Fabian Gubler

Refactoring of a Software System for Industry 4.0

Bachelor Thesis

to achieve the university degree
Bachelor of Arts in Business Adminstration

submitted to
University of St. Gallen

Supervisor
Prof. Dr. Ronny Seiger

Institute of Computer Science

May 2022

Abstract

This is a placeholder for the abstract. It summarizes the whole thesis to give a very short overview. Usually, this the abstract is written when the whole thesis text is finished.

Contents

List of Figures

1 Introduction

2 Theoretical Framework for Code Refactoring

2.1 Background

Refactoring is a well established concept in software development. Especially in complex projects, tasks like cleaning and restructuring code are regularly occurring and often times necessary. When referring to these activities by name, many employees use the term refactoring. The exact definition of the term *refactoring* is not always self-evident, despite its familiarity. To avoid referring to the term too loosely, it is consequently important to give a precise definition as a reference for this thesis.

Martin Fowler (2018, p. xiv) managed to formulate a definition that is both short and precise. He is one of the most prominent figures in the field of refactoring, and has pioneered many concepts, which this thesis is going to discuss. Fowler defines refactoring in the following:

> **Refactoring** is the process of changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure.

According to this definition, refactoring should not change the features of the program. By proper refactoring, we can thus alleviate the risk associated with changing the codebase, such as the creation of new bugs.

2.1.1 Benefits: Why should we refactor?

The benefit of refactoring is not as obvious as other activities in software development. Spending valuable resources on a process that does not add new functionality, might sound unappealing to many managers and software engineers. Likewise, Kim et al. (2012, p. 1) add the problem of not sensing an immediate benefit when refactoring. Furthermore, the value of improving the internals of the code is hard to show to a manager, who is not an expert and even harder to present to the client, who is paying for the work.

Having said that, when refactoring is ignored, there is not necessarily more time available. Through continuous modifications and adaptations to new requirements, the code becomes increasingly complex and drifts away from the original design. By not improving the quality of the software, which would counteract this complexity, a major part of the resources is still spent on software maintenance (Mens et al., 2003, p. 1).

Having Fowler's Definition in mind, we thus aim to mitigate time spent on tedious maintenance, and rather improve the quality attributes of the code beforehand. Neglecting such work, would increase debt, which would have to be repaid in the future in the form of costly maintenance costs. Moreover, by obtaining this so-called *technical debt*, future features are more expensive to implement, and sudden changes are near impossible. The concept of technical debt is important to fully grasp, which is why it is further explained in a future section of this thesis (see Section).

In general, refactoring helps to improve the internal quality attributes of the software (Mens and Tourwé, 2004, p. 129). For instance, some refactorings remove code redundancy, some raise the level of abstraction, some enhance the reusability, and so on. Bass (1998) distinguishes between two types of quality attributes. One type is concerned with the observation of the system at runtime, such as performance and security, whereas the other type does not consider system runtime. During refactoring, we are mostly concerned with the latter, meaning we ignore the performative aspect of software during runtime.

During a study on the value of quality attributes on refactoring, Alkhazi et al. (2020, p. 4) identified six of such quality attributes: Reusability, Flexibility, Understandability, Functionality, Extendibility, and Effectiveness. We will not go into detail of each attribute, but it is still important to remember some of them, to measure whether the refactoring improved the internal quality of the software. In addition, choosing the relevance of quality attributes, highly depends in particular on the project at hand. Fortunately, even without understanding the specificities of these attributes, programmers can get a feel of what code with high quality is. As a rule of thumb, if it takes the programmer a week to make a change that would have taken only an hour with proper quality, we can assume that most of these attributes are at least partly not fulfilled.

The advantage of knowing quality attributes, is the ability to evaluate refactorings. In particular, the effect of a refactor can be estimated, by the level of improvement in these quality attributes. Again, the transformation of the internal structure is the deciding factor on which success is measured. Refactoring adds value if at least one quality attribute has significantly improved.

As previously mentioned, there is one requisite when measuring the added value. The external behavior must not altered. As an example, changing the code base could make the program more modular, but during the process create bugs that didn't exist before. Accordingly, to make meaningful evaluations, it must be proven that only the internal structure has changed. This can be achieved by writing software tests beforehand and then testing the features after the refactor. Consequently, software tests is a major component when deciding the success of a refactor.

2.1.2 Criteria: When should we Refactor?

From a business standpoint, there needs to be little to no explanation, when debating time spent on new features and fixing unresolved bugs. As discussed earlier, it is significantly harder to persuade doing a refactor, as there are fewer immediate benefits. For that reason, one needs to verify the necessity of a refactor and compare it to the need of other pending tasks. Fowler (2018, p. 5) points out that if the code works and doesn't ever need to change, it's fine to leave it alone. He suggests that as soon as someone needs to understand how the code works, and struggles to follow it, one has to do something about it. Accordingly, a potential improvement in quality, doesn't warrant a refactor by itself.

It is crucial to note that refactoring does not have to be a massive, obscure undertaking. Even if there is a demand for a huge refactor, the refactorings themselves are minor. Refactoring is all about applying small behavior-preserving steps and making a big change by stringing together a sequence of these behavior-preserving steps (Fowler, 2018, p. 45). For this reason, refactoring as a principle, always plays a key role in software development, as code being easier to understand and cheaper to modify is vital. Therefore, with projects that appear to be qualitatively sufficient, active monitoring of the code quality and continuous improvements through refactorings are still advisable.

This approach leads to the presumption that these activities can be done in parallel. It is, however, important to note, that this parallelism doesn't suggest that one can add features and refactor at the same time. The interplay between them can be described making use of an analogy (Fowler, 2018, p. 47). Fowler sees software development as wearing two hats, proposing that time is divided between adding functionality and refactoring. He argues that these activities should be distinctively separated, meaning that during a refactor one should not add functionality and vice versa. To support his point, Fowler adds that "often the fastest way to add a new feature is to change the code to make it easy to add" (Fowler, 2018, p. 53) Conversely, once the code is better structured, time can be efficiently spent on adding new capabilities.

This approach illustrates nicely that refactoring should be an integral part of software development according to Fowler's standpoint. Moreover, it challenges the frequent notion of refactoring being cleaning up quality deficient code and fixing past mistakes.

Once the relationship between refactorings and adding features is understood, one can better estimate the timing of each activity. Even though we learned that refactoring at its best is fairly small, by understanding the precise relationship, we can argue that planned refactoring is not always a mistake. In more concrete terms, if we agree that better structured code allows us to add features quicker, we can infer that there could be moments where dedicated time spent on refactoring is necessary to get the code base into a better state for new features (Fowler, 2018, p. 53).

Contrasting large and potentially complex refactors with our previous definition of refactoring being smaller steps, we realize that there exists a difference in scope. The difference in scope being precisely the amount of small refactoring steps needed to be applied. Therefore, with certain projects, doing continuous refactoring during the development process might not suffice, and an overarching refactor is necessary.

