etc. Like other tools, it also guarantees management and traceability throughout the development lifecycle, from conception to deployment [10]. Like *Visure*, *Orcanos* can import/export from *MS Word* and *Excel* and guarantees management and traceability throughout the development lifecycle. With *Orcanos*, it is also possible to automate the risk management system of a development project.

Other tools like *ReqSuite RM*, *Doc Sheets*, and *Pearls*, also share some of the features already mentioned, such as better efficiency, automated process, team collaboration, etc. These are also widely used tools nowadays, which greatly benefit the management and traceability of requirements in a software development cycle.

It was already proven that one of the biggest causes of software product failure relies on a bad management of changes in requirements [32]. To get to the bottom of the question, it is necessary to understand what goes wrong in the management of changes in order to develop new techniques and methods, or improve the ones that already exist.

According to Shalinka Jayatilleke and Richard Lai [32], there are currently multiple methods and techniques used for addressing and managing requirements change problems. One of them is using semi-formal management requirements change methods. All the semi-formal methods talked about in the article [32] are based on academic works and all of them consist in a sequence of steps to follow, to achieve the best requirements change management possible.

2.4.1 Semi-formal Requirements Change Management Methods

The first one is the Leffingwell and Widrig proposal [37]. This proposal first recognizes the possibility of requirements changes, and then it is made a plan for the eventuality of that happening. Then, it is made some kind of control step, in which the initial requirements are compared to the proposed requirements changes, to see if such changes are feasible. The third step defines a person or a group of people responsible for controlling the changes, and the fourth step establishes a control system where the changes can be captured and then managed in a hierarchical form. This proposal describes what needs to be done when handling requirements changes, but it fails to explain the conditions a change must have to be necessary.

The second proposal [24] starts by making an initial evaluation of the changes management while considering the necessary requirements and the opinions of the stakeholders, customers, etc. If a change request fits the requirements essential necessities, a proposal for that change to happen will be made. After that, a plan is made to analyze the change proposal and develop potential solutions for it. In this step, one of the potential solutions is chosen, but it isn't clear what are the parameters of choice for that. Then, if approved, another plan will be made to analyze the impact of this change on the necessary requirements in more detail. If this last stage is approved, then the change will be applied.

Kotonya and Sommerville's proposal [35] describes a three-step process to manage requirements changes successfully. The proposal identifies potential problems with a specific requirement or multiple requirements. These requirements are then analyzed, and at the second step, the implications of the changes are also examined, especially in monetary and time costs estimation. After that, the changes are implemented and at the end the new requirements are validated by a quality checking procedure.

Strens and Sugden suggest a five step proposal [51]. The first step consists of identifying the reasons for the need for change. Next, it is necessary to identify the requirements that will be affected by these changes and analyze the implications that these changes will bring. At the fourth

step, a change analysis is directed to other requirements, costs, design, performance, reliability, etc. Based on the impact and change analysis, the fifth and final step is to decide if the changes will be applied and managed. However, the decision terms of implementing the changes are not that well defined. The weight of each change analysis parameter (other requirements, costs, design, etc) isn't specified in the final decision-making.

The fifth proposal [45] has a process model focused on requirements elicitation and documentation, their verification and validation, and management. It starts by tracking the changes on the requirements, identifying, right after, the links between the requirements that will suffer the changes and the rest of the system. While recognizing the connections between requirements and system, their dependencies will also be identified. Then, a decision to accept or not the changes will be made. If the decision is favorable, the change request will be validated, and the respected changes will be tracked.

In the second to last proposal, Tomyim and Pohthong propose a method based on the use of UMLs [54]. Unified Modeling Language (UML) is a modeling language that defines a way to visualize the way a system has been designed [26]. The model of this proposal consists of a system procedure - operations to be carried out throughout the process - and work instructions - explanation of every task, step by step. At first, like other proposals, the change request is identified. After its identification, the related system procedure and work instructions are also identified. In the end, an analysis of the impact of the changes is made, and a decision is made based on that impact.

Finally, the last proposal gives some attention to ways of managing informal requirements changes [31]. Once again, it starts by identifying the changes that, in this case, are informal, and an analysis of the possible impact is carried out. Since these changes are based on what happens internally in the development process, the stakeholders don't really know much about it. So, it becomes necessary to discuss the changes with them. If the changes are approved, the only thing left to do is to decide when, in the development process stage, to include the changes.

2.4.2 Formal Requirements Change Management Models

All these proposals discussed above are only that, just simple proposals. They're not formalized models used for requirements change management [32]. Some examples of formal models that are currently used for requirements change management are the Leffingwell and Widrig model, Olsen, V-Like, Ince's, Spiral, NRM, Bohner, CHAM, Ajila, Lock and Kotonya.

To better contextualize these models, their main management activities that each focuses on and some of their possible limitations will be introduced. Requirements change management activities are the actions that happen during a requirements change management process that define clear objectives of each model [32].

Leffingwell and Widrig model [37] can make a plan for the supposed changes and analyze the impact they can have on developing a product and the effect in terms of costs. However, it doesn't consider the implementation of the requirements changes and, therefore, the verification of the new requirements isn't possible.

The Oslen model [43] is the opposite of Leffingwell and Widrig model. It only offers the possibility of implementing the changes and verifying them. It is a simple model that doesn't guarantee full tracking of the requirements because of a lack of documentation.

V-Like [41] is more focused on understanding the problem, analyzing the impact of the changes on the functionalities and the solution, implementing the changes and validating them. It only misses the analysis of the impact on the cost.

Ince's model [41], like Oslen, guarantees the implementation of the changes, but it validates them instead of verifying them like in Oslen's. On the other hand, the Spiral [41] centers its activities on understanding the problem and analyzing the possible solution. Another model that is very much similar to Ince's, is NRM [34]. The advantage of NRM is that it also includes a verification of the changed requirements.

The Bohner model [19] relies primarily on understanding the problem in cause and then applying the changes. It doesn't make an analysis of the impacts, though.

Ajila [12] is similar to V-Like. It only lacks the estimation of costs. The last two, CHAM [36] and Lock and Kotonya [38], are the two models most complete in terms of the activities they offer. The only problem with them is that CHAM doesn't analyze the impact, and the Lock and Kotonya model doesn't consider identifying changes.

2.4.3 Requirements Change Management Techniques

By the formal models mentioned above, we can conclude which areas we should focus more on to manage the changed requirements better.

Identifying changes is crucial for adequately managing the requirements [32]. It will only be possible to manage requirements changes if the correct change on the correct requirement is identified right away. Following the article [32], the identification of changes will lead to two fundamental requirements change management activities, change elicitation and change representation. Concluding the proper elicitation of a change in the requirements will provide more information details about that change, furthering our knowledge about what the change will do to the current system.

Once the change is identified, the next step is to analyze that change [32]. The change analysis is mainly to understand the impacts that change can have on a software system before it is implemented. Sometimes, a slight change in a requirement can disrupt the entire system on a large scale. That is due to complex connections between the requirements and consequent complexity of the system. That is why guaranteeing proper traceability to the requirements is helpful for the adequate management of a project. While tracing the requirements, it is possible to understand their impact on the project development. Change analysis can also be complemented by the predictions of changes in requirements [32]. Using previously existing content, one can predict which requirements may suffer changes and the impact that those predicted changes may have.

Cost/effort estimation is the last crucial area of requirements change management techniques. Costs/effort estimation measures the effort, time, and actual monetary costs that a change in the

requirements may have in a software project [32]. These estimations are always done at the beginning of a development process. Still, in case of a requirement modification in the middle of the development, the most probable is that those initial estimations will also change.

2.5 Requirements in Reverse Engineering

Requirements Engineering and *Reverse Engineering* are considered two different processes in a software development process by traditional practices [25].

However, in a Reverse Engineering process, there's often a necessity to recover the requirements of the legacy system, and some of those requirements may have become obsolete, may need to suffer some changes, or there's even the need to add new ones. The requirements must be recovered because the development team must understand what is already implemented and what implementation changes have to be done.

In [25], the benefits of including those recovered requirements in the Requirements Engineering phase of the SDLC are argued. Doing this will contribute to a better elicitation process and understanding of what is essential.

The need to resort to a Re-engineering process comes from the constant evolution of the software requirements. Stakeholders can change their minds, or after some time, there is a better understanding of what they want and need, or there is a change in business-related rules, environments, and needs, etc [32]. Regardless of why the requirements change, it is necessary to ensure that they are applied to the new version of the legacy system.

While a legacy system is being Re-engineered, there is a part (phases of architecture and design) of the process where the Reverse Engineering and the Requirements Engineering meet. The combination of those two processes can benefit the better understanding of the old and new requirements, what should be reused, what should suffer changes, and what became obsolete [25].

Fahmi and Choi [25] defend a new integration idea of a Re-engineering process, where the outcome of the Reverse Engineering and the phase of Requirements Engineering are combined. They state that most researches done so far about the issue of understanding a legacy system focus more on design recovery, and the authors argue that changing the focus to the requirements and combining those two processes brings many advantages when trying to tackle that problem.

Beyond that, there aren't a lot more researches about this topic. However, the theoretical advantages will most definitely improve the overall Re-engineering process.

2.6 Agile Methods

Agile is used to manage software development processes and deliver content to the customers in the fastest and most efficient way [7]. It is defined as an iterative approach since content delivery is made in small, productive increments to the final product throughout the development process. In each increment sprint, every team member is constantly involved with the requirements, plans, and results of what is being developed [7].

Nowadays, there are multiple Agile frameworks out there being used by software companies in their projects. Examples of these frameworks are Scrum, Kanban, Extreme Programming, etc. All of them are governed by the *Agile Manifesto* principles. The *Agile Manifesto* emerged to guide software developers' work [3]. Since its origin in 2001, it is still used by developers worldwide to improve their work efficiency.

Between all the existing frameworks, Scrum is the one where this dissertation will be centered. Scrum is an agile framework that helps manage a software team and the product they are building to create the most valuable product possible in the most effective way [9]. It involves *Scrum values*, a *Scrum team*, *events* and *artifacts*.

Scrum is based on the three pillars of empirical process control: transparency, inspection and adaptation [47]. The transparency is guaranteed from the frequent reviews it is subjected to. Every team member knows what is going on in the development process at the moment. The constant reviews and the retrospectives at the end of each development sprint require the inspection of what has been done and how is has been carried out. Another aspect that distinguishes Scrum from traditional development practices is its quick adaptation to change. Once a change is made in the development process, be it in terms of requirements, environments, etc, Scrum's response should be almost instantaneous so that it won't affect development costs.

Furthermore, the success of a Scrum team depends on the five Scrum Values: Commitment, Courage, Focus, Openness, and Respect [47]. All these values are crucial to maximize the teamwork's potential and, consequently, the efficiency of the development cycle.

The Scrum team must consist of a small number of people, in which one of them assumes the role of the Scrum Master, another is the Product Owner, and the others are developers [9]. The team will then work on sprints. Sprints are short periods when the team has to develop specific requirements and issues [47].

Normally, the requirements will be expressed in a *User-Story* (US) form. A *User-Story* is an explanation of a software feature/requirement written in a general and from the perspective of the end-user [15]. They are expressed in a single sentence in the following form:

"As a [persona], I [want to], [so that]." [15]

The issues that will be developed in the sprint are chosen at the beginning of that same sprint, at the sprint planning. Each day of a sprint, a daily Scrum meeting is held. The daily meeting is a short moment, normally only up to 15 minutes, where the developments are discussed. At the end of the sprint, a review and a retrospective are made, for all the team members to look at and understand what has been done and how it was implemented. A new sprint is started right after finishing the other.

To maximize transparency in the development process, Scrum uses a *Product Backlog*, which is an ordered list of the necessary requirements for the final product success [47]. The backlog may be subject to a refinement process, where the requirements become more precise and detailed, so more value to the product can be delivered. To complement the Product Backlog, the Sprint Backlog shows the goals of the development process, the requirements that are currently being worked on in the sprint, and the ones that are already completed [47]. When the sprint ends, and

more issues are worked on and finished, the increment step is performed. An increment is added to all previous increments towards the goal of the success of the final product [47].

