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

Refactoring is known to improve the quality of a software system by restructuring computer code. Here it is well established that refactoring is in close relationship with the detection of code smells. This thesis aims to extend the current understanding of refactoring by examining what further connections can be drawn in areas of software engineering and business. Specifically, it develops a process that positions refactoring in a larger context by depicting its key drivers.

To test whether refactoring is a conjoint activity, the investigation follows a case-study approach, utilizing a software system in the domain of industry 4.0. In doing so, the thesis presents a comprehensive refactoring methodology suitable for this particular case. The findings provide insights to guide practitioners in their refactoring endeavors and offer practical advice for refactoring software systems specific to industry 4.0.

The results show refactoring having multiple connections to other domains by presenting an interconnected refactoring process. Findings suggest the detection of code smells, testing environment and the measurements of quality improvement as substantial contributors to refactoring success. In addition, the thesis suggests businesses to establish a business case before refactoring and recommends communicating their decisions using the concept of technical debt.

Contents

List of Tables

List of Figures

1. Introduction

Kent Beck stated "I am not a great programmer; I am just a good programmer with great habits" (Fowler et al., 2012). One of these great habits is the ability to write code that is not only written for machines, but can easily be read, modified, and extended by humans. Corresponding programming tasks carried out by humans are prohibited by internal problems existing in the code base, commonly referred to as code smells. Here, refactoring is frequently prescribed to solve these issues.

Various work has been conducted on refactoring and code smells (Fowler, 2018; Lacerda et al., 2020). Several studies have been published on the detection of code smells (Chen et al., 2016; Menshawy et al., 2021) and techniques suitable for refactoring code (Fowler, 2018; Mens and Tourwé, 2004). These studies have consistently found a close relationship between code smells and refactoring. Despite this, very little has been done to explore how refactoring is related to other fields in software engineering and beyond.

The primary aim of this thesis is to investigate, whether further areas of software engineering can be found that closely relate to refactoring. In addition, the thesis does not limit itself to software engineering practices and also sets out to examine refactoring from a business perspective. The investigation follows a case-study approach, utilizing a software system in the domain of industry 4.0. Combining established literature and an in-depth analysis of

the software system, the thesis aspires to develop an overarching framework, which will be referred to as the *refactoring process*. The purpose of this approach is to argue against the notion of refactoring being a distinct activity. Hence, this investigation strives to explore and extend the current understanding of refactoring. The subsequent findings aim to provide a basis for other refactoring endeavors and particularly offer advice related to software systems in industry 4.0.

The thesis is composed of six themed chapters. The first chapter provides an overview of the case study. The aim of the chapter is to introduce the context of industry 4.0 and present the learning factory and its corresponding software system. This chapter previews some of the connections that are to be drawn from refactoring. The second chapter gives a brief overview of the theoretical framework of refactoring. The main topics covered in this chapter are the benefits, criteria, and challenges associated with refactoring. In addition, the most prominent code smells are introduced and the relationship to refactoring is discussed. Following, the third chapter will examine refactoring by introducing the concept of technical debt. Here, a close connection between refactoring and the business sphere will be demonstrated. The fourth chapter identifies new areas that are tightly related to refactoring, extending the scope of the term refactoring. Additionally, the software system is taken as a reference to develop a suitable refactoring methodology. Chapter five offers concrete results using the developed methodology from the previous chapter. At this stage, the prevalence of code smells is examined and corresponding refactoring measures discussed. At last, the thesis closes with a conclusion, which summarizes the main findings and discusses their relevance.

2. Contextualization of the Learning Factory

2.1. Case Study in Industry 4.0

The Fourth Industrial Revolution (Industry 4.0) is considered a new industrial stage integrating the manufacturing environment with the cyber world to form cyber-physical systems (CPS) (Jazdi, 2014; Wang et al., 2015). This transition creates opportunities towards more autonomous and intelligent production lines (Malburg et al., 2020). To conduct research in this field without depending on expensive production lines, learning factories are used to physically model realistic shop floors on a smaller scale (Malburg et al., 2020). The thesis extensively uses the Fischertechnik learning factory when conducting research on refactoring. This allows to run experiments at low costs and transfer results to real smart factories (Malburg et al., 2020), a key construct of industry 4.0. (Osterrieder et al., 2020).

Throughout this thesis, different terms are used to refer to the case study. When referring to the hardware environment of the learning factory, the term *CPS* is used. For discussing the software elements of the learning factory, the term *software system* is used. Further, when collectively concentrating on hardware and software, the term *learning factory* or simply *factory* is used.

2.2. Industry demands of Refactoring

Industry 4.0 is closely linked with the business space. Industry-wide developments promise more efficient production processes, optimized supply chains, and cost reductions (Malburg et al., 2020). By the increasing relevance of software in manufacturing companies, financial implications on business caused by refactoring should be explored. Section 4.1 explains the concept of technical debt, which provides a foundation to argue for refactoring from a business perspective. Further, once a business case for refactoring is established, industries want to efficiently allocate their resources. Section 4.2 elaborates on the topic of justifying company-wide refactoring decisions and discusses the communication of these decisions.

Moreover, by examining refactoring theory, businesses are able to learn the benefits of refactoring (section 3.2) and understand what criteria influence refactoring (section 3.3). In addition, to fully profit from refactoring, organizations should have an understanding of associated challenges (section 3.4) and challenges specific to CPS (section 5.2.2).

2.3. Fischertechnik Learning Factory

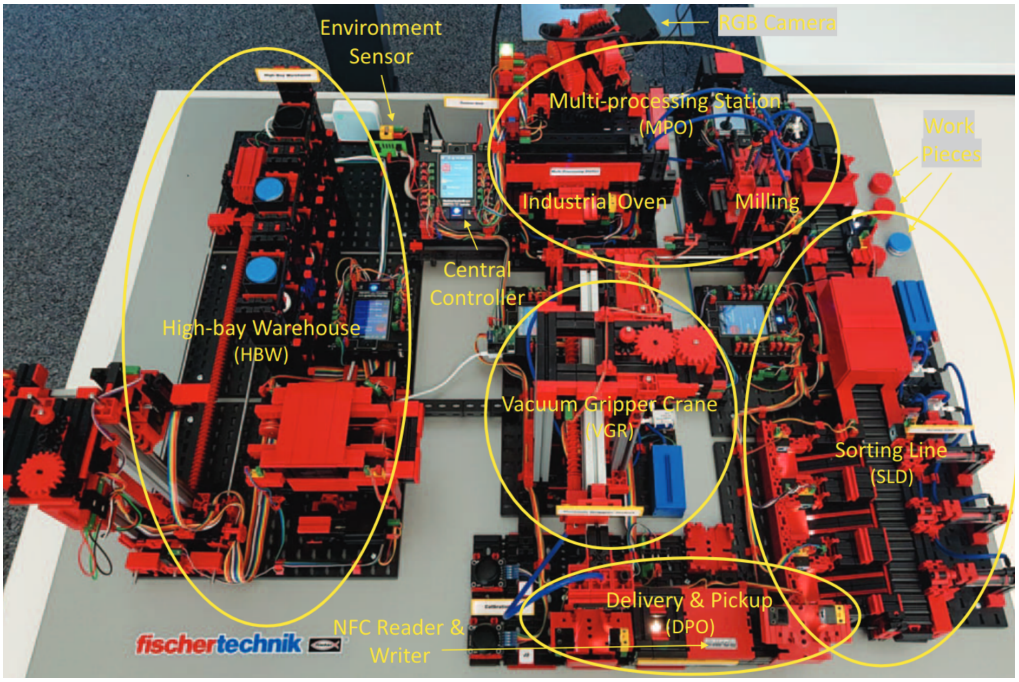

Figure 2.1.: Fischertechnik Learning Factory (Seiger et al., 2020)

Figure 2.3 presents an overview of the learning factory utilized for the case study in the thesis. The factory consists of five workstations: a sorting line (SLD) with color detection, a multi-processing station (MPO), a milling machine (MM), a high-bay warehouse (HBW), and a vacuum gripper robot (VGR) (Seiger et al., 2020). Including sensors, actuators and the Fischertechnik TXT controllers, the learning factory features a complete production line (Malburg et al., 2020).