According to these presumptions, the conclusion would be that with increased code quality small refactors suffice, whereas with a lack of code quality larger refactorings are required. Although this is typically valid, there are exceptions one needs to keep in mind. Indeed, there are scenarios, which challenge both assumptions.

The assumption that small and continuous refactor suffice, for code bases with high quality, does not hold if one of the quality attributes gains significantly in importance, Such a situation could justify a larger refactor of code, even though it is perceived as being high in quality. This is typically the case, with approaching of large transformations to the code base. For example, unprecedented security risks might demand the code base to be more testable. Another case would be a migration to the cloud, which would require the code being high in modifiability. Consequently, a larger refactoring would then be used to satisfy these requirements.

The second assumption was that code that clearly lacks quality would inevitably result in a lot of time spent in refactoring. In contrast, if a part of code is relatively insignificant, an argument for a planned refactoring is difficult to be made when accounting the associated costs. Here, minor restructurings and bug fixes could suffice. However, once the part gains in importance, a larger refactoring could then be considered and might be necessary. This means that successful refactoring can also be thought of allocating the companies' resources efficiently. Having a business case for refactoring is such an important topic, that it will be discussed later in section 3.

In summary, it is safe to say that large refactoring efforts and tedious maintenance in post is not desirable. This is because ignoring refactoring, even though it is required, will result in costs and risks. The ideal case is, then, that decisions are conducted in advance by establishing refactoring as an integral part of the software development process. Refactor can then be thought of as a potential investment that would be repaid in the future.

2.1.3 Challenges: What to watch out?

With a thorough understanding of the complexities such as benefits, scope and timing of refactoring, the effective act of refactoring seems fairly easy in comparison. In other words, the actual difficulties might be related to the planning rather than the implementation.

The difficulties of planning become even more evident, when working within teams. Going back to the analogy of the two hats, we assumed that the software development process is conducted by an individual. In practice, however, the development is typically done with more than one person. This makes refactoring considerably more difficult, knowing that refactoring and adding new features should be a distinct activity. Thus, a programmer decides to refactor in a team, said programmer could inhibit others from working on the code at the same time. In other words, it is particularly easy to swap hats when coding alone, but much more difficult when the coding is done in teams. As a result, it is a challenge to clearly separate which part of the code base is being refactored, and which is added functionality. The underlying goal is to reach a state, where the separate parts are entirely independent. When ignoring this division, the risk of bugs being introduced becomes much higher. As a consequence, the essential premise of refactoring, being that no observable behavior is changed, cannot be achieved.

These challenges related to refactoring may sound intimidating and time-consuming at first. Fowler Fowler (2018, p. 56) argues against this belief, by suggesting that the whole purpose of refactoring is to speed things up. The key is, as with most software-related work, to not do it blindly, and try to do the refactoring with intent. This means to not only focus on the practical steps of refactoring, but also to always have an overview of the entire project. Situating refactoring within the broader scale of the project allows to actively decrease risks and enables to focus on the overall quality of the code base. Even though refactoring is often done on a small scale, its effects can be examined throughout the layers of the software architecture. This is especially apparent when incrementally improving previously discussed attributes, such as Extendibility, Reusability, and Flexibility, which in turn affect the whole state of the project.

In practice, "too little refactoring is far more prevalent than too much" (Fowler, 2018, p.56). The suggestion that people should attempt to refactor more often, is, however, much easier said than done. Besides the mentioned challenges, refactoring can add considerable value, with the precondition that it is properly carried out. Carrying out refactorings, however, is not always an easy task, requires education, and most definitely experience. This can indeed be seen as another challenge, which depends on the competences of the software engineers who are actively involved.

2.2 Formalization of Code Smells

Knowing just the importance and applications of refactoring does not suffice, as one still needs to understand the indications that lead to an individual refactoring. Fowler (2018, p. 71) argues that deciding when to start refactoring, and when to stop, is just as important to refactoring, as knowing how to operate the mechanics of it. Particularly, there is no clear-cut moment when to refactor, there are only indications that there is trouble that can be solved by a refactoring. In practice, these indications are known as code or design smells (Lacerda et al., 2020, p. 2). These two terms are distinguished by being either located at a lower level, known as the code level, or on a higher level, the design level. Hence, depending on the degree of complexity, the smells are named code smell or design smell. As throughout this work we are not concerned with design decisions of the codebase, we will from now on refer to the indications as code smells.

The term "smell" is used in reference to an internal software problems Lacerda et al. (2020, p. 2), which can negatively impact software quality

(Sonnleithner et al., 2021, p. 1). One might think that software bugs fall into this category, but this is a false assumption. Although bugs also negatively impact the state of software, code smells do not necessarily cause the application to break. Nonetheless, these internal problems may lead to other negative consequences, such as the "impacting on software maintenance and evolution" Lacerda et al., 2020, p. 2.

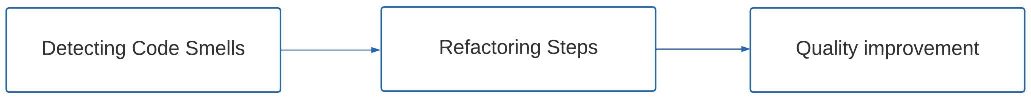

Figure 2.1: Simplified model of Refactoring

Knowing that code smells refer to internal problems, it becomes apparent that they are linked to refactoring, which tries to improve the internal state of the software. In other words, refactoring can be thought of getting rid of code smells.

To better comprehend these code smells, it is best to list some of them. However, to stay within the scope of this thesis, we will limit the examples to the most prominent ones. In their work, Fowler and Beck introduced a total of 24 code smells to avoid. Therefore, the following overview briefly summarizes the top ten most frequently reported code smells according to [p. 15]Lacerda et al. (2020).

2.2.1 Code Smells

Summary of code smells identified by Fowler and Beck Fowler (2018)

Code Smells	Description
Duplicated Code	Consists of equal or very similar passages in different fragments of the same code base.
Large Class	Class that has many responsibilities and therefore contains many variables and methods.
Feature Envy	When a method is more interested in members of other classes than its own, is a clear sign that it is in the wrong class
Long Method	Very large method/function and, therefore, difficult to understand, extend and modify. It is very likely that this method has too many responsibilities, hurting one of the principles of a good OO design
Long Parameter List	Extensive parameter list, which makes it difficult to understand and is usually an indication that the method has too many responsibilities.
Data Clumps	Data structures that always appear together, and when one of the items is not present, the whole set loses its meaning
Divergent Change	A single class needs to be changed for many reasons. This is a clear indication that it is not sufficiently cohesive and must be divided
Shotgun Surgery	Opposite to Divergent Change, because when it happens a modification, several different classes have to be changed
Lazy Class	Classes that do not have sufficient responsibilities and therefore should not exist

Despite having an infinite amount of potential code smells, the list gives a good overview of internal problems. Being aware of these pattern is crucial in order to achieve the goals of improving the state of software quality. Ignoring these smells might not seem to be that big of a problem. In cumulation, however, getting rid can have a significant effect. Furthermore, looking at it from a broader perspective, code decay inhibits the software development project. Refactoring then becomes a tool for programmers to invert this decay. If ignored, this effect becomes harder and harder to manage, as it is increasingly difficult to both change the internal state and add features, with software that is lacking quality. This is why it is essential not to do refactorings in stages, but in a simultaenous manner. Disregarding this, leads to the accumulation of debt, which is formaly known as technical debt. The concept of technical debt will be the main theme in the following part, which focusses on the business case of refactoring.