At figure 2.4, the flow of all these different steps of a Scrum sprint is shown.

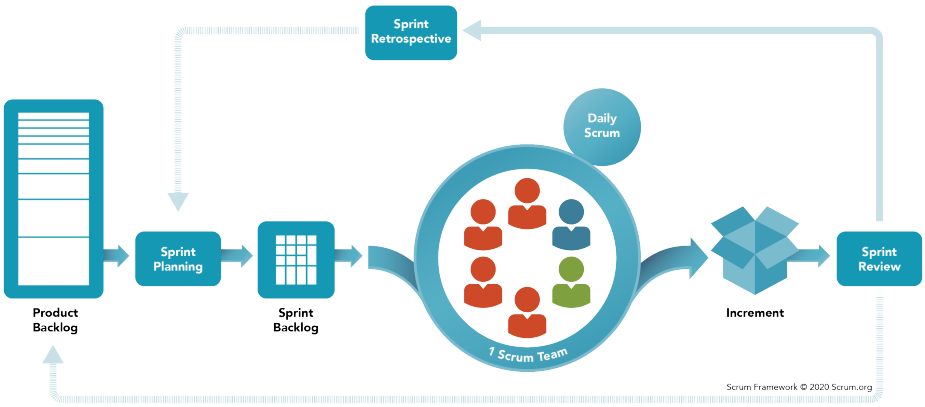

Figure 2.4: Scrum Framework from *Scrum.org* [9]

As already mentioned before, one of the elements of a Scrum team is the *Product Owner*. The *Product Owner* (PO) is the person responsible for maximizing the product value and for the management of the Product Backlog. The Product Backlog's primary goal, as already mentioned, is to help on delivering value. For this addition of matter to the product to be actually made, it is necessary to have a person responsible for its management to guarantee that the development team doesn't develop features that won't be useful for the final product [5]. This person takes on the role of the Product Owner. The PO takes on the mission of managing and ordering the Product Backlog while being responsible for the vision of the product and the value that comes from that vision [5].

All the interests from the stakeholders are represented by the PO, as that person handles all the external and internal parts of the business [5]. Those stakeholders' interests will then turn into a product vision, and the Backlog will document what is has to be done to achieve that vision [5]. Since the product vision is based on the stakeholders' interests, one could assume that they could take on the role of the Product Owner. However, according to the Scrum Book [5], it is not common for stakeholders to take on the PO's role, because most of the time their business and technical knowledge isn't enough, they are also more likely to have conflicts of interests and can lack authority in decision making. All the decisions the PO makes must be respected and heard by the team members to maintain the team's focus on value delivery [5].

When the Product Owner defines a concrete vision, the next step is to find ways to start creating value [5]. Value is basically what the stakeholders want, what they need the final product to do. When a Scrum team starts creating value, it means they are starting to implement things that the

final product needs to have to the stakeholders' satisfaction. For this value creation to happen, there is a process in Scrum called *Value Stream*.

The *Value Stream* is the set of steps used to implement continuous value increment to the customers.

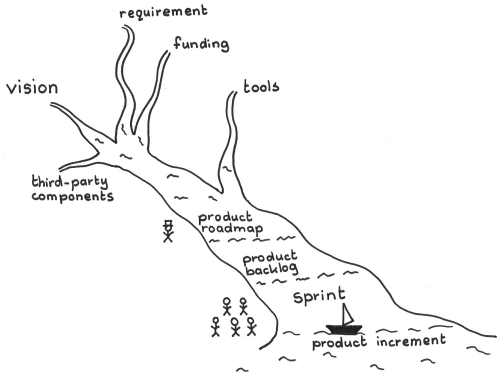

Figure 2.5: Value Stream Flow from *scrumbook.org* [5]

As seen in figure 2.5, the Value Stream sequence covers all the Scrum processes and steps that guide and help a team to develop new features to increment value. The product's vision flows to the Product Backlog, initializing a sprint, and then there is a new product value increment. The people involved in this stream are also crucial for it to work [5]. If the team is not committed to each one of the steps, a product increment won't happen.

2.7 Agile in Reverse Engineering

Scrum has already been used to implement Re-engineering processes in the past [49]. The agile framework helps to improve the Re-engineering process by taking advantage of its flexibility [49].

As stated in [50], using an Agile Re-engineering approach can bring significant benefits and improvements in cost of maintainability reduction. Additionally, the results from the study [50] show benefits in requirements implementation, cost estimation, and improved performance when using an agile framework, such as Scrum, in a Re-engineering process.

Usually, from what is described in the [50] and [49] studies, a typical agile Re-engineering process starts by estimating the cost. To do that, a technique called *Planning poker* is used to estimate effort. *Scrum poker* is when requirements previously transformed in User-Stories are estimated in terms of the effort they will take to be completed and not the time they will need to be finished. Each User-Story is assigned a *story point*, that is a number belonging to the *fibonacci sequence*.

This estimation of the requirements is essential to reduce the overall design complexity [49], which is exactly what is pretended by using agile in re-engineering. Scrum aims to reduce the system's complexity so that it becomes easier to practice Re-engineering since the Re-engineering process can be time-consuming and complex.

Additionally, nowadays, many companies demand the rapid construction of software systems. This quick implementation of new systems can bring issues such as the constant changes of the initial requirements, which can even turn some of them obsolete over time [20]. Because of that, agile practices have gained a lot of interest from software development companies [20, 23].

This issue of the ever-changing needs and quick construction or adaptation of software systems is what sometimes defines a need for a Re-engineering process. Joining a framework methodology with a development process where both have the primary goal of adapting a system as quickly as possible so that it can fulfill the new needs of its clients can only bring benefits to the overall efficiency of the development process.

2.8 Verification And Validation Practices

The agile requirements of a project need to be validated and verified. For that to happen, some practices have to be adopted.

Bahill and Henderson [53] provide definitions separately for requirements verification and validation and system verification and validation. System validation wants to assure if the system is being built on the right way [53], while verification is to ensure if the right system, with all the correct requirements, is being constructed [53]. On another note, requirements validation is required to make sure the defined requirements are complete and well specified and that it is possible to create a model and a solution where they are implemented [53]. Moreover, requirements verification serves to prove that all requirements were satisfied [53].

It is essential to know the differences between these concepts to see whether they are being applied to the development process. Studies [53] already proved that multiple software systems already failed worldwide, because of deformities on the *Requirements Engineering* process, system validation and verification, and requirements validation and verification.

In article [18], the problems that the lack of perception between the requirements engineering process and the verification and validation steps can represent to the success of the final product are discussed, as well as solutions and current practices to avoid that problem.

Making a plan for a *Software Reverse Engineering* process while coordinating the gathered requirements and their respective *verification and validation* is the main focus of this dissertation. One of its goals is to study how that coordination/alignment can improve a *Re-engineering* process and, consequently, the final product success and the overall efficiency. As stated by the authors of [18], the connection between the requirements and respective tests is used to verify what is being done, helping the development team understand what is being implemented and if that is what the Product Owner wishes.

When the requirements aren't coordinated with the rest of the development and testing steps, in addition to creating problems in development efficiency, it can also disturb the cost and schedule of a project, and the quality of the development [18]. This coordination between phases is too based on the communication between the various members of the team responsible for each phase. For example, while facing a change, a requirements engineer can modify the specifications of

the requirements and not inform the testers of the team by mistake, and vice-versa [18]. This dependency on human communication and collaboration will always be prone to error.

Challenges

Emphasizing the importance of human collaboration in this theme, the article [18] notes Organisational and Human issues has being one of the major challenges in RQE and VV alignment, for the reasons already stated above. The other main challenges of RQE and VV alignment are stated below.

Another challenge is the Definition of good quality requirements. Requirements have to be clear and complete not to create ambiguity in the development of features [18]. Developers need to know exactly what they have to implement, and, in the case of a *Re-engineering* process, they also need to understand what is already implemented. If the requirements are unclear, it is possible that what will be implemented won't be what the Product Owner wants. It won't be the developers' fault in a situation like this, but the final product will most likely fail.

The quality can also be a challenge when it comes to verification and validation. Testing a software system completely can be extremely hard. Besides that, it is necessary to have a verifying process and also verify the quality requirements [18].

While facing a Re-engineering process, the maintenance of the coordination between RQE and VV can be an issue. In a Re-engineering process, the requirements can change easily because some become obsolete and have to be replaced by new ones adapted to the new reality, or simply because the client's needs change. The tests of a software system will also suffer alterations when there is a change in the requirements, and for that to happen, the requirements have to be well traced, so the information won't be lost [18]. Once again, both these processes depend on the people responsible in the team, so they are prone to human error.

The company's size where the software development is occurring can also be a challenge. In small-scale companies, it is easier to make this alignment of the RQE and VV, while in large-scale companies, more powerful tools are necessary to guarantee it [18].

Traceability and documentation can also present challenges because sometimes it's difficult to trace requirements while being aware of all documentation [18]. It is even more difficult in a *Reverse Engineering* process where documentation is scarce.

Practices

While talking about the challenges is important, it is even more essential to talk about already existing practices for this problem.

One of the first practices stated in [18], is *Requirements Engineering*. Constant collaboration between team members and communication with the stakeholders help improve *Requirements Engineering*. In that way, the requirements won't be ambiguous and will always describe what has to be done in the most specific way.

Validation practices assure that all the features the system must possess are being implemented, or have already been implemented, in the proper way. These practices can include test case reviews, customer acceptance tests, and prototype reviews [18].

As well as validation practices, verification practices can also be a good way to prevent discoordination between RQE and VV. Verification checks if all the features that a system is supposed to have are being implemented. These practices involve starting verification in the early stages of development, so there is time for feedback and possible changes, using independent and specialized test teams and taking into account previous projects' feedback [18].

The management of possible changes is, once again, crucial. *Change Management* practices can improve the alignment of RQE and VV. Some of these practices involve the incorporation of testing roles [18].

There is a need for incentives for applying practices that guarantee the coordination between the RQE and VV. The application of such practices comes from the company and development team's incentives to use techniques and processes that guarantee requirements traceability, and documentation [18]. Implementing tracing practices between requirements and tests can also help with this alignment problem.

Practices such having a person responsible for tracking the requirements, having support tools for tests and requirements management and alignment metrics, such as test case coverage, can also help in this situation [18].

2.9 Automated Acceptance Testing

Software Testing is a process to *validate and verify* a software system. It is a way for software development companies first to evaluate a software product that is being developed [13]. It guarantees that a software product is working as it should, ensuring that the client's needs are being fulfilled.

Testing Levels	**Brief Description**
Unit Testing	Small part of the code is taken into consideration and tested
Integration Testing	The interaction between two or more components are tested
System Testing	The entire software system is tested to verify it meets the requirements
Acceptance Testing	Tests if user needs are being met

Table 2.3: *Testing Levels*

The usage of agile methods and frameworks pressures development teams to constantly test their work [13]. The need to deliver content as fast as possible gives development teams minimal margin for error. During the development, the code must be constantly tested to understand if

anything is failing. If testing isn't carried out, the probability of the final product not behaving like it should gets bigger.

Besides existing several software testing techniques already, there are also four primary software testing levels: *Unit Testing*, *Integration Testing*, *System Testing*, and *Acceptance Testing*. A brief description of those levels can be seen in Table 2.3.

To perform any type of testing level mentioned above, a set of conditions must first be defined, called *test case*.

A *test case* consists in conditions that must be checked to test a system written in a specific form. In software testing, *test cases* are used to check if the system's output is equal to the expected result.

Acceptance Testing is the testing level discussed in this dissertation. It has the ability to represent customers' needs, and interests [42]. It works by doing something to the system and checking the results while comparing the system's response to the expected one.

Performing acceptance tests manually can be too time-consuming and expensive for a software development team. To address this issue, an automation of this process is proposed to increase efficiency [28].

The concept of *Automated Acceptance Testing* (AAT) is to specify and document the requirements demanded by the clients in a way where the desired results are also included so that they can be repeatedly tested automatically [28].

