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 Exploration of the Fourth Industrial Revolution

2.1 Terminology and Context

2.2 Defining Smart Factories

2.3 Economic Relevance

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 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 paper.

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 paper 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.

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) adds 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 paper (see Section 3.3).

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. 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.

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 best described by the hat analogy proposed by Fowler and Beck 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 ugly 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 (X).

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.

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.

3.2 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.

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 paper, 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 will briefly summarize the top ten code smells according to Lacerda et al. (2020).

Table 1
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
Refused Bequest	It 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 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 int12340ernal 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.

3.3 The Business Case for Refactoring

4 Methodological approach

4.1 Motivation and Procedure

4.2 Means of data collection

4.3 Methods of analysis

4.4 Limitations and justification

5 Main Findings

6 Discussion

7 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

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 (cit. on p. 5)

Bass, L. (1998). *Software architecture in practice*. Addison-Wesley. (Cit. on p. 4) Open Library ID: OL668208M.

Fowler, M. (2018, November 30). *Refactoring: Improving the Design of Existing Code* (2nd edition). Addison-Wesley Professional. (Cit. on pp. 3, 6–10).

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 (cit. on p. 4)

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 (cit. on pp. 9, 10)

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2004.1265817 (cit. on p. 4)

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 (cit. on p. 4)

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 (cit. on p. 10)