Despite having an infinite amount of potential code smells, the list gives a good overview of possible internal problems. Being aware of such pattern is crucial to improve the state of software quality. The act of ignoring a code smell, might not seem to be that big of a problem at first. Cumulatively, however, getting rid of smells does have a significant effect. From a broader perspective, not acting on internal problems results in code decay, which in turn inhibits the software development project.

Refactoring then becomes a tool for programmers to be able to invert code decay. In other terms, neglecting code smells becomes harder and harder to manage, as it is increasingly difficult to both change the internal state and add features, with software that is lacking quality. This is why it is essential not to do refactorings in stages, but in a simultaneous manner. Disregarding this, leads to the accumulation of debt, which is formally known as technical debt. The concept of technical debt will be the main theme in the following part, which focuses on the business case of refactoring.

3 The Business Case for Refactoring

During the lifetime of a project, one has to inevitably deal with internal problems appearing in the software systems. While having these code smells in our programs, the company has to suffer in terms of costs that we can only be mitigated by getting rid of the smells by refactoring. A prototypical example of a common financial cost is the additional time spent, whilst programming with deficient code. Naturally, employees are spending less time when their code is easy to read, understand, and modify. Conversely, through the accumulation of code smells in software systems, companies are exposed to a financial burden they might want to get rid of. Examining this discussion between code smells and financial costs paves the way to consider refactoring decisions from a business standpoint. The relationship will be the subject in the following part of the thesis.

3.1 Technical Debt

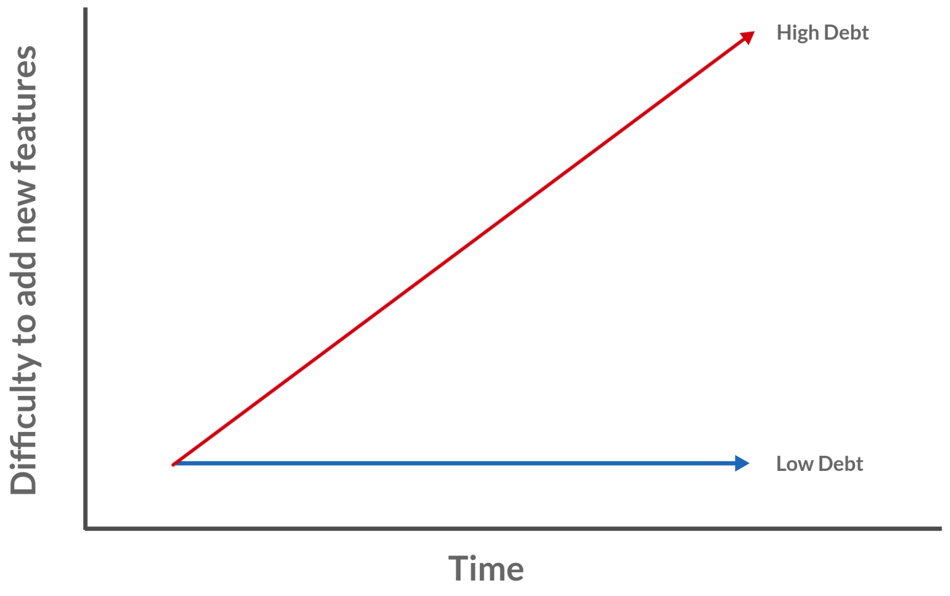

Figure 3.1: Simple depiction of Technical Debt

Technical debt is a helpful metaphor that provides a framework that allows thinking about code smells in terms of financial debt (Fowler, n.d.). In concrete terms, the term Technical debt, coined by Ward Cunningham, describes the extra effort it takes to add new features (Kruchten et al., 2012). In financial terms, the additional effort is the interest that has to be paid on the debt (Fowler, n.d.). Fowler presents the example of a confusing module structure in a code base. He illustrates that a programmer could need four days to add new features with a clear structure and six days with code smells. Then, the two additional days is the interest a company has paid on the debt. We see that a company can either decide to pay the interest on technical debt, or reduce it by means of refactoring. Within the context of our thesis, when referring to technical debt, it is implied code smells are the subject of discussion. In practice, however, Technical debt must not be limited to code smells. The overarching metaphor of Technical debt encompasses anything that adds friction from software development endeavors (Kruchten et al., 2012).

One approach to think about solutions is to argue for investing in experienced programmers to avoid debt entirely. A company could then do such an undertaking, if the additional money spent on the salaries is offset by the cost endured from debt. In practice, it is not that simple, as technical debt will always accrue, even with experienced programmers. For instance, sometimes debt accumulates because one quality attribute gained in importance, but the code did not change. Other times it could just be due to the fact that code base exists for a very long time. In both cases, refactoring can be a crucial mechanism in order to get rid of acquired technical debt.

There can be instances where technical debt is substantial enough, where associated interest payments can not be handled. In some cases, it might be more cost-effective to rewrite an entire code bases instead of refactoring it. As a consequence, it becomes apparent, that businesses have financial incentives to make the right decisions in order to efficiently allocate their resources. Such difficult decisions require competent employees that on the one hand know in what ways to deal with the debt, but more importantly to prevent such cases from happening completely. These employees are not confined to one type of job As a consequence, lacking the relevant expertise in either management or software engineering departments can lead to undesirable financial ramifications. Such conclusions encourage us to look at refactoring from a business standpoint. As a consequence, the section below will explore this topic even further.

3.2 Communication Device

Being aware of the financial relationship, we see that refactoring must not be restricted to programmers and can be much more relevant to upper team leaders, or product managers. Having knowledge of abstract concepts, like technical debt, allows employees from other domains to make decisions and communicate them, without any programming experience. It can be extremely freeing not to rely on moral principles, but instead provide an evaluation that an also be defended from a business standpoint. Moreover, some might even argue that decisions regarding refactoring should be purely economic. [p.57]Fowler (2018) claims that it is beneficial to have this attitude and argues that economic benefits should always be the driving factor of refactoring.