Fit is a framework for developing automated acceptance tests. This framework's benefits and disadvantages are discussed in [28]. It tests business rules, and the acceptance tests are put in a table where the actual and expected results are side-by-side. *Selenium* is another framework of its kind.

With AAT, the improvement of the development efficiency and the maintenance of the writing tests can be achieved, but it comes with a cost. Because of this, it's essential to make a previous consideration of its benefits against its costs [28].

In [28], the authors admit that AAT can bring many benefits to an agile development environment. However, they also say that AAT still requires some improvements until it can reach its full potential.

Gherkin, Cucumber And BDD

Automated Acceptance Testing can be achieved with a *BDD approach* using *Gherkin* and *Cucumber*.

Gherkin is a language that describes use cases. It uses plain English, so it is effortless for anyone on the development team (developers and non-developers) to understand what is described there [8].

It can also be a good starting point for teams with junior developers with little or no experience, integrating a development team for the first time. *Gherkin* allows a greater understanding of the

projects' requirements and the development life cycle [8], which can help a lot of new software developers to adapt more quickly to a development environment.

Gherkin uses a specific syntax for the writing of their scenarios. It uses some particular keywords [2]:

- Feature
- Rule
- Example(s)/Scenario(s)
- Given, When, Then, And, But or *
- Background
- Scenario Outline/Template

In a *Gherkin Scenario*, each of its steps starts with *Given, When, Then, And, or But*. A summary of every *Gherkin keyword* and respective description is given in Table 2.4.

Gherkin Keywords	**Brief Description**
Feature	Describes a software feature in a high-level way
Rule	Available in version 6. Represents a rule that must be implemented
Example/Scenario	Illustrates a rule in the form of an example
Given	Describes initial context of the system
When	Used to describe events or actions
Then	Describes expected results
And	When successive *Given*s, *When*s and *Then*s exist
But	Same as *And*
*	Replaces *And*
Background	Allows to not repeat *Given* step in multiple scenarios
Scenario Outline	Allows to run same scenario multiple times with different values

Table 2.4: Summary of *Gherkin keywords* [2]

This easiness of involving all of the team in the understanding of the requirements and development life cycle is characteristic of a *Behavior Driven Development* (BDD) [8]. BDD is an agile methodology that promotes collaboration between a software team, while emphasizing verification and validation steps. It is seen as an extension and combination of *Test Driven Development* (TDD) and *Acceptance Testing*.

In TDD, the tests are written before the developers start to code, and then they have to write code that will pass those tests. *Acceptance Testing* is about tests made by the customers to understand whether the needs of the software system are being guaranteed.

One of the goals of BDD is to make sure that all requirements are well understood by the team [8]. If the requirements are well understood by everyone right from the beginning, the likelihood of finding possible errors early in the development process will cause errors to also be resolved at an early stage of development [8]. This can help a company reduce costs, since finding a problem in the requirements only in the middle of the development process is much more expensive.

Cucumber is defined in [8] as "an open-source software testing tool". It supports BDD, since it uses Gherkin to create tests. Cucumber reads Gherkin's syntax to develop tests and validate the code implemented by the developers. Cucumber executes each step in the sequence of *Gherkin Scenario* steps [2]. Since Gherkin describes the system's requirements, with Cucumber using Gherkin to write acceptance tests, it will always be guaranteed that the tests will validate if the code meets all the specified requirements or not.

Using Gherkin and Cucumber in a development software process will benefit the alignment of RQE with VV steps, which will also help its efficiency and increase the probability of making the PO and Stakeholders satisfied with the final product. This situation of improving the alignment of RQE with VV also gains more interest when considering the environment where this dissertation will occur. In a team with less experienced developers, beginners need to understand the importance of the requirements and their validation.

Chapter 3

Problem

Contents

In this chapter, the problem of this dissertation, already briefly introduced in Section 1.3, will be further deepened and detailed.

At the start of the chapter (in Section 3.1), the issues that this dissertation will try to mitigate are described, followed by a more detailed contextualization (in Section 3.2) of Vodafone's reality, including some of the made assumptions (in Section 3.3). The overall Methodology is defined in Section 3.4 and a brief summary of the whole chapter is given in Section 3.5.

3.1 Problem

Vodafone develops their projects in an agile way. Having to face a *Reverse Engineering process*, from the requirements management point of view, there is a need to arrange the best approaches to specify, manage and validate the agile requirements of the project to ensure the process's efficiency.

The usage of an Agile framework such as Scrum helps increase the efficiency of the *Software Development Life Cycle*. It does it in a way where the management of the development team and the overall development process contributes to a quick development pace, always thinking about the satisfaction of the Product Owner and Stakeholders.

However, even if Scrum contributes to the constant integration of the entire development team in all the development steps, a direct communication between the different phases of SDLC doesn't exist. This situation may present an obstacle to the final product's success in case a requirement change in the middle of the development process occurs.

A software system that needs to be Re-engineered is a system that doesn't satisfy the needs of its users anymore. Not meeting the needs means that the requirements have changed, or new ones have to be added, or old ones have to be removed. Whatever the case, the requirements must be adapted to the new reality.

These changes can happen for multiple reasons. The stakeholders change their minds, the Product Owner gets a better understanding of the product necessities, the external or business environment changes, etc. For an Agile team, changes like these shouldn't be a problem. However, the situation takes a turn when dealing with a Re-engineering process.

At Vodafone, there is a need to structure Re-engineering processes in some teams. Not having a well-defined plan for it makes the first Re-engineering phase, the Reverse Engineering process, much more time and cost-consuming since the best approaches won't most likely be applied.

Reverse Engineering is a highly complex process. The complexity of this process can affect its efficiency, and that is an issue that needs to be addressed and mitigated. It gets even more complicated when there aren't many project artifacts from which to study and analyze what already exists in the legacy system.

When there are no previously documented requirements for the legacy system, the existent artifacts have to be analyzed, studied, and understood so that the development team knows what's already been done. At the same time, a kind of Requirements Engineering process is started. New requirements are proposed, and they have to be added to the ones recovered from the Reverse Engineering process, or it can even happen that the new thought requirements are a changed version of the ones recaptured from the legacy system.

All these changes in the system's requirements have to be well managed and validated. To manage all these requirements, they must first be well specified. The requirements have to be specified in a way that everyone on the team understands the PO's wishes. Not specifying the requirements or making them ambiguous creates problems within the development since the developers and the testers won't understand what is pretended. The risk of them implementing the wrong features is high, which is a problem that all development teams want to avoid.

Requirements validation serves to check if everything is being met and implemented. In a Reverse Engineering process, even if the requirements are specified after studying all system artifacts, there is a need to guarantee that they are actually implemented. This happens because we have no guarantee that the legacy system artifacts are updated to the project's current situation, and this uncertainty is a problem that can affect, by a lot, the process's efficiency. Not only that, but there is also a need to check if the new requirements that the new version of the legacy system has to meet won't compromise old but still needed requirements.

To check and validate the requirements, software testing must be performed. Vodafone's teams require software testing methods to validate their systems, and they are interested in investing in

automated approaches. Automated software testing methods greatly benefit the lack of communication between different development phases, improving the process's efficiency.

All system requirements have to be met and validated to guarantee the final product's success. Failure to comply with the requirements changes will cause the dissatisfaction of the Product Owner with the final product.

The main goal of any software development process is to guarantee the Product Owner and Stakeholders' satisfaction with the final product. If they aren't satisfied, the product will be considered a failure, even if the development was efficient, delivered on time, and without any additional associated costs.

3.2 Problem Contextualization

A brief contextualization for this dissertation was already given in Section 1.1. However, to better understand from where this problem appeared, it is better to specify in more detail why some of Vodafone's software development teams have these problems.

The Vodafone software development team that I integrated and where this dissertation was based on is a team formed by senior and junior developers. Junior developers don't have much experience yet, so they don't fully understand the importance of a project's requirements, nor why testing is crucial.

The need to ambiance them in the best possible way gains new relevance. It's up to the team to find an easy way for the junior developers to understand the importance of a system's requirements, to improve the development process efficiency. The understanding of those requirements will also pass by implementing tests that will validate them or not.

Besides that, requirements are constantly changing at a big company like Vodafone, since their needs depend on their market needs. This makes software systems that are currently being developed more likely to suffer changes in their initial requirements. If a good management of these changes isn't provided, the probability that the final product won't meet the new market needs is very high.

The management of the changes in requirements goes further beyond the traditional development cycle. A system that, in other times, has already achieved its purpose may have to be adapted to new business realities. This adaptation is based on a Re-engineering process. When Reverse Engineering a project, the old and new requirements must be adequately specified and managed so that a team understands what they have to do and what's already been done. They also have to be validated by software testing.

To achieve proper specification, management and validation of the requirements in Reverse Engineering, it is necessary to implement an efficient plan covering all these essential aspects.

3.3 Assumptions

The work developed is considered beneficial, especially to Vodafone's reality, since it explores new ways of tackling Reverse Engineering processes while also exploring the concept of AAT.

To try and mitigate the current problems stated in Section 3.1, two assumptions about requirements and validation in Reverse Engineering are considered:

1. **Product Owners must always be aware of everything that goes on in the requirements and testing phases.**

   As stated several times throughout this document, the final product's success depends on the PO's satisfaction. This satisfaction will only happen if his needs are guaranteed.

   This guarantee will be provided by the correct specification and management of the requirements and by the proper validation.

   The requirements of a product should always be specified clearly, so there are no ambiguities when passing their knowledge to the developers and testers. If they do not ensure this clarity, the most probable thing to happen is the development not being adequate for what is wanted. Moreover, the testers also need this clarity from the specified requirements to validate the essential features that the PO wants to see in the final product.

   The PO is the person that states what is needed for the final product and what the requirements are, so it's normal for him to be aware of every step that happens in the requirements phase. However, the testing phase is just as vital for him. Even if he doesn't play any role in the development of the tests, everything that he idealizes will be assured (or not) in that phase. Because of that, he needs to be sure that the developed tests involve the validation of the right parts of the code and the right features.

2. **Automated Acceptance Tests improve development efficiency.**

   Aligning the requirements and testing phases would make the PO awareness of the two above-mentioned stages a possible and more efficient reality. Time-consuming-wise, the PO wouldn't have to spend much attention on the testing phase if it was automatically connected to the requirements specified by him. He would only have to write the requirements and respective test cases, and then the tests would be almost automatic.

   With AAT, this reality would be possible. The communication between the two phases would be facilitated since the tests would be aware of what is going on with the requirements and vice-versa.

   If the requirements and testing phases are always aware of each other, the probability of the development efficiency increasing grows. There is no waste of time checking manually one by one if the requirements are being met or not. In case of changes in the requirements, the effects of these changes are also seen in the work environment, so if they cause issues, they can be fixed immediately.

3.4 Methodology

This dissertation aims to provide a plan for a Reverse Engineering process for Vodafone to use in their teams. The scheme aims to improve the process's overall efficiency while focusing on aspects they want to improve in their teams, such as requirements management and validation methods. To make this a feasible solution, the work was divided in three parts:

(a) **State of the Art**

The first stage conducted for the realization of this dissertation was the research of the background and related work. To make a *Reverse Engineering* plan, it was crucial first to understand its and the *Re-engineering* concept (Section 2.1), not forgetting the *Agile* environment where the work was being done (Section 2.6). The efficiency of the plan and the points to be improved are based on the *requirements* specification and management (Section 2.5) and *validation* (Section 2.8), so those concepts had also to be scrutinized.

Analyzing how a Software Reverse Engineering process must be directed was essential to start my work. Trying to adapt that process to Vodafone's reality wasn't easy, and it involved understanding the best way to improve a development process while considering Vodafone's necessities and areas where they feel improvement is needed.

The studies about AAT in Agile environments were also essential to this dissertation. Understanding how it works and the benefits that it brings was one of the bases of the plan developed. However, the studies about it are still very few, so there weren't a lot of resources from which to make conclusions.