To enable programming of the learning factory on the abstraction level of business processes, Malburg et al. (2020) found that an additional software stack is required on top of the existing IoT components. The software system used for the case study, fully implements this abstraction, by using web services.

This enables the learning factory to be controlled by a process automation platform Camunda using Business Process Model and Notation (BMPN).

However, the contribution of computer code that was needed to achieve this task has not been refactored. In addition, it is envisioned for the software system to migrate towards microservices. In particular, it would be desirable for each workstation to be its own microservice. For instance, fault tolerance could be introduced, by having the ability to restart each station independently. These conditions inspire an exploration into refactoring this software system. Given the present circumstances of the learning factory, it is appropriate to analyze both the methods (chapter 5) and the urgency (chapter 6) of refactoring this system.

3. Theoretical Framework for Code Refactoring

3.1. Background

Refactoring is a well established concept in software development. Especially in complex projects, tasks like cleaning and restructuring code are regularly occurring and oftentimes necessary. When referring to these activities by name, many employees use the term *refactoring*. The exact definition of the term 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 the following parts of the thesis.

Martin Fowler (2018) 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 and ideas, many of them discussed throughout this thesis. 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.

3.2. 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) discuss the problem of not sensing an immediate benefit when refactoring. Furthermore, the value of improving the internal structure of the code is hard to demonstrate to a manager, who is not an expert and even harder to present to a client, who is paying for the work.

Having said that, when refactoring is ignored, more time is not necessarily accessible to programmers. Mens et al. (2003) argue that through continuous modifications and adaptations to new requirements, the code becomes increasingly complex and drifts away from the original design. Hence, by not improving the quality of the software, resources will still need to be spent on software maintenance. Bearing in mind Fowler's Definition of refactoring, we therefore aim to mitigate time spent on tedious maintenance, by instead improving attributes of the code beforehand.

In particular, refactoring helps to improve the internal quality attributes of the software system (Mens and Tourwé, 2004). For instance, some refactoring activities remove code redundancy, some raise the level of abstraction, some enhance the reusability. Bass (1998) distinguishes between two categories of quality attributes. The first category observes software attributes at runtime. This includes the performance and the security of the software system. The other category does not consider system runtime, but focuses on the quality of the source code. During refactoring, we are only concerned with the latter. This means that we do not consider software quality with respect to its performance and security.

During a study on the value of quality attributes on refactoring, Alkhazi et al. (2020) identified six of such quality attributes: Reusability, Flexibility, Understandability, Functionality, Extendibility, and Effectiveness. As a general principle, refactoring adds value if at least one quality attribute has significantly improved, given that the external behavior was not altered. 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, to then test whether features remained the same after refactoring (Fowler, 2018). As a result, software tests are a major component when deciding the success of a refactor (for more information refer to 5.2).

3.3. Criteria: When should we Refactor?

From a business standpoint, there needs to be little explanation, when debating time spent on new features and fixing unresolved bugs. As discussed earlier, it is significantly harder to persuade refactoring, as there are fewer immediate benefits. For that reason, one needs to verify the necessity of refactoring and compare it to the need of other pending tasks. For instance, Fowler (2018) points out that if the code works and does not ever need to change, it is fine to leave it alone. In contrast, 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.

It is crucial to note that refactoring does not have to be a large and complex undertaking. Even if there is a demand for it, refactoring activities themselves are minor. Fowler (2018) describes refactoring practices as the application of small behavior-preserving steps, allowing to produce a big change by stringing together a sequence of these behavior-preserving steps. For this reason, refactoring as a principle, can always play a part in software development.

Therefore, for projects that appear to be qualitatively sufficient, active monitoring of the code quality and continuous improvements through refactoring is still advisable.

This approach leads to the presumption that these activities can be done in parallel. However, adding features and refactoring at the same time is not recommended. The interplay between them can be described well by making use of an analogy developed by Martin Fowler (2018). He sees software development as wearing two hats, and proposes that time should be distinctively divided between adding functionality and refactoring. This means that during refactoring 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). Conversely, once the code is better structured, time can be efficiently spent on adding new capabilities.

This analogy illustrates nicely that refactoring should be an integral part of software development. Once this relationship between refactoring 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). As a consequence, doing continuous refactoring might not suffice for larger projects.

However, there needs to be a note of caution, in assuming that for software systems with good internal structure, small refactors suffice. Likewise, in software systems that lack quality, one might think that larger refactoring is required. Although intuitively both assumptions make sense, there are exceptions to both

cases that are important to keep in mind.

The first assumption, associated with well-structured software, can be disputed if one of the quality attributes gains significantly in importance. Such a situation alone may justify large efforts of refactoring, necessary to reach the desired state. The second assumption, associated with software systems that lack quality, does not hold for modules of the codebase that are not a crucial part of the software system. Here, considering the associated costs, it is hard to make a case for planned refactoring and minor restructuring could suffice.

3.4. Challenges: What to watch out?

Having a comprehensive understanding of the benefits, scope, and timing of refactoring, it becomes evident that many of the difficulties are related to planning refactoring rather than its execution. The complexity of planning become more apparent, when working within teams. Going back to the analogy of the two hats, we assumed that the software development process is conducted by a single person. In practice, however, software development in organizations is typically done within a team. This makes refactoring considerably more difficult, knowing that refactoring and adding new features should be a distinct activity. Thus, when a programmer decides to refactor parts of the software system, it could inhibit others from working on the software at the same time. As a result, it is a challenge to clearly separate which part of the code base is being refactored, and which is added functionality. It is important to be aware of this challenge, because otherwise the risk of bugs being introduced becomes much higher.

Comprehensively planning refactoring may sound intimidating and time-consuming at first. Fowler (2018) argues against this belief, by suggesting

that the whole purpose of refactoring is to speed things up. Positioning 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. In practice, "too little refactoring is far more prevalent than too much" (Fowler, 2018). The suggestion that people should attempt to refactor more often, is, however, much easier said than done. Besides the mentioned difficulties, refactoring can add considerable value, with the precondition that it is properly carried out. Carrying out refactoring, however, is not always an easy task, requires education, and most definitely experience.

3.5. Formalization of Code Smells

Programmers need to be aware of indications demand refactoring. Fowler (2018) 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. There oftentimes is no a clear-cut moment that requires refactoring. Instead, there commonly exist indications of issues that can be solved by refactoring. In practice, these indications are known as *code smells* or *design smells* (Lacerda et al., 2020). These two terms are distinguished by existing either 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 software system, we will from now on refer to the indications as code smells.