As alluded to above, Technical debt can also be used as a communication device. It is a simple model to form economic decisions, that are potentially very complex. This device can be used in two directions, top-down or bottom-up. Top-down communication by experienced team leaders using the term technical debt provides directions to their team members. As an example, management has decided to migrate their systems to the cloud, but it is realized that the code contains a lot of code smells, which makes it hard to undergo such a change. Consequently, by communicating this issue in regard to having too much technical debt in that specific domain. This aids employees in knowing the particular area to focus on, and prevents them to refactor at places that do not directly contribute to the migration.

Additionally, technical debt as a metaphor can also support bottom-up communication. This metaphor can be used to provide sufficient arguments for refactoring, when clients or upper management are in doubt. For instance, it could serve as a tool to reason about the difficulties of adding new features or the issues of making major changes to the application. As a result, barrier are reduced between software engineers, other departments, and clients, when discussing the underlying problem of code smells. Furthermore, with proper communication, clients, for instance, might even be encouraged to further reduce technical debt in order for the software to be adaptive to upcoming trends or customer requests. Moreover, keeping technical debt low prevents clients from destroying current value and keep their product to stay successful over the long term.

4 Thesis Context and Design

4.1 Fischertechnik Factory Control

[Would be nice to have a picture of the Factory here]

In the section below, a smart factory produced by Fischertechnik and a software system that controls it will be presented to provide context for upcoming sections. Although not yet introduced, both will be referred to numerous times throughout the following sections. First, the hardware is introduced very briefly. Second, the software system will be covered in more detailed. We will focus more on the software component, as many of the concepts are primarily applicable to the software system.

The following parts take on a more practical approach than the previous parts. The thesis focuses on one particular software system. By means of this restriction, it should be noted that as a consequence, we reach conclusions that might not be applicable to all software systems. Although certain assumptions will not be generalizable to all software system, through this approach, we are able to provide much more practical insights and are able to present detailed description on the basis of the application. Additionally, this restriction requests to take design decisions on the preconditions the software system demands. Forced to take such decisions further provides valuable insights to a more practical setting.

One specific restriction of the software system is its deep integration to hardware components. The factory, being a cyber-physical system (CPS), is bound to its physical environment. The present software system enables users to interact with the factory with HTTP requests, including executing certain events and getting sensory information. These smaller interactions in turn allow the creation of complex business processes. Further, multiple processes have been formalized through Business Model and Notation (BPMN), allowing these processes to be automatically executed.

[This part must definitely be expanded, in order to provide a better overview to the reader] [Industry 4.0 was not explained nor mentioned throughout the thesis (even though it is in the title)]

4.2 Motivation and Objectives

As was evident in the theoretical framework and the business case, the sphere of refactoring should not be limited to the scope of the mere implementation of practical refactoring measures. We saw that refactoring influences various aspects of the software development process and offers multiple opportunities from a business perspective.

In addition to the theoretic contributions made in the previous chapter, the primary aim of the following work is to provide a comprehensive view on refactoring. It will be argued that refactoring must be placed in a much broader context, having multiple software domains, such as the detection of smells, testing, and measurements. The emphasis will be put on connecting several aspects into one model, the importance of each domain in regard to refactoring will be discussed. These insights aim to provide a basis for other refactoring endeavors and should encourage discussions to further explore the topic. Moreover, the thesis should offer applicable advice directly related to similar software systems in the area of CPS. It is not the focus of this thesis to deliver a refactored product, nor to record details of practical execution. On the contrary, the thesis focuses on exploring the intricate essence of refactoring and tries to discover relationships between different aspects within the context of software development.

In order to provide this comprehensive view on refactoring, the thesis follows several objectives. On a conceptual level, the thesis aspires to develop an illustrative model of a refactoring process, where refactoring itself is only one component of a broader paradigm. Here, it is important to present how other parts of the software development process influence the success of refactoring. By having an understanding of this model should promote an understanding that separate activities are a fundamental prerequisite to appropriately refactor code. In other words, an argument should be provided that refactoring can not be thought as a distinct activity. Along with developing a refactoring process, the thesis has the objective to develop an appropriate method for each stage of this model. Methods are chosen in conjunction with the software system at hand. Here, the design decisions that led to the selection of the methods should be clearly visible. This is vital, to allow transferring the acquired knowledge to other projects. Lastly, an important objective for the thesis is to think critically of each decision by discussing limitations of the methodological approach provided. The processes effectiveness and validity should not only be measured in regard to the present software system, but also measured with respect to the generalizability to software systems in general.

5 Methodology of Refactoring Process

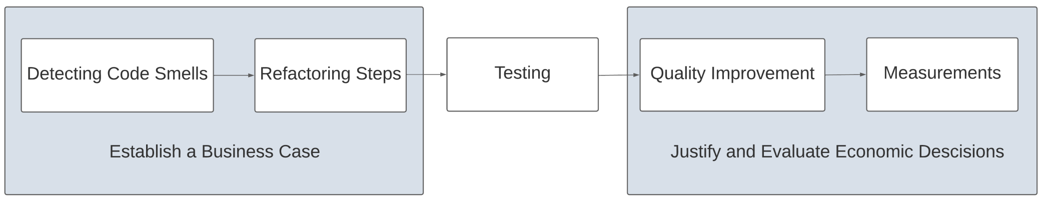

Figure 5.1: Extended model of Refactoring

[Transition needed that introduces chapter with its contents]

[Also the model should be briefly described with its broader implications]

5.1 Detecting Code Smells

The following part of this chapter moves on to describe in greater detail the detection of code smells within the software system. It provides a brief overview of the methods used and mentions the prevalence of the program *Sonargraph* as the software tool used in the detection activities. Moreover, this section provides necessary clarifications to understand why certain methods were chosen over others.

As previously stated, not all 24 code smells that are examined throughout Martin Fowler's book were used in the detection work. Instead, the 10 most frequently reported code smells have been chosen according to the work of Lacerda et al. (2020) Further, two additional smells *refused bequest* and *data clumps* were discarded. The decision was made due to their focus on inheritance and data structures respectively. More particular, these two programming abstractions are not utilized in the code, and are thus not of relevance in the detection. In order to revisit the code smell catalog with their corresponding explanation, refer to 2.2.1. Two primary reasons exist that have led to the decision of restricting the amount of smells for this section. First, it is not in the scope of the thesis to find each and every code smell in the codebase. The aim is rather to find the problems that are most apparent. Second, including more smells would be less of a problem, if automatic code detection tools could be used. Without being able to detect automatically, a limited amount of smell is more appropriate in order to not overspend time. A detection approach based on automatic tools is unfortunately not available for the python programming language, as will be alluded to in the section below.