(b) **Reverse Engineering Plan**

With the developed work a Reverse Engineering efficient plan was created. The creation of this plan involved several steps and will be all specified in detail further on in Section 4.

The plan improves the efficiency of the development process while investing in automated testing methods that facilitate the product's validation.

(c) **Qualitative Validation**

The validation of the work developed in this dissertation comes from a qualitative type of validation.

Five experts in Product Owner roles or similar from Vodafone participated in interviews where the implemented work was validated. For the most reliable validation possible, a demo was shown to the interviewees beforehand. Once they understood all the development steps, a total of 10 questions, not counting the questions of profile characterization, were asked.

The questions asked were intended to understand the interviewees' opinions about using Gherkin and Cucumber in a development process (especially in the Reverse Engineering

one) and the benefits that tools like those would have if they were implemented in their respective teams. The detailed questions and their respective results are revealed in Section 5.

3.5 Summary

The problem of this dissertation and its context were completely specified and detailed in this chapter, in Section 3.1 and Section 3.2.

After analyzing the research made and thinking about Vodafone's current reality, all the current issues were stated, and some assumptions about the requirements and validation phases were also considered (Section 3.3).

In the end, a brief description of the methodology was given in Section 3.4.

Chapter 4

Proposed Solution

Contents

A proposed solution for the issues stated in Chapter 3 is discussed in this chapter.

To better understand what was proposed, the first step is to understand the project on which the work was based. In Section 4.1, a brief context to it is given, while in Section 4.2 the proposed solution is explained step by step. An explanation of how the implemented plan was developed thinking in the team is given in Section 4.3, and a summary of it all can be found in Section 4.4.

4.1 Project Characteristics

In my Vodafone's development team, a project that had already been implemented was proposed to go through a Reverse Engineering process. For confidentiality reasons, the real name of the said project cannot be disclosed, so for the proposed of this document we will call it *Proj A*.

Proj A is a platform for some of Vodafone's services. Once again, because of confidentiality reasons, the detailed description of this project can't be revealed. However, some other details, for contextualization, can.

It is an extremely extensive java-based project. It has about 40 classes, and each one varies from 300+ to 1000+ lines of code.

Big projects like this are typical in a company as large as Vodafone. Since it is a system that possesses crucial features for the good functionality of Vodafone's services, it needs to be updated from time to time to guarantee the company's needs.

Going through a Reverse Engineering process is already too time and cost-consuming. It takes a lot of time to study, analyze and understand a legacy system, even when most project artifacts exist.

In the case of *Proj A*, documentation exists. However, due to the system's long history, it is spread across various formats, templates, levels of detail, and even languages. Since it's difficult to consolidate all this information, it was decided to experiment with a new RE plan where the only project artifacts provided were the source code and documentation of over 70 pages. While this increases the complexity of the already complex RE process, starting almost from scratch allowed the realization of an even more complete plan that can help even the most complex RE processes.

Also, for the purpose of the experiment done with this plan, the test methods used to validate *Proj A* weren't provided. The goal was to introduce new automated tests to the system, so considering the old ones wouldn't make sense.

This became an even bigger issue to the plan's complexity since *Proj A* is a complete back-end system. A back-end system, in short, deals with server requests and the system's database, which means that there's no way to visualize what the code does like it would happen if it had an interface. With no way to visualize what the system does and running the code not being an option, understanding the expected results and how the system should behave can only be supported by studying and analyzing the code and documentation.

There was a need to separate the source code from what I would be developing, so it was decided together with a senior developer of the team that it would be best not to implement the *Gherkin* scenarios and tests at the same place as the source code. This guaranteed the differentiation of the RE plan and implementation.

Any software development team typically uses standard tools such as *Jira* to manage the Product Backlog. To diversify this plan from those commonly used tools, another tool was used to manage the requirements and its Product Backlog.

Given the extensive nature of this project, only the analysis and studying of half of it was considered for the development of the RE plan.

4.2 Reverse Engineering Plan

The production of the Reverse Engineering Plan was divided in four different steps:

1. **Analysis and Understanding of the Legacy System**

   To understand the core of the LS, it was necessary to study and analyze the source code and the existing documentation in detail.

The procedure is further explained in Sub-Section 4.2.1.

2. **Managing Requirements with Gitlab**

   This step of the process is something that will take place throughout the entire plan. Each time a new requirement is gathered or thought it will be put on the Product Backlog, and each time a *commit* is made the management has to be guaranteed.

   More about this step is described in Sub-Section 4.2.2.

3. **Passing Requirements to Gherkin**

   In Sub-Section 4.2.3, the procedure of passing/specifying the requirements to Gherkin is addressed. Some examples of the implementation are also shown to give more context.

4. **Tests with Cucumber**

   Consequently to the requirements passed to gherkin, the Cucumber tests are implemented based on the Gherkin syntax. In Sub-Section 4.2.4, examples of the tests implemented can be visualized, together with an explanation of how they were developed.

As all these steps will be more detailed in the following Sub-Sections, it is vital first to understand the connection between them. In Figure 4.1 the connection and flow between all the steps can be visualized.

As shown in Figure 4.1, a Reverse Engineering process must start when facing a LS. The first step of that process is to understand what already exists to recover the system requirements.

While recovering the requirements, some new ones may appear, according to the PO's needs. If this happens, new requirements must be gathered in a similar process as Requirements Engineering. When gathering new requirements, there's a possibility that the new additions may be requirements already defined in the legacy system but with minor changes. In this case, we have to go back to recovering the requirements and proceed with the changes (Sub-Section 4.2.1).

The new and old requirements are in constant interchange. They must be aware of one another so that situations where a requirement compromises the other happen.

When a requirement is recovered or gathered (depending on its origin), it is then placed on the Product Backlog as an Issue, expressed as an User-Story. The new issue becomes part of the list of Issues related to that process, and in the Issue Board, it can be seen in the *Open* section of the board. The management related to the Product Backlog will be specified in detail in Sub-Section 4.2.2.

In every Issue, a test case or a document related to its development can be attached or written as a comment. This additional information helps with the writing of *Gherkin Scenarios*, as described in Sub-Section 4.2.3.

When a *Gherkin Scenario* starts to be implemented, the Issue related to it moves from section *Open* to *In Progress (Gherkin)* in the *Issue Board*. When the *Cucumber* tests start being implemented (Sub-Section 4.2.4) the same alterations happen in the *Issue Board*, but instead of being

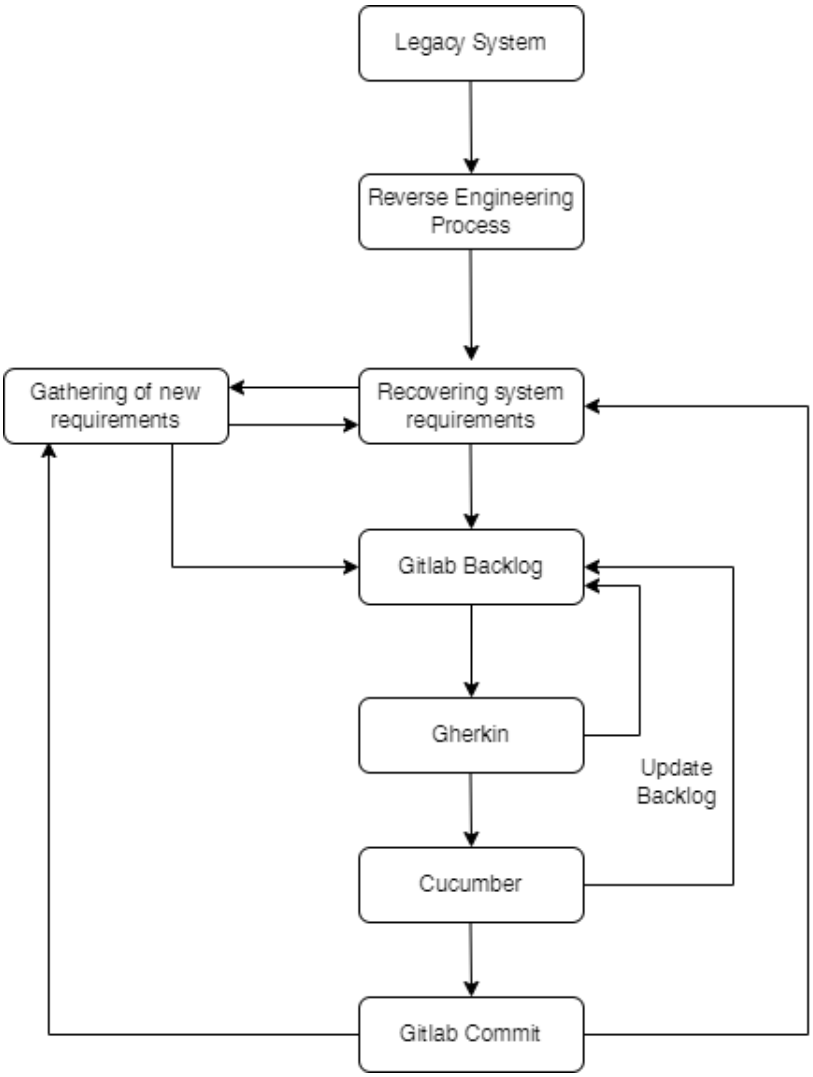

Figure 4.1: Solution's Flowchart

from section *Open* to *In Progress (Gherkin)* is from *In Progress (Gherkin)* to *Testing (Cucumber)*. The Product Backlog is being constantly updated.

When a Gherkin Scenario and its respective Cucumber test are completed, a *commit* to Gitlab is made with a specific *commit* message detailed in Sub-Section 4.2.2.

Once a test is on Gitlab, one can try to recover more requirements or gather more new ones.

4.2.1 Analysis and Understanding of the Legacy System

The LS we were dealing with in this dissertation is *Proj A*, already characterized and contextualized in Section 4.1 above. As already mentioned, it is a back-end java project with only the source code and 70-plus pages of documentation associated with it.

For the purpose of this work and main aspects that Vodafone is trying to improve in their software development team, the existence of specified project requirements was beyond necessary.

Since documented requirements weren't provided, the first thing to do was to understand what was implemented and which features *Proj A* already possessed. This process was easier said than done, as studying and analyzing a project of that extension is very time-consuming.

One of the main goals of this dissertation is to improve the efficiency of the RE process. Taking too much time to analyze what already exists wouldn't contribute to the desired efficiency improvement.

Tactics to speed up and help with this process had to be defined, especially since, at the time the study of the code was being conducted, the documentation related to it hadn't been provided yet.

The source code included *comments* explaining how some functions and some of its parts worked, which in theory could be something beneficial. However, most of those *comments* were already outdated, which did not favor the situation.

Once it became clear that another path had to be followed, together with the team's supervisors, it was decided that the best way to proceed was first to study the connections between the classes and functions. For that, *sequence diagrams* were constructed.

Sequence Diagrams are diagrams where it's possible to visualize the connections between all the different objects of a system's code, focusing on the time each object instance lasts.

A total of 14 sequence diagrams associated with the primary classes on which this work was based were implemented. Once again, because of confidentiality reasons, the diagrams cannot be disclosed. However, a part of one of the diagrams is shown in Figure 4.2, duly censored to accommodate the confidentiality issues.

The sequence diagrams helped to understand the flow of the code. But, even if it helped to comprehend a part of the code, it wasn't enough.

When finalizing all the sequence diagrams, a complementary documentation was delivered. The document consisted of 70-plus pages with information on the application's main features, including some of the responses to server requests.

The document was directed at developers. By this, I mean that the information it contained was a more technical type of information, and a person who didn't have a lot or even any kind of experience with coding wouldn't understand what was there redacted.

Even from a developer's perspective, the document was not the most straightforward reading for those who had not yet had contact with the code development. The information contained in the paper was crucial to specify the requirements in Gherkin. Because of its importance and difficulty understanding, a senior developer of the code was needed to help me clarify my doubts throughout this process.

The help consisted of clarifying some doubts about the expected response from the server and providing data beyond my reach and knowledge. He also offered some files needed for the development of the tests.