The term *smell* is used in reference to an internal software problem (Lacerda et al., 2020), which can negatively impact software quality (Sonnleithner et al., 2021). 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 "impacting software maintenance and evolution" Lacerda et al., 2020.

Figure 3.1.: Refactoring process (Simplified Model)

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

To better comprehend the concept of code smells, it is best to list some of them. To stay within the scope of this thesis, we will limit the examples to the most prominent ones. Based on the survey conducted by Lacerda et al. (2020), the following summarizes the ten most frequently reported code smells.

Summary of code smells identified by Kent Beck and Martin 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	A method that is very likely to have too many responsibilities, hurting one of the principles of good object oriented 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.
Refused Bequest	Indicates that a subclass does not use inherited data or behaviors.
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 modification requires changing several classes.
Lazy Class	Classes that do not have sufficient responsibilities and therefore should not exist.

Table 3.1.: Code Smells Overview

4. Technical Debt

During the lifetime of a project, companies have to inevitably deal with code smells appearing in the software systems. Consequently, associated financial costs can only be mitigated by getting rid of them through refactoring. A typical example of a common financial cost is the additional work required, when programming with deficient code. Naturally, employees are spending less time when their code is easier to read, understand, and modify. Examining the discussion of the relationship between code smells and financial costs paves the way to consider refactoring decisions from a business standpoint.

4.1. Establish a Business Case

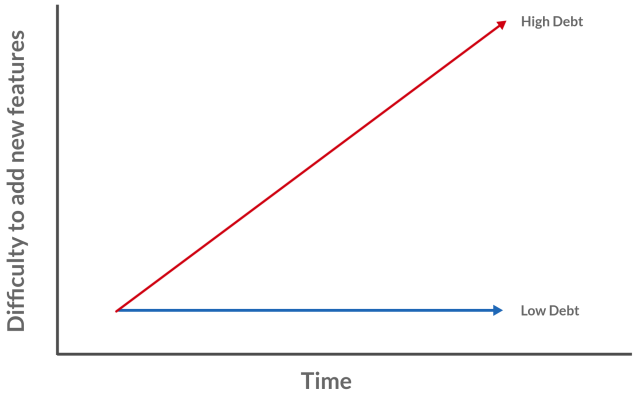

Figure 4.1.: Technical Debt Simplified Depiction

Technical debt is a helpful metaphor that allows assessing code smells in terms of financial debt (Fowler, 2019) in order to establish a firm business case. In concrete terms, the concept of 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 accumulated debt (Fowler, 2019). Fowler presents the example of a confusing module structure in a code base. He illustrates that a programmer needs four days to add new features with a clear structure and six days with code smells. Then, the two additional days spent 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.

One approach to think about solving technical debt is to argue for investing in experienced programmers to avoid debt entirely. Based on this assumption, it would be reasonable to do so, if the additional money spent on the salaries is offset by the cost endured from debt. In practice, it however is not that simple. Technical debt will always accumulate, even with the most experienced of programmers. For instance, technical debt could be created solely because one quality attribute gained in importance. It could also be due to the fact that a code base exists for a very long time. In both cases, refactoring is a crucial mechanism in order to get rid of technical debt.

Consequently, it becomes apparent, that businesses have financial incentives to make the right decisions in order to efficiently allocate their resources. 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 base instead of refactoring it. Difficult decisions require competent employees that are able to decide in ways to deal with the debt. These employees are not limited to one type of job role. Relevant expertise in both the management and software engineering

department can lead to substantial financial opportunities.

4.2. Justify Economic Decisions

It was mentioned that refactoring must not be restricted to programmers and can also be relevant to managerial roles such as team leaders, or product managers. Having knowledge of abstract concepts, like technical debt, allows employees to participate in decisions regarding refactoring and communicate them, without deep knowledge of software engineering. Instead of relying on software engineering best practices, employees are able to offer an evaluation that can be defended from a business standpoint. Moreover, some might even argue that decisions regarding refactoring should be purely economic. Fowler (2018) claims that it is beneficial to have this attitude and argues that economic benefits should always be the driving factor of refactoring.

Technical debt can be a useful device to communicate decisions relating to refactoring. It is a simple model to form significant and potentially complex economic decisions. The device can be used in two directions, top-down or bottom-up. In top-down communication, using the term technical debt, provides directions to team members. As an example, management decides to migrate their parts of their systems to the cloud, but team leaders realize that the code contains a lot of code smells, making it hard to undergo such a change. Communicating this issue using the concept of technical debt prevent refactoring parts of code that do not directly contribute to the objective of migration.

Likewise, technical debt as a metaphor can also support bottom-up communication. This metaphor can be used to provide adequate arguments for refactoring, when clients or upper management are in doubt of such a decision. For instance,

it could serve as a device to reason about the difficulties of adding new features or the issues of making major changes to the application. Overall, independent of the direction of communication, knowing technical debt allows companies to have less friction between software engineers and its other departments.

5. Methodology of Refactoring Process

In addition to the theoretic contributions made in the previous chapter, the primary aim of the following work is to provide a more comprehensive view on refactoring with a direct application to our case. In this chapter it will be argued that refactoring should be placed in a much broader context, as it directly relates to multiple software domains, such as code smell detection, testing, and measuring quality improvements. Including these domains, Figure 5.1 portrays a resulting *refactoring process*. It depicts refactoring as a sequence of connected stages, and thereby furthers our current understanding. In addition, both business related sections of chapter 4 are incorporated in this process, indicating the stages they influence.

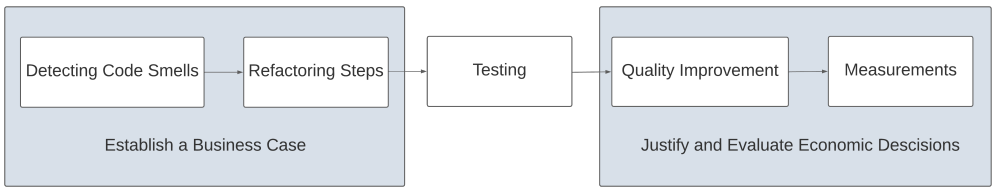

Figure 5.1.: Refactoring process (Extended Model)

5.1. Detecting Code Smells

As stated earlier, not all 24 code smells listed by Martin Fowler will be examined. Instead, the ten most frequently reported code smells have been selected according to the study of conducted by 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. These two programming abstractions are not utilized in the software system, and are thus not relevant in the detection. In order to revisit the code smell catalog with their corresponding explanation, refer to Table 3.5.

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 every single code smell of the software system. Instead, the aim is 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 having automatic detection, limiting the amount of smells is reasonable. A detection approach based on automatic tools is unfortunately not available for the python programming language, as will be alluded to in the following section.

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 metrics-based approach. This approach measures source code elements and takes decisions based on threshold values (Menshawy et al., 2021). It fulfills two important requirements necessary for the following work. On the one hand, it contains automation to a certain extent, increasing the efficiency and standardization compared to manual approaches.

Further, it can be used for the python programming language, which is an essential prerequisite for the software system at hand. Menshawy additionally points out that a metrics-based approach does not provide metrics for every code smell. Despite this, for the eight smells relevant for the thesis, an appropriate metric was found (see Table 5.1).