5.1.1 Detection Methods

Many tools have been created to automatically or semi-automatically detect code smells (Menshawy et al., 2021). In the context of our thesis, the code smells are detected semi-automatically using a metric-based approach. This approach measures source code elements and takes decisions based on threshold values (Menshawy et al., 2021). It is fulfills two requirements necessary for the following work. On the one hand, it to a certain degree contains automation, increasing the efficiency and standardization in comparison to manual approaches. Further, it can be used for the python programming language, which is an essential prerequisite in the software system at hand. Menshawy additionally points out that a metrics approach does not provide metrics for every code smells. Nevertheless, for the eight smells the thesis is focussing on, as will be seen afterwards, an appropriate metric was found.

[Personal Note: This can also be argued by lacerda's work, which states for all of these metrics a metric-based approach exists]

Using a detection approach utilizing automated tools would have been more attractive, but unfortunately not possible. It is the most used approach (Menshawy et al., 2021), its availability is however highly dependent on the programming language the software system is written in (Menshawy et al., 2021). Further, looking at Menshawy's research on the most cited tools, it can be observed that a significant difference exists between java programming language, which was supported by 48% of the tools, whereas the python was supported by mere 4%. In addition, by researching possible automated tools appropriate for python, it was evident that at this moment of time, the offering is insufficient to accomplish a comprehensive detection strategy.

In the beginning, the author also considered following a manual approach. In contrast to automation, the manual detection relies on human perception of smells by applying predefined guidelines (Menshawy et al., 2021). It is characterized as highly time-consuming and prone to human error. Therefore, when comparing this approach to a metrics-based, it is less desirable and was subsequently discarded as a detection technique.

Another detection approach that has not yet been mentioned, is the detection by means of code visualization. Although it only is used to detect a subset of smells, visualization provided to be of use in two occurrences throughout the thesis. First, it had been used to get a broad overview of the code base, which helped the author with orientation. Second, it was used to aid in judgments regarding coupling and cohesion of the code base.

5.1.2 Sonargraph

As indicated in the beginning of this chapter, the detection of code smells relied heavily on a code analyzer tool called Sonargraph. Its usage demonstrated to be extremely useful in the detection of code smells, by its ability to compute and list metrics of the software system. The software included sufficient and diverse selection of metrics. These metrics provided insights by both evaluating the entire project and also its individual modules. For each metric, Sonargraph offered explanations, which allowed for an appropriate metric to be found regarding each relevant code smell.

Compared to other software solutions, Sonargraph was chosen for several reasons. Although the full suite of Sonargraph Applications is not free of charge, its graphical application *Sonargraph Explorer* was free to use. For the thesis, it provided both numerical and visual methods to make observations about the code base. All the tools needed to compute metrics were accessible. Another noteworthy benefit of using Sonargraph during the detection, was its ability to observe the entire project, and not just at individual files. This was essential for metrics taking account of dependencies between modules. In addition, it was beneficial not to rely on a multitude of software tools, by having just one software.

It is important to note that there was not an appropriate metric that could easily detect code duplication. However, in its paid version, accessible within a trial period, an automatic tool to detect code duplicates was provided. Previously, it was stated that it was difficult to find automatic tools to detect code smells in a software system written in python. Duplicate code particularly differs from the other smells, as it can be detected language agnostic. In other words, to detect duplication of code in programs, the software does not need to understand the programming language. This same mechanism could theoretically be applied to other types of text that are not code. Surprisingly, the restriction of a trial period did not prove to be a problem. It was possible to renew a trial period multiple times, although this was not indicated on its website. As no guarantee can be given on the amount of reactivation, one must still be aware of this restriction.

5.1.3 Metrics Overview

The above table presents for each code smell a corresponding metric. In addition, for some metrics, a threshold value is given. When the threshold value is surpassed, there is an indication that this code smell exists in the software system.

Individual metrics are categorized into three subgroups, named after the quality attribute they best represent. More importantly, however, this distinction is made, as each of the group differs in regard to how a metric results in a smell detected. For example, as mentioned before, the subgroup

Category	Code Smell	Metric	Threshold
Readability	Duplicated Code	Number of duplicates	Automated
Complexity	Long Method	Numbers of Statements	100 Statements
Complexity	Large Class	Lines of Code (LOC)	900 Lines
Complexity	Long Parameter List	Number of Paramers	8 Parameters
Complexity	Lazy Class	Lines of Code (LOC)	50 Lines
Cohesion/Coupling	Shotgun Surgery	Physical Coupling	Dependency Graph
Cohesion/Coupling	Divergent Change	Physical Cohesion	Dependency Graph
No detection	Feature Envy	Relational Cohesion (Not provided for Python)	

Figure 5.2: Metrics without prioritization

readability differs considerably due to the fact that code duplication is automatically detected. In this case, the application limits the detection to a minimum of 25 Lines. This means that only duplicated are shown if they surpass this amount.

Half of the code smells, in the subgroup called complexity, follow a typical metrics based approach. The threshold value indicate when a code smell is potentially present. Notably, all the metrics in this category involve counting of some sorts. In particular, the smells are detected by counting the number of statements, lines of codes, and numbers of parameters of methods. Prioritization of given smells can be done by comparing the extent of how much a given threshold value is surpassed. Similarly, code smells can be discarded if the associated metric does not surpass the threshold value.

Lastly, Shotgun Surgery and Divergent change respectively are bound to the coupling and cohesion of the software system. The metric used measures the dependencies "to" and "from" other components, in either the same module (cohesion) or in other modules (coupling). The appropriate threshold value highly depends on the size of the software system. By adding code to the code base, we naturally increase the amount of expected dependencies. Therefore, the author decided to use incorporate a dependency graph in addition to the metrics, in order to evaluate the prominence of these two code smells.

5.2 Testing

In the chapter that follows, the relevance of testing as a fundamental part of the refactoring process will be discussed. Subsequently, the challenges of testing a cyber physical system will be described. Based on these findings, the particular methods used for testing of the software system will be outlined at the end of this chapter.

5.2.1 Importance of Testing in a Refactoring Process

Turning now to the importance of testing within the context of the refactoring process. In order to understand the significance, it is appropriate to return to Martin Fowler's definition of refactoring (Fowler, 2018) as first discussed in section 2.1. According to his definition, refactoring must not alter the external behavior of code. It was pointed out in the theoretical part of this thesis that by not changing the features of the program, programmers can alleviate the risks associated with changing the codebase, with the predominant focus of not introducing new bugs. As a result, to demonstrate that the behavior of the code base was not altered during refactoring, testing is necessary. Moreover, to confirm of an unaltered state of code behavior is vital to verify the improvement of code quality. Therefore, without testing we would get a false sense of an improvement of code quality, as potentially negative alterations are not taken into consideration. Hence, only with testing we can ensure that after refactoring the software system is in a better state than before.

5.2.2 Testing Cyber Physical Systems

Knowing the benefits of testing, it is also important to discuss the challenges, granted that our software system is a cyber-physical system (CPS). These challenges arise due to the unique characteristics of a CPS and need to be taken into account when determining the methodology of testing these systems.