The first requirements started to be understood with the study and analysis of all these things: code, sequence diagrams, documentation, and the senior developer's help with my doubts. Once this happened, they were passed to Gitlab's *Issue Board* in the form of *User-Stories*.

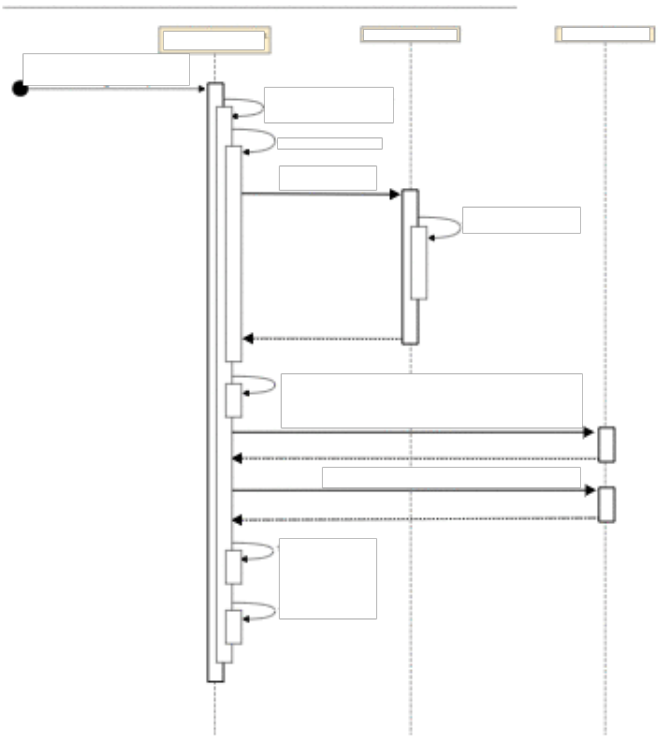

Figure 4.2: Part of a implemented *Sequence Diagram*

At the same time, new requirements needed for the new version of the LS were also gathered. In the case of this project, most of those new requirements were already existing LS requirements that needed to suffer changes. However, more new requirements will appear even after the delivery of this document.

4.2.2 Management with *Gitlab*

After the first requirements appeared, the first step was to document them in *Gitlab*.

On the team I was part of, *Jira* is used to manage the Product Backlog. *Jira* is a platform that helps development teams to manage their work [16]. It can manage and track all kinds of requirements and test cases [16].

Jira is Vodafone's election management tool for a project's requirements. It's a tool that Vodafone has already adopted for many years, and only if they see benefits in other tools will they consider changing.

In a company where they are used to a particular tool, it is risky to try inserting a "new" one into the development process. *Gitlab* is not exactly a new tool to Vodafone's teams. It is used as a code repository for most software projects. However, *Gitlab*'s ability to manage requirements/issues is never utilized.

This dissertation's work proposes the use of *Gitlab* to manage the issues because of one functionality. There's the possibility of connecting each issue to the respective *commit* where that issue

is being implemented, both with *Jira* and *Gitlab*. However, in *Gitlab*'s case is easier to do that compared with *Jira*. Furthermore, it is better to have all of that in a single platform.

Because of those reasons, it was decided to work with *Gitlab* instead of *Jira* to give Vodafone new perspectives. If they choose to apply these tools and this plan to their future projects, they can always opt to use *Jira* instead of *Gitlab* since it's possible to work with both of them in really similar ways.

Regarding the documentation of the requirements in *Gitlab*, it was necessary to pass them to a *User-Story* form. *User-stories* promote more focus on tasks and team collaboration [14]. For this plan, *User-stories* were decided as the best way for the *issues* descriptions since they are easy to read and anyone from the team, experienced code people or not, can understand them.

When a new *issue* is created, assigning it to the person responsible for it is necessary. Most of the time, some other tasks like estimating the effort of the US are required to be done right after the *issue* creation. However, some particular restrictions were faced while working with *Gitlab*.

Those limitations were related to the *Gitlab*'s version used by Vodafone. Vodafone uses the most basic *Gitlab*'s version available. Until now, they didn't feel the need to upgrade it, but if they find benefits in it, they are willing to invest in the premium version.

Gitlab's premium version is similar to *Jira*'s. In the premium version, *issues* can be specified as being *user-stories*, *epics*, *tasks*, etc; if an *issue* is dependent of another *issue*, we can point out that first *issue* as being *blocked* until the second one is implemented; the effort of an *issue* can be estimated; among other features [6]. While *Gitlab* allows to relate *issues* to one another, that *blocked* feature doesn't exist is the basic version.

The impossibility of using features like these slightly limited the work development. The management of the *Backlog* couldn't be performed in the best way possible. Even though it still did the work, it could be significantly improved with the premium version.

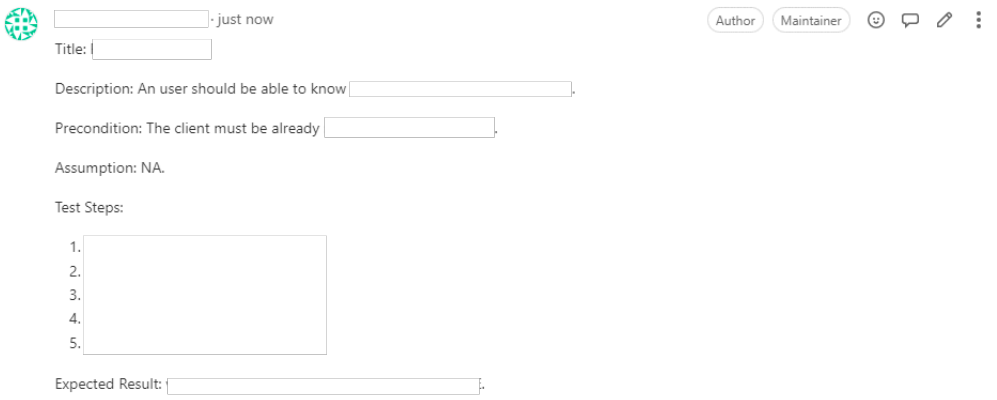

Figure 4.3: Example of *test case* in an *issue* page

After assigning the *issue* to the responsible person, it's time to specify the *test case*. The *test case* is defined inside of the *issue* page in the form of a comment with a specific structure. An example is shown in Figure 4.3, while again omitting some parts because of confidentiality issues.

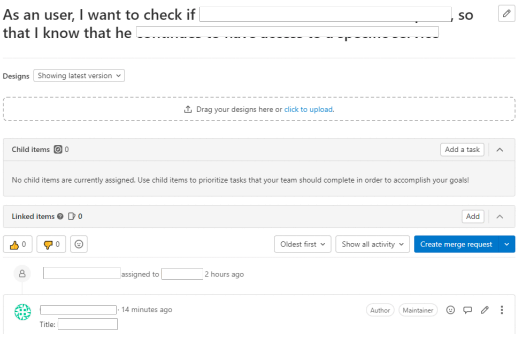

Figure 4.4: Place to attach files in *Issue* page

With the *test case* duly commented, it's time to attach some important files needed for the development of the *issue* in the space reserved for such (demonstrated in Figure 4.4) in its page.

In this case, the essential files that could have been attached weren't because *Gitlab* didn't support their formats, so it wasn't a handy feature for this project.

The initial part is finished, so it's time to start the development. When the development of an issue is started, on the *Issue Board*, it is passed to the *In Progress (Gherkin)* block. Once the test for it starts, it is then moved to the section called *Testing (Cucumber)*.

All these steps must be done manually by the person responsible for the issue. However, in the last step of moving the issue from *Testing (Cucumber)* to *Close*, the manual way is no longer applied. When an issue is passed to *Close*, the final *commit*, with the gherkin scenario and test, is made. The *commit message* is written in the form showed in Figure 4.5. The *issue* is mentioned in the *commit message*, and by doing this, it will be automatically passed to the *Close* section of the board. The rest of the sentence is written to ensure uniformity.

Figure 4.5: *Git commit messages* structure

If a mistake happens at the moment of the *commit*, like writing the *commit message* differently than the way pretended, there's the possibility of reverting that mistake. It can be done by typing the following commands in a terminal in the project's directory:

1. ***git rebase –edit-todo***

This command opens the *commits* history in the default editor. Once the editor is open, a lists of the *commits* will appear and each one of them will have the word *pick* before the *commit message*. That word needs to be changed to *reword*. To do this, first press ***SHIFT + R***, change *pick* to *reword*, then press ***ESC*** and save it by pressing ***:WQ***. Type the new *commit message* in the new editor that opens and save it.

2. ***git push –force***

This practice is encouraged since it is essential for a *commit* always to have an associated issue, as it helps with the traceability.

After a while of doing all these steps for the multiple requirements that keep emerging, this project's *Issue Board* will look like what is in Figure 4.6. Once again, omitted information is essential to keep confidentiality.

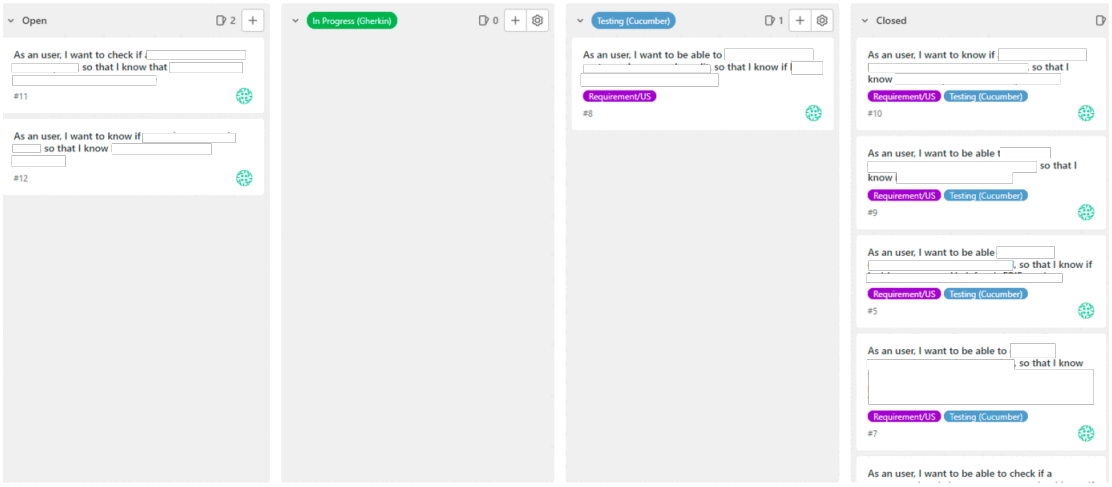

Figure 4.6: *Issue Board* at one point of the development

4.2.3 Specifying Requirements with Gherkin

To specify a requirement with Gherkin, one has to consider the *test case* defined in the *issue* comments.

Gherkin uses the sequence of the *Test Steps* specified in the *test case* to write a scenario. The first *Test Step* equals to the *Given* statement, the second to the *When* statement, and so on. From the description of the *Step*, it is pretty perceptible to each statement it refers to.

All the *Gherkin* scenarios are implemented in the folder *features*, as shown in Figure 4.7.

For each *Gherkin feature*, multiple *Gherkin Scenarios* can be attached to it. In addition, in some of the cases, instead of a *Gherkin Scenario* having the typical *Given*, *When*, *Then* structure, it can look like what is shown in Figure 4.8.

In this specific case, writing scenarios like that is necessary since a request can depend on another request, and there's no other way to implement it.

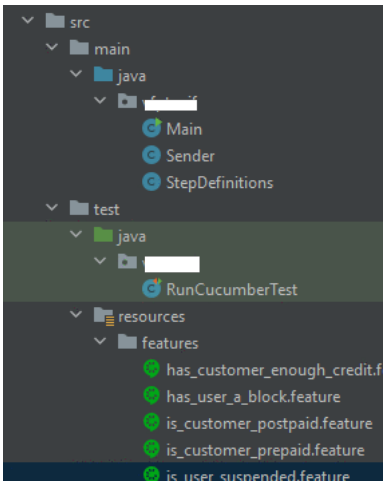

Figure 4.7: Project's code structure

The help of the senior developer was the most crucial in this and the *testing* phases. Here, the *Given* statement started to analyze a particular unique field. This field identifies each Vodafone client who belongs to that system, so I didn't have access to it right away for safety reasons. I had to ask the senior developer if he could provide me with that parameter, and only then would I have access to it.