Using a detection approach utilizing automated tools would have been more attractive, but is unfortunately not possible. It is the most used approach (Menshawy et al., 2021), but availability strongly depends 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 the java programming language, which was supported by 48% of the tools and python, which 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 can only detect a small subset of the code smells, visualization supported the detection of code smells by incorporating a dependency graph (see figure 6.2).

5.1.2. Sonargraph

The detection of code smells relied heavily on a code analyzer tool called *Sonargraph*. Its usage proved 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 a diverse selection of metrics. These metrics provided insights to find problems regarding the entire project and its individual modules. For each metric, Sonargraph offered explanations, which allowed for an appropriate metric to be found for the relevant code smells.

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 individual files. For instance, this was essential for metrics inspecting the dependencies between modules. In addition, it was beneficial not having to rely on a multitude of software tools, by having just one software solution.

It is important to note that there was no appropriate metric that could 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 independent of the programming language. To detect duplications of code in programs, the software does not need to understand the syntax. This same mechanism could theoretically detect duplications in other types of text that are not programs.

5.1.3. Metrics Overview

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)	

Table 5.1.: Metrics Overview

Table 5.1 presents the corresponding metric for each code smell. In addition, for certain 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 terms of the detection approach. For example, as mentioned before, the subgroup 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. Meaning that only code duplicates are shown if they surpass this amount.

Code smells in the subgroup named complexity, follow a typical metrics-based approach using threshold values. The threshold values 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, Feature Envy and Divergent change respectively are bound to the coupling and cohesion of the software system. We use the metric *Physical cohesion* for both Feature envy and Divergent change, and *Physical coupling* for Shotgun Surgery. In Sonargraph the metric*Relational cohesion*, which would better detect Feature envy, is not provided for the python language. Consequently, we will use Physical cohesion for the detection of this code smell as well. The metrics themselves measure 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 lines of code to the software system, we naturally increase the amount of expected dependencies. Therefore, in order to appropriately argue the findings, additionally incorporating a dependency graph seemed necessary to evaluate the prominence of these code smells (refer to 6.1).

5.2. Testing

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 3.1. 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 avoiding the changing of features, programmers can alleviate the risks associated with changing the codebase, which predominantly includes the introduction of new bugs. As a result, to demonstrate that the behavior of the code base was not altered during refactoring, testing is necessary. Therefore, without testing, we would get a false perception of the code quality improvement. This is due to the fast that potentially negative alterations would not be 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

Robots and other CPS react to information from the physical world and must operate safely even in the presence of uncertainties (Geiß, n.d.). In such a dynamic real-world environment, Kapur (2020) describes CPS testing, as a time-consuming and a particularly complicated task for programmers. He points out that this is due to the difficulty to ensure that a CPS will behave as expected. 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 introduction of automated or semi-automated testing methods. Here, the feasibility of testing is limited to the hardware components of the CPS. For instance, a CPS with physical motions requires a considerable amount of time to run tests. Additionally, it potentially demands 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), also referred to as *digital twin*. Such solutions do exist in practice, but are unfortunately unavailable to our software system.

This challenging physical environment of CPS results in programmers having to satisfy multiple layers at the same time, when testing their software. The layers

of CPS software testing contain verifying software, hardware, network, and the integration of all these components to work as a single system (Abbaspour Asadollah et al., 2015). In general, there is a high interest to use automated testing, as manual testing is more likely to produce errors (Turlea, 2019). Despite automated tests having multiple advantages, such as more testing in less time, 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 a dynamic and customized environment (Taipale et al., 2011). Consequently, software testing demanding developing a testing 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 additional endeavor.

One characteristic of the physical nature of CPS is that we are able to make visual observations. This feature can be viewed as an advantage to advocate manual testing of a particular CPS. Whether the observations of a CPS can be used in testing, is strongly dependent on the type of CPS. 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.

Overall, the lack of software tests and the ability to visually observe behavioral changes encouraged the decision to include manual testing as the preferred method. Even though test coverage is not thorough, this approach of testing can to some extent support the claim that no alterations are present within a particular processes. As a result, it is an approach that suits the preconditions within our context, as it investigates changes within the scope of processes.

5.2.3. Method selection

To reduce the inherent risks of uncertainty in manual testing, 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 platform Camunda is used to automate the business processes and the domain specific language Gherkin 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.

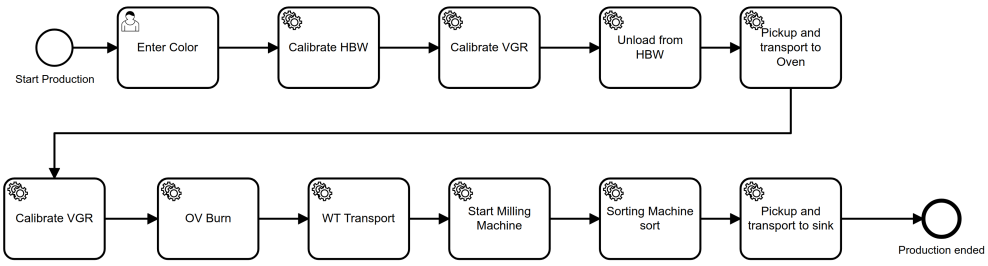

Figure 5.2.: Camunda Production Process

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 and can be executed by the Camunda BPMN platform. This allows the automation of these processes. For the testing, the production process has been chosen to best reflect the software system, as it includes all components of the factory. As a result, the testing can cover a major part of the software system's functionality. Using the contents of this file, the Camunda platform is able to sequentially execute predetermined HTTP requests, which otherwise would have been manually conducted. Performing these requests manually would not only be very tedious,

but also take away standardization, resulting in the risk of making mistakes. In addition, the BPMN notation used by the Camunda platform allows us to visually inspect, which parts of the process are currently running. This is an exceptionally useful feature, when trying to find out where a current issue might be located, in case the tests fail.

Gherkin on the other hand is not a software application. Instead, it is a domain specific language that is generally used in conjunction with software. Hereby, one can formulate behavior in the form of specifications written in plain text, which is understandable by both humans and computers ("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. Overall, this allows us to objectively test whether the software system fulfills these specifications, that can be routinely repeated in a standardized way. Refer to B.1 to view an illustrative example in which the Gherkin keywords are directly applied to the software system.

5.3. Measuring Quality Improvement

5.3.1. Importance

Having incorporated testing in our refactoring model, we are now in a position to consider methods to measure code quality after refactoring. Measuring code quality improvements allow 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 these measurements, we can reflect on refactoring endeavors, which helps

us in making future decisions related to refactoring. Moreover, if the metrics are sufficiently accurate, one could argue when to initiate and when to stop refactoring, by using them as indicators. However, measurements are not limited to the comparison of the initial and final state of a refactoring. Measurement results can be further expanded by periodically measuring code quality. For instance, this allows us to individually attribute the quality improvement in relation to refactoring specific code smells.

One possible metric that allows for quantifiable measurements, is the maintainability index, which measures how maintainable the source code is. This measure would be suitable for our case, as it offers single-valued quantification of code quality, while also considering multiple aspects of code quality. A more detailed account of the index will be given in the following section.

5.3.2. Maintainability Index

or To compute 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 include all of these in its computation. The documentation of the software Radon offers 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:

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

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

Derivative (used by Radon):

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

- C = Percentage of commented lines

The original equation 5.1 is included, as it makes it easier to understand how the individual components contribute to the final result. The second formula, which is used by Radon, is a derivative of the original formula. It is preferred 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 of SLOC, when the source code contains a lot of commented lines. Although not specifically mentioned in the documentation, this is presumably the reason Radon included this additional variable.