Robots and other cyber-physical systems react to information from the physical worlds and must operate safely even in the presence of uncertainties Geiß, n.d. In such a dynamic real-world environment, Kapur (2020) describes testing a CPS, as time-consuming and a particularly complicated task for developers. He points out that this is especially the case as it is difficult to ensure a robot will behave as expected. Similarly, Rajkumar et al. (2010) point out the challenge of verifying and validating software systems due to their heterogeneous nature. Furthermore, it is hard to define the boundaries and physical limitations of the testing landscape made for a CPS (Abbaspour Asadollah et al., 2015).

Another limitation in CPS testing is the introductions of automated semi-automated methods. Here, the efficacy of testing is limited by the element of its hardware components. Moreover, running tests is dependent on the hardware's capabilities, and that it is operating as expected. For instance, CPS with physical motions are subject to considerable time running tests and potentially require human supervision and manual intervention. Some of these challenges can be mitigated by using a physics-based simulation that mimics the real-world environment. (Kapur, 2020). Such solutions do exists in practice, but are unfortunately unavailable to our software system.

This challenging physical environment of CPS ultimately results in programmers having to satisfy multiple levels when testing the software. The levels of CPS software testing contains verifying software, hardware, network, and the integration of all these components to work as a single system abbaspourasadollah2015. In general, there is a high interest to use automated testing, as manual testing is more likely to produce errors turlea2019. Despite automated tests having advantages such as more testing in less time and improvements in quality, it needs to be considered that no software tests are available for the software system at hand. In particular, the major disadvantages of automated tests are the costs associated with developing test automation, especially in dynamic customized environments (Taipale et al., 2011). Consequently, developing a testing suite having to develop a test suite from scratch forces us to choose a manual approach. It is important to note, that in future work, it would be appropriate to consider developing software tests. It is however not in the scope of this thesis to also take on this undertaking.

One advantage of the physical nature of CPS is that we are able to make visual observation. This feature can be viewed as a possible advantage in the manual testing of a particular CPS. Whether the observation of a CPS can be used in testing, is arguably strongly dependent on the type. In our case, there are no restrictions in observing the system visually. The benefits of such an approach become especially apparent when taking into account that our software project is focussing on managing entire processes. In more concrete terms, we are able to perceive visual changes in the behavior of our CPS by running a predefined process. All in all, the lack of software tests and the ability to visually observe behavioral changes encouraged the decision to include manual testing, as a preferred method. Even though this approach can not predict the equivalence of the entire software system, this method of testing can to a certain degree ensure no alterations within a particular processes. Moreover, this approach presumably inferior to comprehensive unit and integration testing. Nevertheless, it is an approach that suits the preconditions within our context, by determining changes within the scope of processes.

5.2.3 Method selection

As just indicated, testing for this thesis will be done manually by means of observing an entire process of the CPS. To reduce the risks of uncertainty, methods are selected based on their ability to test objectively and in a routinized manner. To achieve this goal, two methods have been chosen, where one enables testing automation, while the other improves objectivity. In particular, the software application Camunda Modeler is used to automate the business processes and the Gherkin Language is used to objectively formulate requirements. In combination, these tools deliver a procedure to reduce ambiguity, which is a fundamental prerequisite when trying to capture any cue in observable changes. As a result, this approach reduces some shortcoming of manual testing, while also decreasing human error.

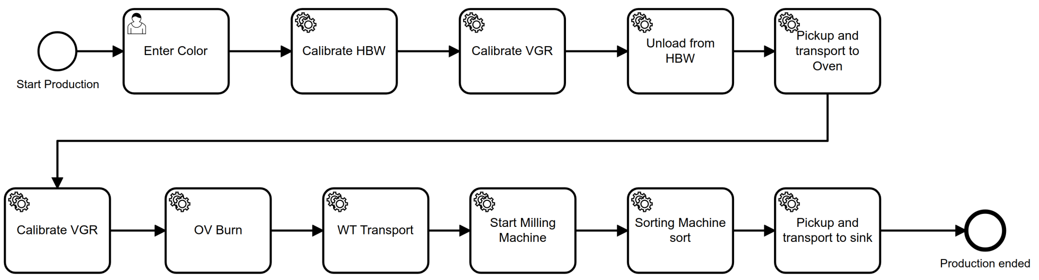

Figure 5.3: Camunda Production Process

Even though there are no software tests available, multiple business processes, using the BPMN representation, have already been implemented for the present software system. These representations are stored in *.bmpn* files are can be executed by the Camunda Modeler. This then allows the automation of the said processes. For the testing, one particular process has been chosen to reflect the software system. It is a process that simulates production and its file is named *production_process.bpmn*. Using this file, we are able to simulate an entire production process that includes all components of the factory and hence can cover a major part of the software system. Using the contents of this file, the Camunda Modeler is able to sequentially do predetermined HTTP requests, which otherwise would have been done manually. Performing these requests manually would not only be very tedious, but also takes away standardization with a risk of making mistakes. In addition, the BPMN notation used by Camunda Modeler allows us to visually inspect at which part of the process one currently is when running it. This is an exceptionally useful feature, when trying to find out where a current issue might be located, if the visual tests fail.

Gherkin on the other hand is not a software application. Instead, it is a language that is oftentimes used in conjunction with software. Hereby, one can formulate behavior in the form of specifications written in plain text and understandable by humans ("Gherkin Reference," n.d.). It uses a set of special keywords that enables formulating these specifications in a structured way. We can then validate whether the software does what the specifications say. This validation is primarily done with corresponding software tools.

However, in our case this will be done manually. Consequently, by using the gherkin language, we are able to objectively check whether the current codebase still fulfills the specification in a form that is both human-readable and can also be routinely repeated in a standardized way. Furthermore, the specific syntax of the Gherkin language, enables us to write objective statements.

Gherkin Framework applied to Production Process

```
Feature: Execute Production Process
  As a user, I want to run the production process in Camunda,
  so that I can test my refactor

  # Fischertechnik Factory
  Background:
      Given: All six hardware components are turned on
      Then: Run FT-Gui program

  # App
  Background:
      Given: init.factory.py is running
      Then: Run app.py

  # Camunda
  Background:
     Given: Laptop is connected to network
     Then: Run Starting scripts
     Then: Open Camunda Modeler

  Scenario: Running production process
     Background:
          Given: Storage process not yet executed
          Then: Run storage process by starting process instance

     When: Starting process instance
     Then: Enter Color as input
     Then: Process will start
```

As seen in the specifications above, one can observe multiple prerequisites that are necessary due to having hardware. This amount of detail is critical when wanting to reproduce this process at a later time. Even though, some specifications might seem self-evident, only then we can diminish the ambiguity associated with testing CPS. Each step must start with one of the Gherkin predefined keywords *Given, When, Then, And, or But* providing the necessary structure. Such structure enables to formulate requirements that can translate to preconditions, user actions, and outcomes of the procedure.