For the following statements, *When* and *Then*, all the information that I needed was either at the project's documentation or, once again, in the *test case*. However, the senior developer's help was as necessary since some doubts arose, especially in cases like Figure 4.8 where the typical scenario structure wasn't used.

The *Expected Results* detailed in the *test case Test Steps* were used in the *Then* statements. It is in that part of the code that the actual response will be compared with the expected results, but that will be further explores in next sub-section.

```
Feature: Has a customer [redacted]?
  A customer with a [redacted] has to have [redacted]

  Scenario: Customer has [redacted]
    Given charging id type is "[redacted]" and charging id value is "[redacted]"
    When I make a "[redacted]" request
    And I make a "[redacted]" request
    Then The "[redacted]" field of the response is the code for the [redacted] step
    When I make a "[redacted]" request
    Then The "[redacted]" field in the response should be "[redacted]"
    And The "[redacted]" field in the response should be "[redacted]"
```

Figure 4.8: Example of implemented *Gherkin Feature and Scenario*

4.2.4 Requirements Validation with Cucumber

The validation of the requirements is based on the *Gherkin* specification of them.

Cucumber takes on the *Gherkin* syntax to develop the acceptance tests. While this is the basis, much more has to happen to make these types of tests in this project.

The back-end origin of *Proj A* makes its tests a constant *request-response* process. To make a *request* to the server and then receive a *response* to it, it's necessary to implement those aspects on the *Gherkin and Cucumber* source code.

This was realized in the *Sender* class. The implemented code for it can't be shown, but a type of preview of what it looks like is shown in Figure 4.9. The part of the *String request* was only possible to implement with the help of some complementary files for the needed *xmls*, provided by the senior developer. These types of files were the ones that weren't allowed to be attached to the *Gitlab's issues* because of their format (*.sh*).

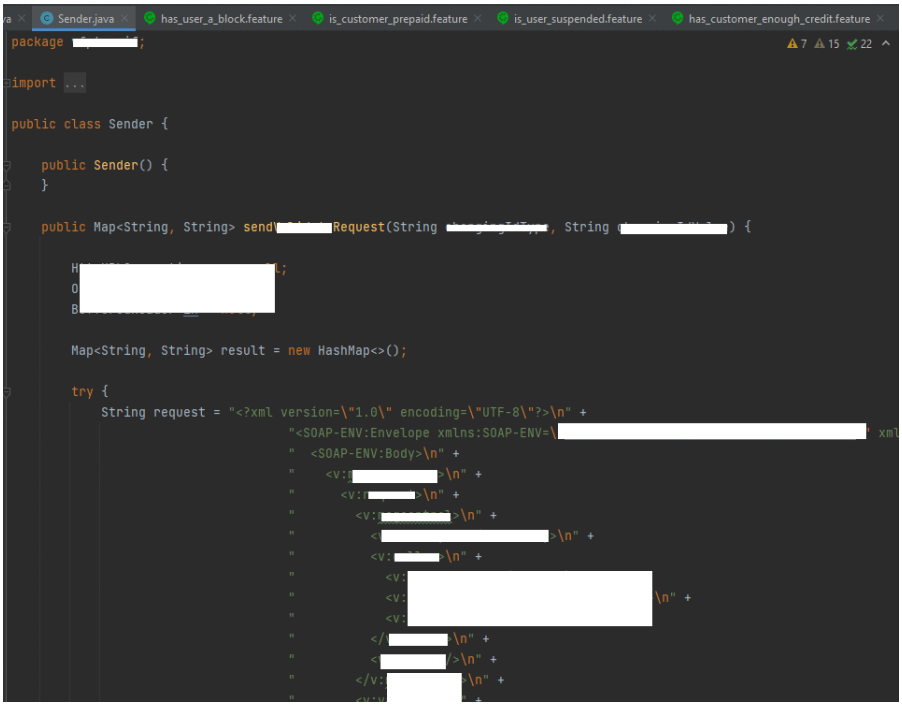

Figure 4.9: *Sender* class

Those *Sender* functions are then utilized in the actual *Cucumber* tests. The tests were developed in the *StepDefinitions* class, as shown in Figure 4.10.

This system's *request-response* behavior makes it possible to reduce the number of tests written. By this, I mean that, for example, for each different *Given* statement of the multiple scenarios, a unique *Given Statement Test* was implemented. One single test was able to test all the *Given* statements of all the scenarios. The same happened to most of the *Then* statements. In the *Given* statements case, the test that dealt with it was a function with a *switch*, which ensured that no more tests would need to be implemented for that statement.

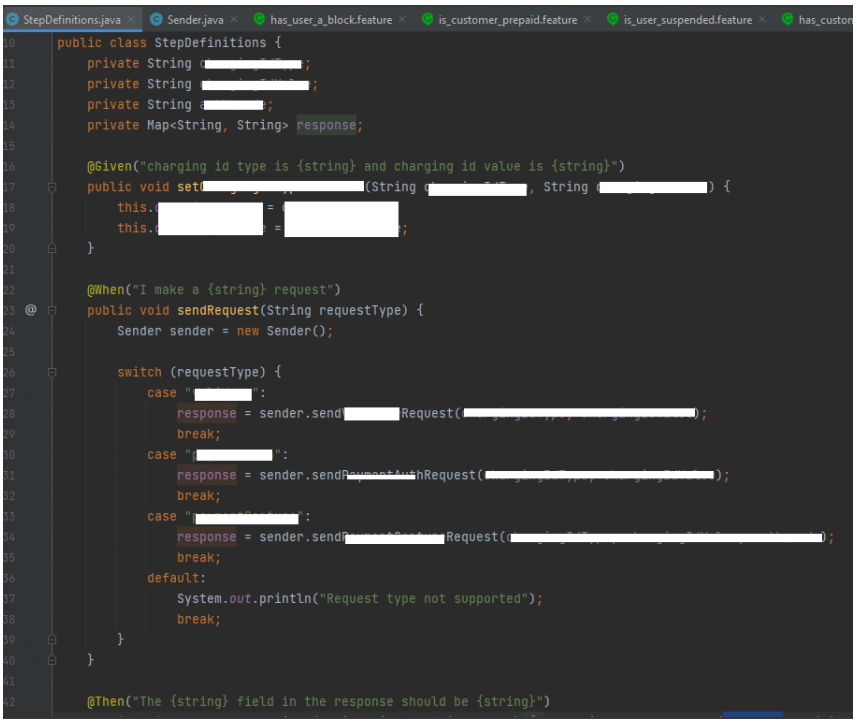

Figure 4.10: *Sender* class

Using *Gherkin* and *Cucumber* to validate this system allowed us to realize the idea of an AAT method. The fact that the same test function provides for the validation of multiple scenarios benefits a software system where new requirements are constantly being added and changed, since it only takes a moment to specify them and the testing is, most of the time, automatic.

Another critical point to add is the help *Cucumber* provided in understanding the *legacy system*. Even though understanding it started right at the beginning of this plan, as I was implementing the *Cucumber* tests, my knowledge of the system increased exponentially. This happened because, while running the *Gherkin and Cucumber* project, it is not only possible to see if the tests are passing or not, but it is also possible to see the response of the code to the *request* specified in the *When* statement of the *Gherkin scenario*. This helped me to see what was happening with the code, which improved my knowledge of the system. An example of those responses with its multiple fields is shown in Figure 4.11.

Figure 4.11: Example of a code *response* when running *Cucumber*

At the end of the analysis of the proposed part of the LS, the validation result is shown in Figure 4.12.

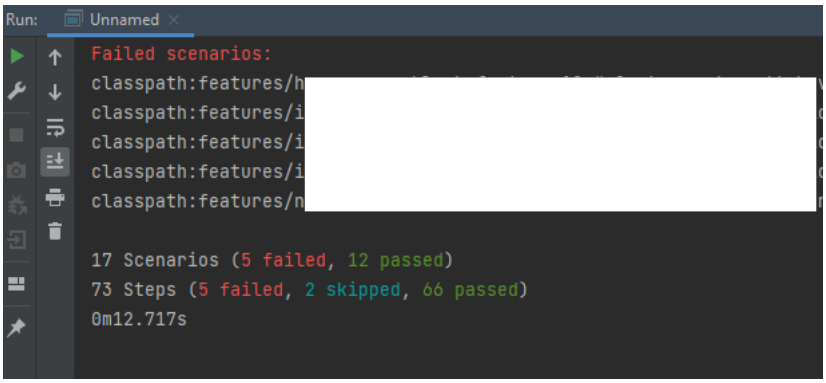

Figure 4.12: *Cucumber* validation results

4.3 Plan in the Team

While the plan for this RE process was all developed by me, it was carried out having in mind a group of people, inside of a *Scrum Team*, that would be in charge of this.

It would be unrealistic to think that only one person could do all this work when its primary goal is to improve efficiency. Only one person would take too much time working on this, and wasting time is something that can't happen at Vodafone.

For this dissertation, logically, all the work had to be developed by me. However, if Vodafone were to implement this plan in other of their projects, a small group of people inside the team would have to be allocated to it.

Ideally, that group of people would consist of a PO, the PO's go-to person in technical aspects (if the PO wasn't technical oriented), and two or three *Quality Assurance* people.

Tools like *Gitlab* for the *issues* management would be beneficial when dealing with a group of people since, in that way, every person would be able to see what the other was doing, improving the PO's knowledge of what is happening in the requirements and tests phases.

4.4 Summary

The proposed solution used tools like *Gherkin*, *Gitlab* and *Cucumber* to define a *Reverse Engineering plan*.

Being defined while thinking in a group of people participating in it, the solution aims to improve the efficiency of the development while promoting an environment where the requirements and tests are conscious of each other.

It provides an *Automated Acceptance Testing* environment, improving the speed and ease with which the requirements are tested.

Chapter 5

Validation

Contents

In this chapter, the results of this dissertation validation are presented. A set of interviews with experts validated the developed plan.

5.1 Methodology

To evaluate the usefulness and helpfulness of the *Reverse Engineering* plan to Vodafone's reality, some interviews took place with experts in the field (e.g., POs, etc).

To conduct these interviews, a list of 13 relevant questions was previously prepared, as well as a *demo* to show the experts the context and the implementation of my work. The *demo* showed the flow of the steps argued at the *Solution* chapter (4).

Five interviews were conducted, and the duration of each interview was approximately one hour. In each interview, the participants were always asked the same set of questions. The interviewees' personal opinion was required for each question, so the answers weren't restricted to only what the questions asked, and freedom to discuss other relevant topics existed.

5.2 Question's List

The question's list is divided in four main groups:

1. Interviewee's profile characterization questions.
2. General questions about requirements, software testing and Gitlab.
3. Question about the Interviewee's current and past teams.
4. Project oriented questions.

All questions are fully detailed in Appendix A.

5.3 Interviewees Characterization

All the interviews were conducted inside the company. However, all five interviewees belong to different Vodafone software development teams, so all their experiences differ quite a bit. Their answers can be seen in Table A.1.

Some interviewees have experience in more than one role, giving them a broader perspective of the issues that a development process can have. Not only that, but the different Product Owners (or similar roles) are divided into more technical and more business ones.

Furthermore, all interviewees' teams follow Scrum, allowing more reliable feedback on the proposed solution.

Since Jira is the go-to Product Backlog management tool, all teams use it for requirements management. Regarding the testing methods used, all teams use different ones. The only similarity is that the techniques are some kind of manual acceptance tests practiced by not technical people.

5.4 Perception of requirements gathering, software testing and Backlog's management importance

An analysis of the answers in Table A.2 was made to understand the interviewees' overall opinions.