5.3.3. Methods used for taking measurements

The act of taking measurements is also a design decision that was consciously carried out. At first, 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 a tedious task. Considerable amount of time would be spent for something that an easily be

automated. There would also be no clear indication on the right amount of measurements to take in a routinized way. By making too many measurements, not only a lot of time is required, but it would also be difficult to differentiate between the individual measurements. Conversely, by having too few measurements, there would not be sufficient information available to arrive at meaningful conclusions.

Regarding the timing of the measurements, it became clear that measuring after each commit in the git version control program would be a suitable approach. That way each measurement could directly be attributed to individual commits, which includes information on code changes (commit message) and the possibility to check the difference to the previous state of the program (diffs).

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 the bash command language, 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 had been used, only a manual approach would have been feasible, if not directly implementable by the graphical software itself.

The bash script performs two distinct functions: computing and saving. Initially, it executes radon to compute the maintainability index, including its subcomponents. 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 attribution of the measurements to each 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 refactoring.

6. Results and Main Findings

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)				

Table 6.1.: Metrics Overview (Prioritized, Values)

Figure 6.1 presents an overview of the detected code smells. In addition to figure 5.1 from the previous chapter, using the value of their metrics the code smells are now prioritized, indicated by their color. First, smells indicated by red are of high priority and should be refactored. Second, smells indicated with yellow suggests that a potential code smell exists. Lastly, smells that are noted to be not a problem, are indicated by the color green.

Code duplication has been identified as the single code smell with the highest priority. This type of smell exists in 24 unique blocks of code, measured by surpassing the threshold of 25 similar lines. Adding these duplications together

leads to 2226 lines of code, which amounts to roughly a fourth of the entire software. In general, it is better to unify parts of code if one sees the same code structure in more than one place (Fowler, 2018). Additional reasons to avoid this particular smell are unnecessary added complexity, necessity to change methods in multiple locations, and increased difficulty to read code. As a result, it is highly recommended to reduce the code duplications in the software system, before migrating towards microservices. Moreover, the mentioned issues contribute to the overarching problem of the software system being more prone to errors when changing code.

Long Method and *Lazy Class* have been identified to potentially signal a code smell. Regarding the smell *Long Method*, one method surpassed the threshold of 100 Statements by having 114 Statements. Long methods could be a problem as, the longer they are, the more difficult it is to understand them (Fowler, 2018). Even though having only one method that exceeds our threshold is not problematic, it is still useful to evaluate whether it would be appropriate to split it up. Further, six classes have been detected to potentially fall into the category *Lazy class*. These individual classes are similar to each other in regard to being in the same directory and representing each station of the factory. Here, we ought to find out whether the classes have sufficient responsibility to justify their existence (Lacerda et al., 2020). In this case, it only makes sense to either refactor all of them or none. This thesis will not reach conclusions on whether *Long Method* and *Lazy Class* constitute a significant smell. Nonetheless, the author encourages investigating this issue further and consider the benefits of refactoring these smells.

Both *Long parameter list* and *Lazy Class* were determined to not depict a code smell. *Long parameter list* did not surpass the threshold of eight parameters. Methods had a maximum of five parameters, which does not constitute to be a problem. Regarding *Lazy Class*, one file exceeded the threshold of 900 lines

during the detection. This file, however, is the web application of the software system and not a class. It is responsible for initializing the web server, executing web services and starting the app. Given the current purpose of the software system, it is not suitable to deconstruct this file. However, in the act of moving towards microservices, the web application needs to be separated for each station.

The code smells *Shotgun Surgery* and *Divergent change* lacked evidence to suggest them being a code smell. Foremost, their respective metrics *physical coupling* and *physical cohesion* did not offer enough evidence, to reach a conclusion. Here, the dependencies among and within modules were less than seven.

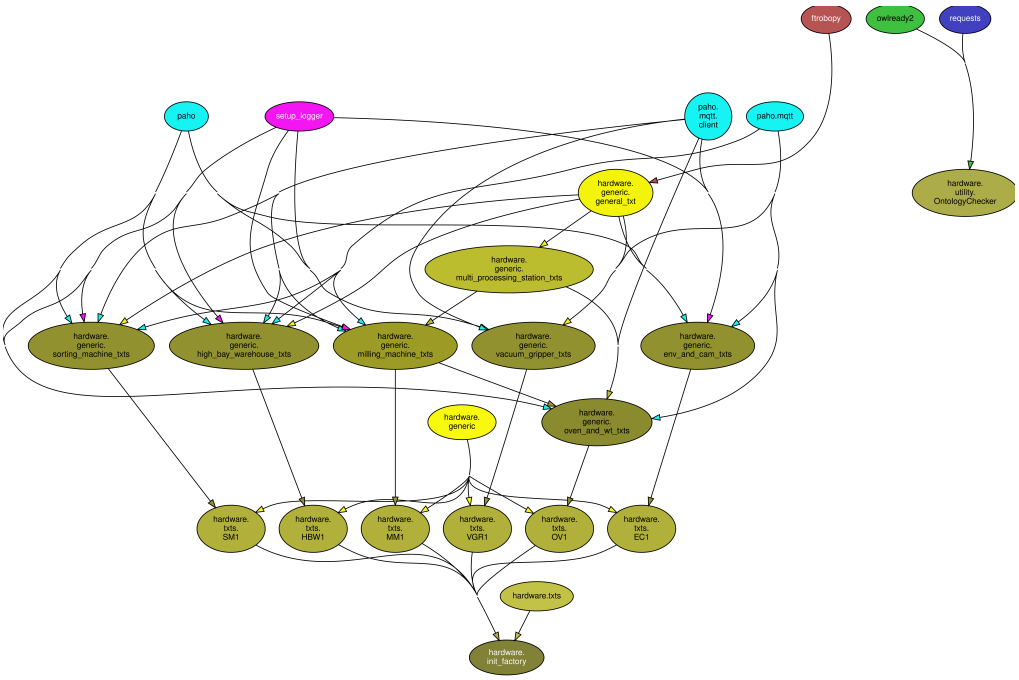

Figure 6.1.: Dependency Graph

A dependency graph in figure 6.1 is included to gain further knowledge on the coupling and cohesion of our software system. In particular, it depicts the dependencies within the classes of the hardware directory of the software

system. By closer inspection, we observe one key characteristic of the modular structure. The characteristic being that most of the modules are related to individual stations, which indicate single responsibility associated to each station. There are no visible dependencies between the stations. This was visible in our analysis of code duplication, which showed that many functionalities are duplicated among the separate stations. As a consequence, having these characteristics, the code is argued to be low in coupling. This means that there are not many dependencies among the modules, which leads to the conclusion that *Shotgun Surgery* should not be considered a code smell. This certainly a good indicator that refactoring towards microservices is feasible in this regard.

Similarly, we can assume that given this modular structure, the software is also high in cohesion. In other words, the structure following this single responsibility principle suggests the code smell *Divergent change* and *Feature Envy* not to be an issue. Importantly, there should be a note of caution to these conclusions. Relying on looking at the modular structure to reach a conclusion about cohesion, does not suffice to disregard a code smell. Hence, the author can not provide a complete guarantee and only offers a prediction.