5.3 Measuring Improvement

Having incorporated testing in our refactoring model, we are now in a position to consider methods for measuring the improvement of code quality after refactoring. As mentioned in the theoretical framework, to measure the internal quality, programmers can consider quality attributes as measurement criteria.

5.3.1 Importance

Measuring improvement in code quality, allows us to evaluate how successful refactoring was. In particular, we can measure the extent of improvement, by comparing the refactored software system with its state prior to the refactoring. Having measurements aids us in making decisions in the future. If the metrics are sufficiently accurate, one could argue when to initiate and when to stop refactoring, by using them as indicators. The comparison of the initial and final state of a refactoring can be further extended by periodically measuring code quality. We then are able to individually attribute the improvement by getting rid of specific smells.

In order to make quantifiable measurements, the thesis employs the maintainability index, which measures how maintainable the source code is. This measure has been chosen primarily, as it offers single-valued quantification of code quality, while considering multiple aspects of the code. A more detailed account of the index will be given in the following section.

5.3.2 Maintainability Index

or For the computation of the maintainability index, the python package Radon was used. Apart from the maintainability index, this software tool is able to compute raw metrics (SLOC, comment lines, blank lines), Cyclomatic complexity and Halstead volume. These listed metrics are all interconnected, as the maintainability index in fact include all of these in its computation. The documentation of the software Radon provides brief explanations of the index and its implementation. The following section makes extensive use of the documentation in order to provide an overview ("Radon Documentation," n.d.). To begin, one should look at the components of the formula of this index to better understand what it wants to achieve.

Original Formula:

$$MI = 171 - 5.2 \ln V - 0.23G - 16.2 \ln L \tag{5.1}$$

- V = Halstead Volume
- G = Cyclomatic Complexity
- L = Source Lines of Code (SLOC)

Derivative (used by Radon):

$$MI = \max\left[0, 100\frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L + 50 \sin(\sqrt{2.4C}))}{171}\right] \tag{5.2}$$

Both formulas calculate the maintainability as a factored equation consisting of three primary components. Halstead volume is measured by multiplying halstead length (number of operators and operands) with vocabulary (number of unique operators + unique operands), Cyclomatic Complexity calculates the number of linearly independent paths in the source code, indicating the complexity of a program. SLOC counts the number of lines in the computer program.

The original equation is included as it makes it easier to understand how the individual components contribute to the end result. The second formula, which is used by Radon, is a derivative of the original formula. It is preffered as it is able to measures the maintainability between a scale from 0 to 100. As a consequence, the computation is easier to understand, as values closer to 100 indicate better maintainability. The second difference is that this formula also takes into account the percentage of commented lines indicated by the variable C. It diminishes the significance SLOC, when the source code contains a lot of commented lines. Altough not specifically mentioned in the documentation, this is presumably the reason why Radon included this additional variable.

5.3.3 Methods used for taking measurements

The method of taking measurements is also a design decision that was consciously carried out. At first, periodically taking the measurements manually was considered as a suitable method. Still after some consideration it was evident that an automated approach is more appropriate. One of the reasons was that manually running the software tool after each change quickly becomes tedious. Considerable amount of time would be spent for something that is easily automated. There would also be no clear indication on the right amount of measurments to take in a routinized way. By manually making too many measurements not only a lot of time is required, but also it would be difficult to differentiate between the individual measurements. Conversly, by having too few measurements, there would not be sufficient information available to make meaningful conclusions.

Regarding the timing of the measurements, it was immediately clear measuring after each commit in the version control program would be suitable. That way each measurement could directly be attributed to each commit, which includes information on code changes, such as a commit message and the possiblity to check the difference to the previous state of the prgam. Fortunately enough there exists an intuitive way to run scripts in conjunction with git activities. These types of scripts are called *git hooks* or simply *hooks*. As a result a hook was written using bash, implemented in a way that it is executed prior to each git commit. Consequently, this approach solved both concerns of time inefficiency and lack of attribution. It is important to note however, that this was only possible due to Radon being a command line utility. If graphical software would have been used, only a manual approach would have been feasible, if not directly implemented by the software itself.

The bash script performs two distinct functions: computing and saving. Initially it executes radon to compute the maintainability index and including its subcomponents. Here, the result is temporarily saved in a variable. Next, these computations are then saved by writing them to a file. This file is saved locally on the computer, named after the date and time of execution. This allows the previously indicated attribution between measurement and commit.

What is special about this methodology is that it enables continuous tracking in three dimensions. First, we are able to attribute individual refactoring steps in the form of commits. Second, we are able to more broadly connect measurements to the refactoring of individual code smells, preferably by indicating smells in commit messages. Lastly, by comparing current measurements to the initial state of the project, we can evaluate the overall progress since starting the refactor.

5.4 Next steps

[Here it could be useful to recap this chapter with a short conclusion]

6 Main Findings and Results

6.1 Code Smells

Code Smell	Metric	Threshold	Value	Source	Notes
Duplicated Code	Number of duplicates	Automated	24 Blocks	Sonargraph	Multiple Occurences
Long Method	Numbers of Statements	100 Statements	1 Method	Sonargraph	114 Statements
Large Class	Lines of Code (LOC)	900 Lines	1 Class	Sonargraph	app.py
Long Parameter List	Number of Paramers	8 Parameters	0 Methods	Manual	<5 Params
Lazy Class	Lines of Code (LOC)	50 Lines	6 Classes	Manual	Everything in /txts/ directory
Shotgun Surgery	Physical Coupling	Dependency Graph	<7	Dependencies 'to' and 'from' other components in other modules.	
Divergent Change	Physical Cohesion	Dependency Graph	<7	Dependencies 'to' and 'from' other components in the same module.	
Feature Envy	Relational Cohesion (Not provided for Python)				

Figure 6.1: Prioritization of Smells according metrics

6.2 Refactoring

6.3 Measurements

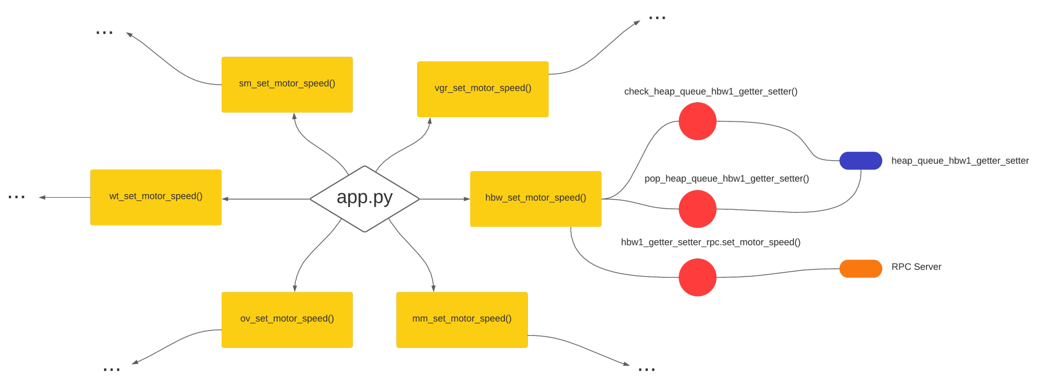
app.py
sm_set_motor_speed()
vgr_set_motor_speed()
wt_set_motor_speed()
hbw_set_motor_speed()
ov_set_motor_speed()
mm_set_motor_speed()
check_heap_queue_hbw1_getter_setter()
pop_heap_queue_hbw1_getter_setter()
heap_queue_hbw1_getter_setter
hbw1_getter_setter_rpc.set_motor_speed()
RPC Server
...