5.4.1 Requirements gathering importance

Of the five interviewed persons, four classified *requirements gathering* as a **fundamental process of the development cycle**.

Participant *B* somewhat discards the importance of requirements. Instead, he states that defining a goal is the most critical part of a development process. As a goal is defined, requirements are specified, and the development can proceed. To him, describing the purpose of the final product is key to success.

The other participants support the idea of the importance of the requirements, but their specification right at the beginning of the SDLC is not always something possible to happen. While they believed that **specifying the requirements at the earliest phase possible is the best way to assure product quality**, this is not always possible because of the necessity to deliver the product as fast as possible. In addition, **new requirements or changes can arise during the development process**.

5.4.2 Software Testing importance

All participants also consider *Software Testing* to be a **crucial step** in SDLC.

The opinions about whether the testing process should start early on the development cycle or only after code implementation differ slightly. Some participants state that it should begin as early as possible, while others say it should start after a part of the code is implemented. If some parts of the code are tested right away, the prevention of the spread of errors is ensured, reducing the SDLC.

Some teams are trying to start the testing process while the development is still in progress but haven't been successful yet.

5.4.3 Product Backlog Management

All software development teams in Vodafone use Jira as their Product Backlog management tool.

As stated by participant *C*, there must be **uniformity of the tools used by Vodafone**. The teams are already used to working with Jira, which can be suitable for the efficiency of the development process since they won't spend too much time working with it.

For this reason, most participants don't feel there is a need to change from Jira to Gitlab.

However, participant *E* feels like Gitlab's backlog management tool is better.

5.5 Problems in teams and tools benefits

Based on Table A.3, some problems in the development teams were understood and the benefits of tools like Gherkin, Cucumber and Gitlab were argued.

5.5.1 Communication issues

Regarding communication issues between different SDLC phases, **all participants said that they have suffered or are still suffering from it**.

There are moments when stakeholders do not communicate a requirement correctly to the developers. In those cases, participant *C* states that those problems can be easily fixed. However, participant *E* doesn't share that same opinion and says those communication issues are the hardest to resolve.

Scrum helps in eliminating those issues, as well as adopting a collaborative position in the team.

5.5.2 Gherkin and Cucumber usage benefits

Some participants feel that using tools like **Gherkin and Cucumber would benefit their current and future teams**.

At Vodafone, most software development **teams need to adopt automated test methods**, and tools like Gherkin and Cucumber could be a good option.

Some of the teams have already tried to use Cucumber in the past. However, it wasn't possible because they couldn't find the right way to implement it. The participants feel these tools are only helpful for specific projects, like micro-services, since in those systems is easier to implement acceptance tests.

5.5.3 Product Backlog management tool efficiency

The opinions about Gitlab's backlog management tool were somewhat mixed. Most participants weren't against the tool's usage, but neither were in favor.

Gitlab's premium version looks like it is similar to Jira. However, for teams in Vodafone to start using it, it **needed to be evaluated the added value that it could bring**.

Participant *E* feels like the tool provided by Gitlab could be beneficial and **save a lot of time** in the overall development process.

5.6 Conclusions about the tools and RE plan

The conclusions stated on Table A.4 are discussed in more detail in this sub-section.

5.6.1 Opinions about Solution's benefits to RE

All participants agree that the **solution presented in this dissertation is interesting and relevant** and that it **can bring benefits to a RE process**.

Participant *D* evens states that it goes beyond the RE process since it can be **beneficial to any development process**.

However, there must be some **investment in terms of time and money** for it to start being implemented in Vodafone's teams. They would need to hire or invest in training to have Gherkin/Cucumber experts that would be comfortable working with the tools.

5.6.2 PO's understanding of Gherkin

Most participants feel like a **PO should be able to understand what is written in Gherkin**. However, they **don't feel like a PO would be willing to write Gherkin scenarios**. They are too occupied with the business aspects that they wouldn't take the time to write the scenarios with Gherkin syntax.

In some teams, like participant's *D* team, the POs are supported by digital specialists. Digital specialists understand more about technical issues and help POs with that. Writing Gherkin scenarios could be a task that digital specialists could do.

5.6.3 Gitlab's Backlog management

Gitlab's premium version is looked at as **something that could efficiently manage the requirements**. There are still some doubts about the management of tests.

Once again, the participants referred to its utility just for a particular type of project and is raised the question if it would work in projects that deal with more than one technology.

5.7 Opinion about the overall solution

Overall, the participants feel like it's a relevant and useful solution, considering Vodafone's needs (Table A.4).

It is not a solution that can be applicable to all projects, but to the ones it can be applied to, it can fulfill its purpose.

There is the issue that Vodafone would have to invest time and money in hiring or training people specialized in the tools to apply this dissertation's plan. Because of that, adopting this plan wouldn't be easy, but it's a possibility that Vodafone is open to.

The usage of some of the tools is also questioned, especially the use of Gitlab, since Vodafone prefers the usage of Jira and is not that open to changing it soon. However, they feel like the concept of the RE plan could be implemented with other similar tools that would serve the same purpose while not changing some main tools used by Vodafone, like Jira.

5.8 Summary

Once this RE plan was validated, some of the issues that Vodafone faces and some of the benefits of this plan could be better understood.

From the analysis of the interviews, it became clear that the necessity of Reverse Engineering processes and techniques to deal with it is a reality for some of Vodafone's development teams.

While there aren't any issues with managing the requirements, quality assurance is still too manual and, consequently, too time-consuming. New types of automated tests have to be adopted by the teams, but this may mean a need for investment in more specialized people in automated testing tools.

As expected, communication problems between different SDLC phases exist in almost every team. Tools that can help mitigate those situations as much as possible are always beneficial to any team.

Chapter 6

Conclusions and Future Work

Software Reverse Engineering processes are almost inevitable in big companies, such as Vodafone, since their clients' and business needs' vast demand for new things daily makes a system outdated fast.

This work aimed to try and turn this process as efficient and less time-consuming as possible. Considering Vodafone's reality, a plan to improve the efficiency of a Software Reverse Engineering process was proposed.

With the help of three different tools, Gherkin, Cucumber, and Gitlab, the efficiency improvement plan took on the specifications of the requirements in a way it connected them to their respective tests, creating an automated testing environment. This allowed to quickly realize what already existed and was already implemented in a legacy system.

In the validation phase, it became clear that some of Vodafone's systems must go through Reverse Engineering processes. Some techniques have to be adopted to make it as efficient as possible. Furthermore, the adoption of automated tests methods is increasingly essential for Vodafone's development teams, and they need tools that can serve that purpose.

Information loss is also a reality in big companies, and the possible solution discussed in this document for that would be a platform where all project artifacts could be stored. However, development teams at Vodafone are already used to specific platforms and certain work methods, thereby a migration to a unique platform wouldn't be a viable option. That migration would most likely impact the team's efficiency, which is not desirable.

Moreover, specialists in test automation will be, sooner or later, indispensable to the development teams. Creating the automated test environment necessary for the development's efficiency demands a workforce of specialized people in the team that can ensure the quality of the product. Non-specialized people will delay the testing phase and, consequently, the overall development time.

The goal was to define a Reverse Engineering plan that would help with the possible lack of documented requirements and potential changes and additions the list of requirements could

suffer, as well as create an automated test environment to validate what already existed in the system. We believe everything was accomplished by using Gherkin and Cucumber to build an automated test environment while specifying every requirement in a way every team member could easily understand.

While the goal of this work was achieved, the analysis of the legacy system was a challenge in itself. Studying and analyzing a code of that dimension without almost any documentation to help was difficult, and extracting requirements from that was even more difficult. In the end, all the effort was worth it, but even so, it wasn't possible to analyze all the system, and therefore, not all system was tested. That part will have to be continued over time.

Overall, the solution succeeded with extreme relevancy and importance to Vodafone's needs. While not every type of project can benefit from tools like these, it is still a very appropriate solution for some projects, like the one studied in this dissertation (micro-services). It was also seen that the solution went beyond the purposes of a Reverse Engineering process. It is something that can be relevant in any development process and help with the requirements management of any SDLC.

This work showed the importance of a well-structured plan for a development team. Requirements management is essential in any development process, and defining suitable testing methods is crucial for an efficient development process.

For future work, we believe that it would be beneficial to explore, with the usage of Gherkin and Cucumber still, the concept of *Live Requirements*, where an entire development environment is created in a way it would automatically know if the requirements are being met, even if changes occur, and also knows before-end if the required changes are feasible to make.

Furthermore, it would also be beneficial to explore ways of getting the project's requirements from its source code in the most efficient/automatic way. This could reduce the time of a Reverse Engineering process a lot more while also reducing its costs.

Appendix A

Interviews

In this appendix, the questions of the interviews and the answers of the interviewees are disclosed.

A.1 Questions

A.1.1 Characterization Questions

1. Current role in a team.

2. How many years of experience as a PO (or similar role).

3. Context of current team:

   (a) Methodology used.

   (b) Platforms used (e.g., jira, gitlab, etc).

   (c) Testing Methods used.

A.1.2 General Questions

1. How important do you consider the requirements gathering process at the beginning of SDLC to be? Do you consider requirements gathering as being an essential step (if not the most essential) to the success of the final product? Why?

2. How important do you consider software testing to be? In the context of the workflow of a SDLC, do you feel like it's better to implement tests before development begins or after the coding part is done? Why?

3. What tools do you prefer to manage a backlog and why? Do you feel like the one offered by Gitlab is efficient?

A.1.3 Team-Oriented Questions

1. Have you ever been part of a team that had communication issues between different stages of the SDLC? How easy is the communication between the different stages of the SDLC in your current team?
2. Do you think your current or past teams would benefit from the usage of tools like Gherkin and Cucumber? Why?
3. Do you feel that the way of using Gitlab to manage user-stories and tests like I showed you would be beneficial for your current or past teams?

A.1.4 Project-Oriented Questions

1. As an experienced PO (or similar), what do you feel about the proposed work? Do you think it would benefit a Reverse Engineering process, such as the one that I've showed you? Why?
2. Do you feel like a Product Owner that has no previous experience with coding can understand well what is written in Gherkin? If yes, do you feel like he would be able to write Gherkin scenarios by himself?
3. When it comes to the management of User-stories and tests with Gitlab, do you think it was efficient in ensuring good management and traceability of the requirements?
4. Overall, thinking about all the context of this thesis and Vodafone's reality, what is your opinion about the proposed solution?

A.2 Answers

The five participants of this interview are identified as interviewees *A*, *B*, *C*, *D*, and *E*. Their answers can be checked out in the following tables:

1. Interviewees Characterization Questions in Table A.1.
2. General Questions in Table A.2.
3. Team-Oriented Questions in Table A.3.
4. Project-Oriented Questions in Table A.4.