6.2. Refactoring based on an Example

We now turn to discussing how one would approach refactoring the code duplication within our software system. It has been mentioned in the beginning of the chapter that the volume of 24 duplicated blocks is in fact a duplication of 24 unique blocks. This means that each of these 24 duplications occurs multiple times within the code base. Both the amount of occurrences and block length, are crucial in determining which code duplications to start refactoring. The thesis will provide an overview of one particular code duplication, which has

the largest block size and the maximum occurrence of six times. As may be presumed, each duplication is associated with one of the six stations. The particular duplication regards one method, which is responsible for setting the motor speed of the individual stations. Figure 6.2 provides a better understanding of the underlying structure of the code duplication that needs to be refactored.

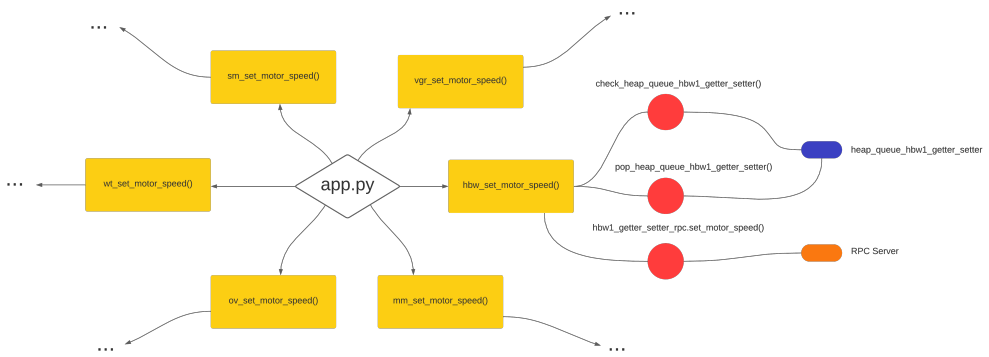

Figure 6.2.: Code duplication Illustration (Before Refactoring)

The six code duplicates can all be found in the web application of the software system. The content visualized for the particular method *hbw_set_motor_speed()*, is the same for the remaining five methods, indicated by the three dots. These methods are related both a queue mechanism and an RPC call that calls a station-specific method setting the motor speed. After carefully inspecting these methods, we arrive at the conclusion that these methods are code duplication themselves. Particularly, there exists a nested structure of code duplicates, in three hierarchical levels. In order to refactor our code duplication, we ought to also get rid of these nested duplications, as they are closely related to the method.

Furthermore, by refactoring all of these methods, we will inevitably realize that these methods are not only used to set the motor speed. Namely, they are also contained at various location in the software system. Consequently, after unifying these methods, we can simplify code even more by transferring them

to other locations. As an example, in addition to the method that sets the motor speed, there exists additional methods that *get the motor speed* and *reset the motor speed*. These all include the same methods that are responsible for the queue mechanism, which further provides opportunities to simplify the code base of the software system.

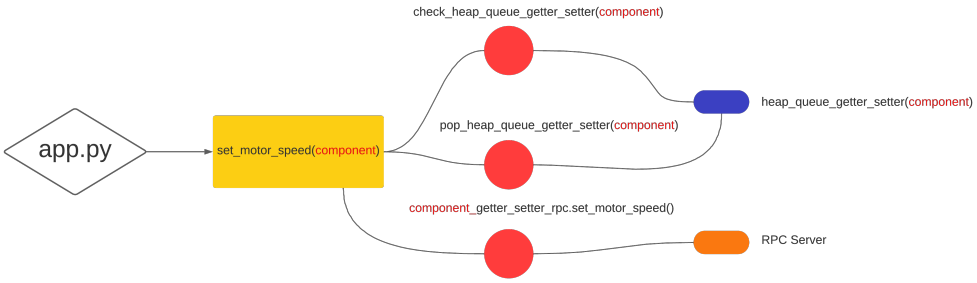

Figure 6.3.: Code duplciation Illustration (After Refactoring)

Figure 6.2 presents an overview of how the code is supposed to be structured after refactoring the code duplication. We immediately see that these changes result in a simpler structure, with much less code. In addition, using this example, we showed refactoring to be an activity that is dynamic. We have started with one unique code duplication, but ended with proposing various structural changes to improve the software system. The nested structure demonstrated the impact of the code smell to be much larger than anticipated. Hence, by refactoring one issue, we might discover another issue that is related to the previous issue.

7. Conclusion

This thesis aimed to identify additional areas in software engineering that relate to refactoring. Based on a case-study approach, an overarching framework was developed to present a refactoring process. The results reveal that refactoring is not a distinct activity having significant influences from the detection of code smells, testing environment and the measurement of quality improvements after refactoring. This thesis has also shown many connections between refactoring and the business sphere. Technical debt as a concept has presented the financial incentives of refactoring and proved to be a suitable communication device to justify refactoring decision-making.

The in-depth analysis of selecting appropriate methods for our case study has shown that many refactoring methods are not generalizable to other software systems. It was highlighted that the programming language influences the availability of automatic tools. Moreover, CPS have presented challenges in testing due to their physical environment and limitations to automate testing. The thesis raised an important interest in measuring the improvement of quality attributes after refactoring. It introduced the maintainability index as a possible measurement technique and presented the importance of taking continuous and automated measurements.

In the detection of code smells, the most obvious internal problem in the software system was the duplication of code. Further, metrics indicated the

possibility of two additional smells: Long method and Lazy Class. Not enough evidence was found to reliably suggest refactoring them. In addition, the thesis examined the modular structure of the software system. It revealed indications that support the feasibility of a migration towards microservices.

Finally, an attempt was made to illustrate the refactoring of code duplication. Starting with just one example, the thesis could propose various structural changes to improve the software system. The results of this illustration suggest refactoring to be a dynamic activity.

The present thesis extends our knowledge of the term refactoring by offering a more comprehensive view. This new understanding should help guide practitioners to make practical decision in their refactoring endeavors. More broadly, by having presented key drivers of refactoring, the thesis offers new opportunities in the development of comprehensive refactoring approach for both software engineering and business.

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."

23.05.2022

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

Appendix A.

Git Hook Bash Script

Figure A.1.: Bash script (executed before to Git Commit)

```
#!/bin/sh
#
# An example hook script to verify what is about to be committed.
# Called by "git commit" with no arguments.  The hook should
# exit with non-zero status after issuing an appropriate message if
# it wants to stop the commit.
#
#
# To enable this hook, rename this file to "pre-commit".

if git rev-parse --verify HEAD >/dev/null 2>&1
then
        against=HEAD
else
        # Initial commit: diff against an empty tree object
        against=$(git hash-object -t tree /dev/null)
fi

# If you want to allow non-ASCII filenames set this variable to true.
allownonascii=$(git config --type=bool hooks.allownonascii)

# Redirect output to stderr.
exec 1>&2

# Cross platform projects tend to avoid non-ASCII filenames; prevent
# them from being added to the repository. We exploit the fact that the
# printable range starts at the space character and ends with tilde.


if [ "$allownonascii" != "true" ] &&
        # Note that the use of brackets around a tr range is ok here, (it's
        # even required, for portability to Solaris 10's /usr/bin/tr), since
        # the square bracket bytes happen to fall in the designated range.
        test $(git diff --cached --name-only --diff-filter=A -z $against |
          LC_ALL=C tr -d '[ -~]\0' | wc -c) != 0
then
        cat <<\EOF
Error: Attempt to add a non-ASCII file name.
This can cause problems if you want to work with people on other platforms.
To be portable it is advisable to rename the file.
If you know what you are doing you can disable this check using:
  git config hooks.allownonascii true
EOF
        exit 1
fi