Figure 6.2: Code duplication illustration

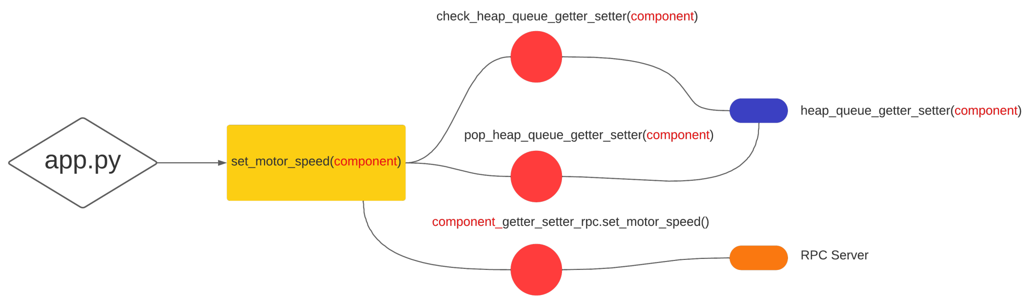
app.py
set_motor_speed(component)
check_heap_queue_getter_setter(component)
pop_heap_queue_getter_setter(component)
heap_queue_getter_setter(component)
component_getter_setter_rpc.set_motor_speed()
RPC Server

Figure 6.3: Removing Code duplication

7 Discussion

8 Conclusion

Declaration of Authorship

"I hereby declare

- that I have written this thesis without any help from others and without the use of documents and aids other than those stated above;
- that I have mentioned all the sources used and that I have cited them correctly according to established academic citation rules;
- that I have acquired any immaterial rights to materials I may have used such as images or graphs, or that I have produced such materials myself;
- that the topic or parts of it are not already the object of any work or examination of another course unless this has been explicitly agreed on with the faculty member in advance and is referred to in the thesis;
- that I will not pass on copies of this work to third parties or publish them without the University's written consent if a direct connection can be established with the University of St.Gallen or its faculty members;
- that I am aware that my work can be electronically checked for plagiarism and that I hereby grant the University of St.Gallen copyright in accordance with the Examination Regulations in so far as this is required for administrative action;
- that I am aware that the University will prosecute any infringement of this declaration of authorship and, in particular, the employment of a ghostwriter, and that any such infringement may result in disciplinary and criminal consequences which may result in my expulsion from the University or my being stripped of my degree."

Date	Signature

By submitting this academic term paper, I confirm through my conclusive action that I am submitting the Declaration of Authorship, that I have read and understood it, and that it is true.

Appendix

Bibliography

Abbaspour Asadollah, S., Inam, R., & Hansson, H. (2015). A Survey on Testing for Cyber Physical System [cit. on p. 205]. In K. El-Fakih, G. Barlas, & N. Yevtushenko (Eds.), *Testing Software and Systems* (pp. 194–207). Springer International Publishing. https://doi.org/10.1007/978-3-319-25945-1_12

Alkhazi, B., Abid, C., Kessentini, M., & Wimmer, M. (2020). On the value of quality attributes for refactoring ATL model transformations: A multi-objective approach. *Information and Software Technology*, *120*, 106243. https://doi.org/10.1016/j.infsof.2019.106243

Bass, L. (1998). *Software architecture in practice*. Addison-Wesley
Open Library ID: OL668208M.

Fowler, M. (n.d.). *Technical Debt*. martinfowler.com. Retrieved May 9, 2022, from https://martinfowler.com/bliki/TechnicalDebt.html

Fowler, M. (2018, November 30). *Refactoring: Improving the Design of Existing Code* (2nd edition). Addison-Wesley Professional.

Geiß, V. M. (n.d.). *Robotics and Cyber-Physical Systems | Computer Science Research at Max Planck Institutes*. Retrieved May 14, 2022, from https://www.cis.mpg.de/robotics/

Gherkin Reference. (n.d.). Cucumber Documentation. Retrieved May 17, 2022, from https://cucumber.io/docs/gherkin/reference/

Kapur, P. (2020, July 1). *Introduction to Automatic Testing of Robotics Applications*. Amazon Web Services. Retrieved May 14, 2022, from https://aws.amazon.com/blogs/robotics/automatic-testing-robotics/

Kim, M., Zimmermann, T., & Nagappan, N. (2012). A field study of refactoring challenges and benefits. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 1–11. https://doi.org/10.1145/2393596.2393655

Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice [cit. on p. 18]. *IEEE Software*, *29*(6), 18–21. https://doi.org/10.1109/MS.2012.167

Lacerda, G., Petrillo, F., Pimenta, M., & Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, *167*, 110610. https://doi.org/10.1016/j.jss.2020.110610

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2004.1265817

Mens, T., Demeyer, S., Du Bois, B., Stenten, H., & Van Gorp, P. (2003). Refactoring: Current Research and Future Trends. *Electronic Notes in Theoretical Computer Science*, *82*(3), 483–499. https://doi.org/10.1016/S1571-0661(05)82624-6

Menshawy, R., Hassan Yousef, A., & Salem, A. (2021, May 26). *Code Smells and Detection Techniques: A Survey* [cit. on p. 1]. https://doi.org/10.1109/MIUCC52538.2021.9447669

Radon documentation. (n.d.). Retrieved May 17, 2022, from https://radon.readthedocs.io/en/latest/

Rajkumar, R., Lee, I., Sha, L., & Stankovic, J. (2010). Cyber-physical systems: The next computing revolution [cit. on p. 763]. *Design Automation Conference*, 731–736. https://doi.org/10.1145/1837274.1837461

Sonnleithner, L., Oberlehner, M., Kutsia, E., Zoitl, A., & Bacsi, S. (2021). Do you smell it too? Towards Bad Smells in IEC 61499 Applications. *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 1–4. https://doi.org/10.1109/ETFA45728.2021.9613379

Taipale, O., Kasurinen, J., Karhu, K., & Smolander, K. (2011). Trade-off between automated and manual software testing [cit. on p. 114, 122]. *International Journal of System Assurance Engineering and Management*, 2(2), 114–125. https://doi.org/10.1007/s13198-011-0065-6