Characterization Questions					
	Role	**Years of experience**	**Methodology**	**Platforms**	**Testing Methods**
A	Product Owner	10	Agile/Scrum	Jira Confluence Teams	Mocking Testing. Testers go step by step through app to see if it equals acceptance criteria
B	Scrum Master/ Product Owner	8	Agile/Scrum	Jira Confluence Gitlab	Acceptance Tests with Jira's *Xray*
C	Team Leader. More in the technical field	20+	Agile/Scrum	Jira Git Gitlab	Acceptance Tests by Client's Area
D	Chapter Leader	3+	Agile/Scrum/ Kanban	Jira Gitlab Github Confluence	Manually Testing
E	Team Leader. More in the technical field	20+	Agile/Scrum	Jira Git Gitlab	Unit Testing

Table A.1: Interviewees Characterization Questions

General Questions				
	Importance of Requirements Gathering	**Importance of Software Testing**	**Tests before or after implementation**	**Product Backlog management tools**
A	Important as the first step of a development process	Essential	Should start at an early stage	Isn't familiar with other tool besides Jira. Gitlab's Backlog looks simpler
B	The team should not worry too much with the requirements specification at the beginning. Defining goals is more important	Essential. Extremely important	Starts at the beginning with tests criteria. However, tests are only implemented after development	Not familiar with Gitlab. Feels like Gitlab should be the repository of the code and Jira the repository of the requirements
C	Critical process. Requirements are insufficient at the beginning because of lack of time, and it makes the quality of the final product falter	Very Important	As parts of the code are developed they must be tested right away to prevent the spread of errors and ensure the reduction of the SDLC	There must be uniformity of the tools used in Vodafone's teams. Gitlab's management is interesting, but is not really necessary
D	Essential. Trying to specified the most requirements at the earliest phase is the best	Extremely Important	Team is trying to implement tests while developers work on the product to increase efficiency	Vodafone uses Atlassian platforms, like Jira, and things have been going well so far, since the team's members are specialized in it, and being comfortable with the platform is the most important. Right now, using Gitlab's tool wouldn't bring any benefit
E	Clearly important to build and structure a project	Important	Before. If you have the requirements you should be able to write tests right away and fix some things later on	Usage of Jira. Feels like the one offered by Gitlab is better

Table A.2: General Questions about Requirements, Tests and Gitlab

Team-Oriented Questions			
	Communication issues between different SDLC phases	**Benefits from Gherkin and Cucumber usage**	**Gitlab for Backlog management**
A	Had those issues	Current team would benefit from it. There's a need for automated tests	Jira is Vodafone's official management tool. Gitlab's premium version seems as good as Jira
B	Has problems with it. There's a need for a collaboration position between all team members	Would bring benefits. Past team already tried the use of Cucumber. Didn't work because of technical restrictions	Could satisfy the flow of the process aspect
C	Can happen that a person misunderstands a requirement, but that is easily fixed	Brings benefits to specialized micro-services	It could maybe bring benefits
D	There's still some of those issues, but not as much when working with Waterfall	Teams would benefit from it. Current team already tried to use them, but didn't find the right way to implement it	The added value level of its use would have to be assessed
E	Been part of a team with those issues. However, those issues are only more difficult to fix when it's the stakeholders' communication to the team	Future teams will benefit from the usage of those tools. Current team not (because of its small size)	Seems useful. Seems like it could potentially save a lot of time

Table A.3: Team-Oriented Questions about Requirements, Tests and Gitlab

Project-Oriented Questions				
	Solution's benefits to RE	**PO understanding what is written is Gherkin**	**Gitlab's efficiency for Backlog management**	**Opinion about Solution**
A	It's important, because sometimes there is a lack of documentation	With due dedication and learning time, the PO could understand it. Writing would be a little harder but not impossible	Gitlab's premium version should be efficient	If Gitlab can overcome Jira, it would be beneficial to change one for the other. With qualified QA people it would be beneficial
B	Quite important and relevant. However, there must be time and investment	Believes he could understand. Not write	Good for management of the requirements, but not so much for the tests	Quite relevant because of Vodafone's necessity. The tools can be discussed and use similar ones, but the concept is really important
C	Solution is interesting and possibly beneficial for a micro-service project	Could understand but not write a Gherkin scenario	Could have advantages just to a few types of projects	Not applicable to all projects, but definitely to some projects or some parts of projects
D	Solution with high interest that ends up going beyond the RE process	Non-technical PO's don't really like to write the requirements. They are too focused on the business part of their work, but they are supported by digital specialists and could delegate these tasks to them	It would be efficient, but doesn't know it it would work for teams that deal with multiple technologies	Liked the solution. Some Vodafone's platforms can really benefit from it
E	Beneficial. It speeds the process and can potentially eliminate miscommunication issues	The PO should understand what is written. However, it would be a bit more challenging for him to write the Gherkin scenarios alone	Ensures good management and traceability	Useful. Directed to a certain target group

Table A.4: Project-Oriented Questions about Requirements, Tests and Gitlab

References

[1] Cradle. http://www.threesl.com/.

[2] Gherkin reference.

[3] Is the agile manifesto still a thing? | atlassian. Atlassian. https://www.atlassian.com/agile/manifesto.

[4] Jama software. https://go.jamasoftware.com/manage-requirements-free-trial.html?utm_source=digital-project-manager&utm_medium=ranking-site-paid&utm_campaign=best-rm-software-list&utm_content=free-trial-url&dpm=39466.

[5] Scrum patterns. https://scrumbook.org/.

[6] Use GitLab | GitLab.

[7] What is agile? Atlassian. https://www.atlassian.com/agile.

[8] What is gherkin + how do you write gherkin tests? Functionize. https://www.functionize.com/blog/what-is-gherkin-how-do-you-write-gherkin-tests.

[9] What is scrum? Scrum.org. https://www.scrum.org/resources/what-is-scrum.

[10] Request evaluation form requirements management - visure solutions. https://visuresolutions.com/request-evaluation-form-requirements-management/?utm_source, 2019.

[11] Overview of doors. Ibm.Com. https://www.ibm.com/docs/en/ermd/9.7.0?topic=overview-doors, 2021.

[12] Samuel A Ajila. Change management: modeling software product lines evolution. In *Proc. of the 6th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, Florida*, pages 492–497, 2002.

[13] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[14] Atlassian. User Stories | Examples and Template.

[15] Atlassian. User stories: Examples and template.

[16] Atlassian. What is jira software used for?

[17] Daniel M Berry and Brian Lawrence. Requirements engineering. *IEEE software*, 15(2):26–29, 1998.

[18] Elizabeth Bjarnason, Per Runeson, Markus Borg, Michael Unterkalmsteiner, Emelie Engström, Björn Regnell, Giedre Sabaliauskaite, Annabella Loconsole, Tony Gorschek, and Robert Feldt. Challenges and practices in aligning requirements with verification and validation: a case study of six companies. *Empirical software engineering*, 19(6):1809–1855, 2014.

[19] Shawn A Bohner et al. Impact analysis in the software change process: a year 2000 perspective. In *icsm*, volume 96, pages 42–51, 1996.

[20] Lan Cao and Balasubramaniam Ramesh. Agile requirements engineering practices: An empirical study. *IEEE software*, 25(1):60–67, 2008.

[21] Alan Davis, Oscar Dieste, Ann Hickey, Natalia Juristo, and Ana M Moreno. Effectiveness of requirements elicitation techniques: Empirical results derived from a systematic review. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 179–188. IEEE, 2006.

[22] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Elsevier, 2002.

[23] Armin Eberlein and JCSP Leite. Agile requirements definition: A view from requirements engineering. In *Proceedings of the international workshop on time-constrained requirements engineering (TCRE'02)*, pages 4–8, 2002.

[24] Khaled El Emam, Dirk Holtje, and Nazim H Madhavji. Causal analysis of the requirements change process for a large system. In *1997 Proceedings International Conference on Software Maintenance*, pages 214–221. IEEE, 1997.

[25] Syed Ahsan Fahmi and Ho-Jin Choi. Software reverse engineering to requirements. In *2007 International Conference on Convergence Information Technology (ICCIT 2007)*, pages 2199–2204. IEEE, 2007.

[26] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[27] Jun Han. Tram: A tool for requirements and architecture management. In *Proceedings 24th Australian Computer Science Conference. ACSC 2001*, pages 60–68. IEEE, 2001.

[28] Børge Haugset and Geir Kjetil Hanssen. Automated acceptance testing: A literature review and an industrial case study. In *Agile 2008 Conference*, pages 27–38. IEEE, 2008.

[29] Jane Huffman Hayes, Alex Dekhtyar, Senthil Karthikeyan Sundaram, E Ashlee Holbrook, Sravanthi Vadlamudi, and Alain April. Requirements tracing on target (retro): improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering*, 3(3):193–202, 2007.

[30] Hennie Huijgens, Arie Van Deursen, and Rini Van Solingen. The effects of perceived value and stakeholder satisfaction on software project impact. *Information and Software Technology*, 89:19–36, 2017.

[31] Waqar Hussain, Didar Zowghi, Tony Clear, Stephen MacDonell, and Kelly Blincoe. Managing requirements change the informal way: When saying 'no'is not an option. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 126–135. IEEE, 2016.

[32] Shalinka Jayatilleke and Richard Lai. A systematic review of requirements change management. *Information and Software Technology*, 93:163–185, 2018.

[33] Andrew Johnson-Laird. Software reverse engineering in the real world. *U. Dayton l. rev.*, 19:843, 1993.

[34] Atsushi Kobayashi and Mamoru Maekawa. Need-based requirements change management. In *Proceedings. Eighth Annual IEEE International Conference and Workshop On the Engineering of Computer-Based Systems-ECBS 2001*, pages 171–178. IEEE, 2001.

[35] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. John Wiley & Sons, Inc., 1998.

[36] W Lam, V Shankararaman, S Jones, J Hewitt, and C Britton. Change analysis and management: a process model and its application within a commercial setting. In *Proceedings. 1998 IEEE Workshop on Application-Specific Software Engineering and Technology. ASSET-98 (Cat. No. 98EX183)*, pages 34–39. IEEE, 1998.

[37] Dean Leffingwell and Don Widrig. *Managing software requirements: a unified approach*. Addison-Wesley Professional, 2000.

[38] Simon Lock, Gerald Kotonya, et al. An integrated, probabilistic framework for requirement change impact analysis. *Australasian Journal of Information Systems*, 6(2), 1999.

[39] Linda A Macaulay. *Requirements engineering*. Springer Science & Business Media, 2012.

[40] Manar Majthoub, Mahmoud H Qutqut, and Yousra Odeh. Software re-engineering: An overview. In *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, pages 266–270. IEEE, 2018.

[41] Minna Mäkäräinen. Software change management processes in the development of embedded software. 2000.

[42] Roy Miller and Christopher T Collins. Acceptance testing. *Proc. XPUniverse*, 238, 2001.

[43] Neil C Olsen. The software rush hour (software engineering). *IEEE Software*, 10(5):29–37, 1993.

[44] Ipek Ozkaya and Ömer Akin. Tool support for computer-aided requirement traceability in architectural design: The case of designtrack. *Automation in Construction*, 16(5):674–684, 2007.

[45] Dhirendra Pandey, Ugrasen Suman, and A Kumar Ramani. An effective requirement engineering process model for software development and requirements management. In *2010 International Conference on Advances in Recent Technologies in Communication and Computing*, pages 287–291. IEEE, 2010.

[46] Linda H Rosenberg and Lawrence E Hyatt. Software re-engineering. *Software Assurance Technology Center*, pages 2–3, 1996.

[47] Ken Schwaber and Jeff Sutherland. The scrum guide. *Scrum Alliance*, 21(1), 2011.

[48] Muhammad Shahid, Suhaimi Ibrahim, and Mohd Naz'ri Mahrin. An evaluation of requirements management and traceability tools. *World Academy of Science, Engineering and Technology, WASET*, pages 1–6, 2011.

[49] Jaswinder Singh, Kanwalvir Singh Dhindsa, and Jaiteg Singh. Performing reengineering using scrum agile framework. In *2020 Indo–Taiwan 2nd International Conference on Computing, Analytics and Networks (Indo-Taiwan ICAN)*, pages 33–35. IEEE, 2020.

[50] Jaswinder Singh, Kanwalvir Singh, and Jaiteg Singh. Reengineering framework to enhance the performance of existing software. *System*, 1139:120–3, 2019.

[51] MR Strens and RC Sugden. Change analysis: a step towards meeting the challenge of changing requirements. In *Proceedings IEEE symposium and workshop on engineering of computer-based systems*, pages 278–283. IEEE, 1996.

[52] Ladan Tahvildari, Kostas Kontogiannis, and John Mylopoulos. Quality-driven software reengineering. *Journal of Systems and Software*, 66(3):225–239, 2003.

[53] A Terry Bahill and Steven J Henderson. Requirements development, verification, and validation exhibited in famous failures. *Systems engineering*, 8(1):1–14, 2005.

[54] Jessada Tomyim and Amnart Pohthong. Requirements change management based on object-oriented software engineering with unified modeling language. In *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 7–10. IEEE, 2016.

[55] Karl Wiegers and Joy Beatty. *Software requirements*. Pearson Education, 2013.

[56] Marvin Zelkowitz. *Advances in computers: advances in software engineering*. Elsevier, 2004.