```

```
# ---
# CUSTOM PART STARTS HERE
# ---

# Date
now=$(date +"%m_%d_%H00")

mkdir /home/fabian/thesis/results/metrics
filename=/home/fabian/thesis/results/metrics/log_${now}

# Break if Report already exists
if [ -f "${filename}.txt" ]; then return; fi


# Exact date as a header
date >> ${filename}.txt

echo "\n--- APP ----------------------------------" >> ${filename}.txt

# Add Metrics to File
echo "\n---\nComplexity\n---" >> ${filename}.txt
echo "\nCyclic Complexity:" >> ${filename}.txt
radon cc app.py -a | grep complexity >> ${filename}.txt
echo "\nHalstead:" >> ${filename}.txt
radon hal app.py >> ${filename}.txt
echo "\nMaintainability Index:" >> ${filename}.txt
radon mi app.py -s >> ${filename}.txt
echo "\n---\nReadability\n---\n" >> ${filename}.txt

echo "Pylint Rating:" >> ${filename}.txt
echo $(python3 -m pylint app.py | grep rated) >> ${filename}.txt

echo "\n--- Hardware/generic ----------------------" >> ${filename}.txt

# Add Metrics to File
echo "\n---\nComplexity\n---" >> ${filename}.txt
echo "\nCyclic Complexity:" >> ${filename}.txt
radon cc hardware/generic -a | grep complexity >> ${filename}.txt
echo "\nMaintainability Index:" >> ${filename}.txt
radon mi hardware/generic -s >> ${filename}.txt
echo "\n---\nReadability\n---\n" >> ${filename}.txt

echo "Pylint Rating:" >> ${filename}.txt
echo $(python3 -m pylint hardware/generic/ | grep rated) >> ${filename}.txt
```

Appendix B.

Formalization of Gherkin Specifications

```
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
```

Figure B.1.: Gherkin Specifications applied to software system

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.

Appendix C.

Sonargraph Graphical Interface

C.1. Metric View

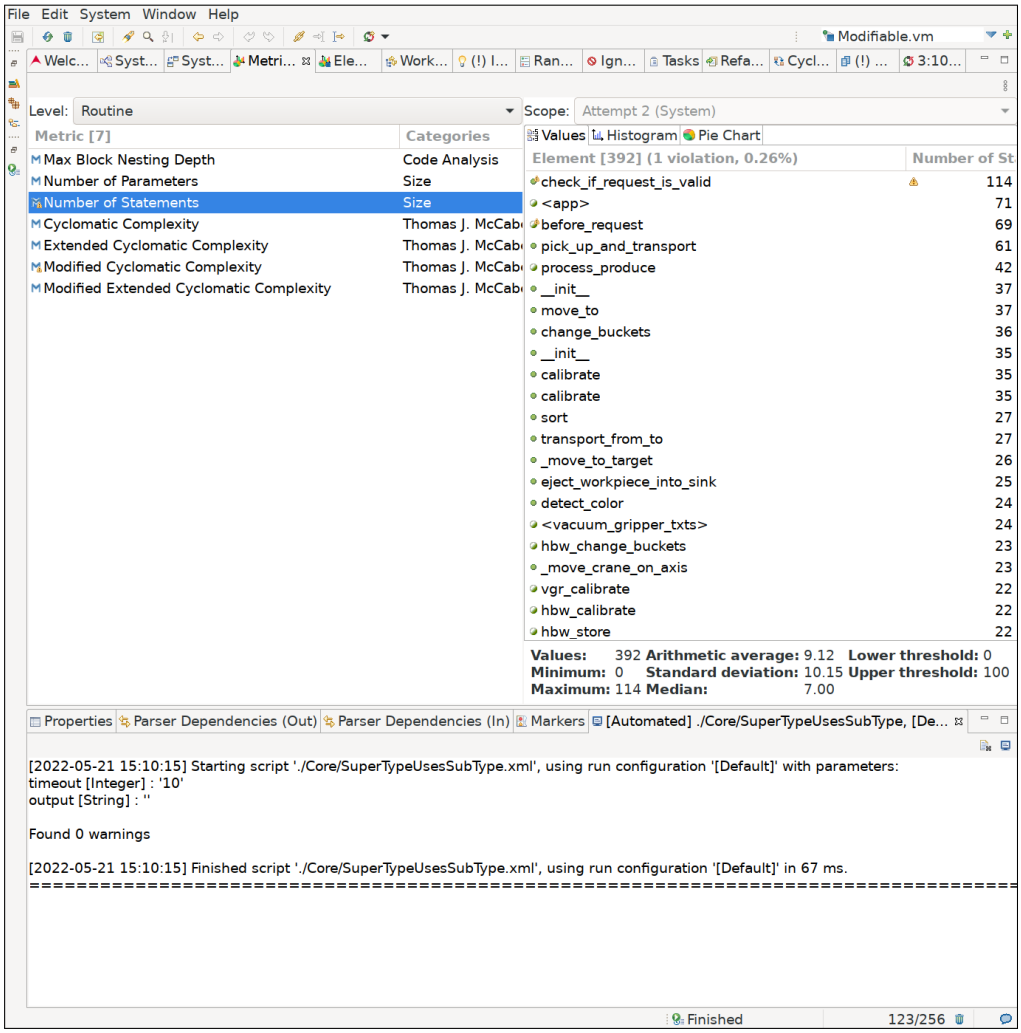

Figure C.1.: Evaluation of Metrics in Sonargraph

C.2. Duplications View

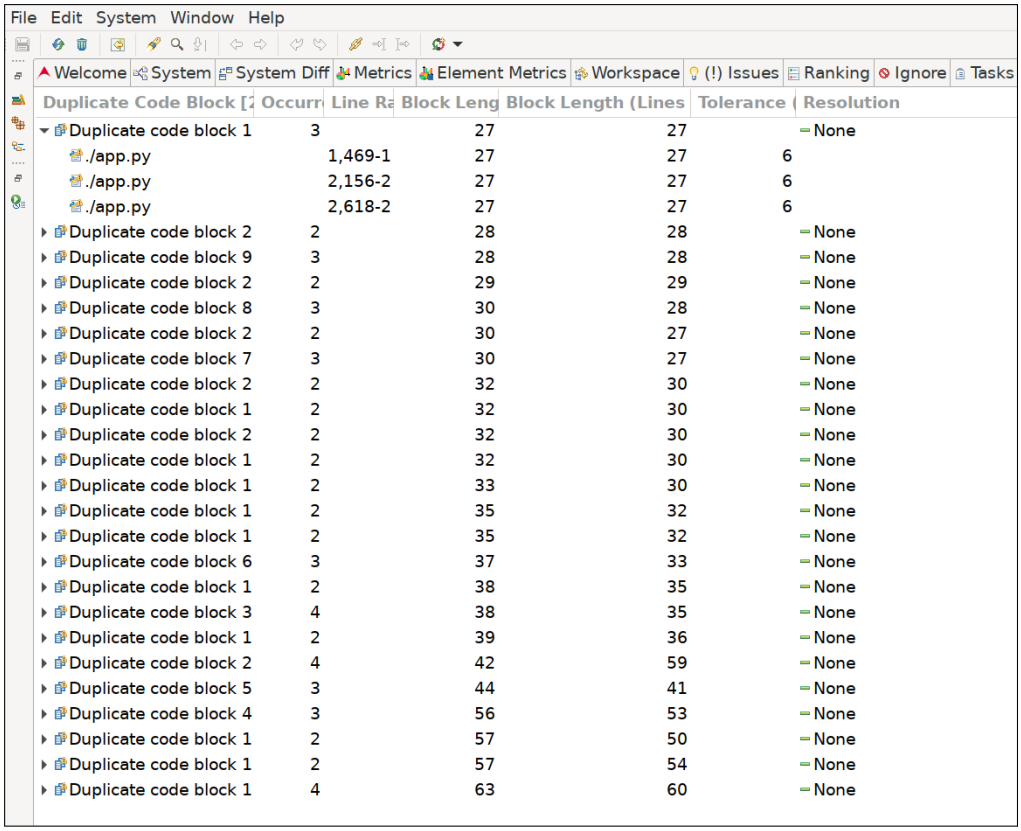

Figure C.2.: Overview of Code Duplications

C.3. Side by Side comparison of duplicates

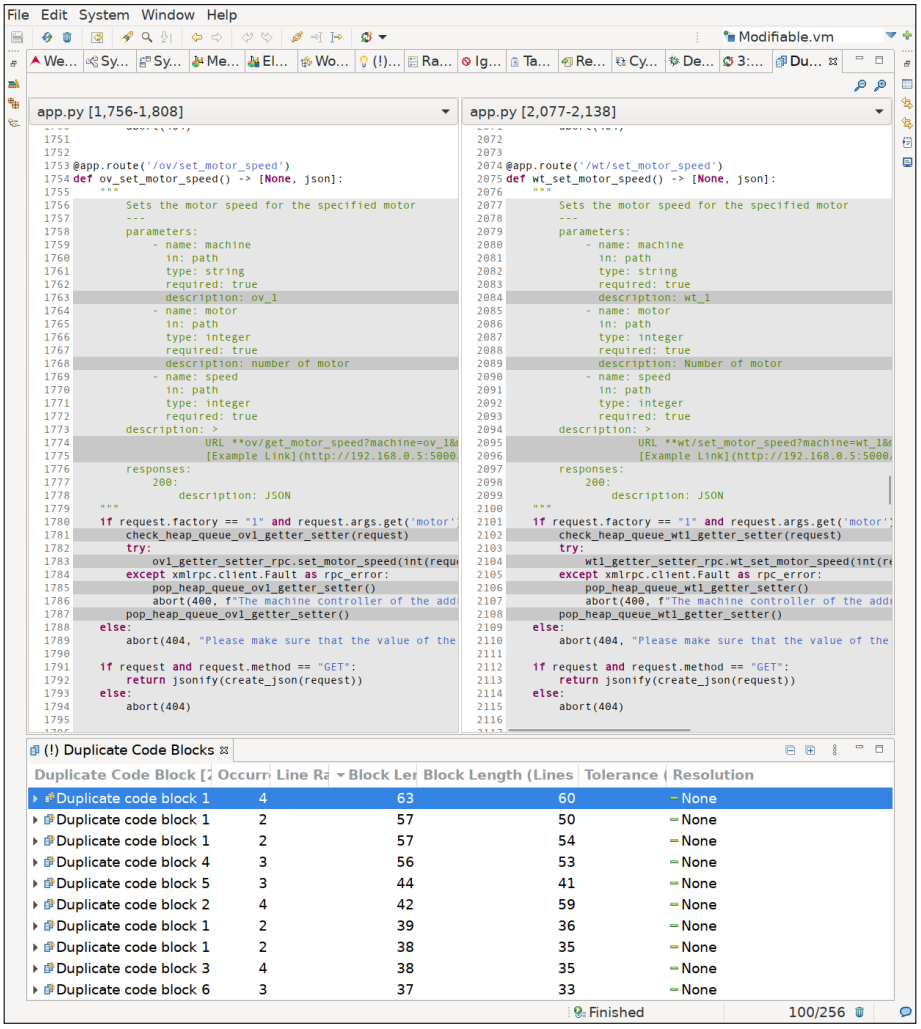

Figure C.3.: Duplicated Code Block Source View

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 [cit. on p. 4]. *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.

Chen, Z., Chen, L., Ma, W., & Xu, B. (2016). Detecting code smells in Python programs. *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, 18–23.

Fowler, M. (2018, November 30). *Refactoring: Improving the Design of Existing Code* (2nd edition) [cit. on p. 5, 45, 47, 53, 56, 57, 71-72]. Addison-Wesley Professional.

Fowler, M. (2019, May 21). *Technical Debt*. martinfowler.com. Retrieved May 9, 2022, from https://martinfowler.com/bliki/TechnicalDebt.html

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2012, March 9). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

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/

Jazdi, N. (2014). Cyber physical systems in the context of Industry 4.0 [cit. on p. 1]. *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*, 1–4. https://doi.org/10.1109/AQTR.2014.6857843

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 [cit. on p. 1]. *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 [cit. on p. 2, 15]. *Journal of Systems and Software*, *167*, 110610. https://doi.org/10.1016/j.jss.2020.110610

Malburg, L., Seiger, R., Bergmann, R., & Weber, B. (2020). Using Physical Factory Simulation Models for Business Process Management Research [cit. on p. 95-97, 99]. In A. Del Río Ortega, H. Leopold, & F. M. Santoro (Eds.), *Business Process Management Workshops* (pp. 95–107). Springer International Publishing. https://doi.org/10.1007/978-3-030-66498-5_8

Mens, T., & Tourwé, T. (2004). A survey of software refactoring [cit. on p. 129]. *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 [cit. on p. 1]. *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

Osterrieder, P., Budde, L., & Friedli, T. (2020). The smart factory as a key construct of industry 4.0: A systematic literature review [cit. on p. 1]. *International Journal of Production Economics*, *221*, 107476. https://doi.org/10.1016/j.ijpe.2019.08.011

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

Seiger, R., Zerbato, F., Burattin, A., García-Bañuelos, L., & Weber, B. (2020). Towards IoT-driven Process Event Log Generation for Conformance Checking in Smart Factories [cit. on p. 22]. *2020 IEEE 24th International Enterprise Distributed Object Computing Workshop (EDOCW)*, 20–26. https://doi.org/10.1109/EDOCW49879.2020.00016

Sonnleithner, L., Oberlehner, M., Kutsia, E., Zoitl, A., & Bacsi, S. (2021). Do you smell it too? Towards Bad Smells in IEC 61499 Applications [cit. on p. 1]. *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

Turlea, A. (2019). Model-in-the-Loop Testing for Cyber Physical Systems. *ACM SIGSOFT Software Engineering Notes*, 44(1), 37–41. https://doi.org/10.1145/3310013.3310019

Wang, L., Törngren, M., & Onori, M. (2015). Current status and advancement of cyber-physical systems in manufacturing [cit. on p. 517]. *Journal of Manufacturing Systems*, 37, 517–527. https://doi.org/10.1016/j.jmsy.2015.04.008