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 such as cleaning and restructuring code are regularly occuring and oftentimes necessary. When refering to these activities by name, many employees describe them as refactoring. Despite of its familiarity, the exact definition of the term *refactoring* is not always self evident. To avoid refering to the term too loosely, it is therefore important to give a precise definition to reference throughout the paper.

Martin Fowler (2018, p. xiv) managed to formulate a definition that is both short and precise. Notably, Fowler is one of the most prominent figure, who has pioneered many concept in the field of refactoring, which validates the utilization of his definition. Fowler defines refactoring 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.

It is therefore an integral part of reactoring to not change the features of the program, while improving the quality of the code. By proper refactoring, we can thus eleviate the risk associated with changing the codebase, while improving the internal structure of the program.

Objective

Spending valuable ressources, while not adding new features, might sound unappealing to many managers and even programmers. Kim et al. (2012, p. 1) mention the problem of not having an immediate benefit when refactoring, unlike new features or bug fixes. In addition, 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. Through continuous modifications and adaptations to new requirements, the code becomes increasinlgy complex and drifts away from original design. As a consequence, a major part of the resources is spent on software maintenance (Mens et al., 2003, p. 1).

However, having Fowler's Definition in mind, we aim to mitigate time spent on tedious maintenance, by improving the quality attributes of the code and thus its internal structure. Conversely, by ignoring the internals of the program, there would exist a continuous increase of debt, which would have to be repaid in the future in the form of maintenance costs. Moreover, by obtaining so called technical debt, future features are more costly to implement and sudden changes are near impossible. The concept of technical debt is an important one to fully grasp, which is why it is further explained in a future section of this paper (see Section 3.3).

Refactoring helps to make the code more readable, as well as improving the internal quality attributes of the software (Mens and Tourwé, 2004, p. 129). Some refactorings remove code redundancy, some raise the level of abstraction, some enhance the reusability, and so on. On a higher level, these quality attributes include whether the software is maintainable, modifiable, testable and much more. The effect of a refactor can to a certain extent be estimated, by the level of improvement in these quality attributes. Consequently, the transformation of the internal structure is the deciding factor on which sucess is measured. Ultimatlely, refactoring adds value if at least one quality attribute has significantly improved.

There is however one caveat when measuring the added value, which is that the external behavior is not altered. As an example, changing the code base could make the program more modulear, but during the process create bugs that didn't exist before. Accordingly, it must be proven that only the

internal structure has changed. This is achieved by writing tests beforehand and then testing the features after the refactor. Thereby, software tests is a major component when deciding the sucess of a refactor.

Benefits

There needs to be less justification when it comes to time spent on new features and fixing 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 that code works, and struggles to follow it, one has to do something about it. Hence, we can conclude that the potential improvement in quality, doesn't warrant a refactor by itself.

It is crucial to state that refactoring should not be considered to be a massive, obscure undertaking. Even if there is a demand of 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, as code being easier to understand and cheaper to modify is vital in software development. Therefore, with a project that is in active development, active monitoring of the code quality and therefore being willing to do continous improvements by means of refactorings is advisable.

As one may presume, such activities can be done in parralel to the software development process. 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 a metaphor 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, he 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 efficientlyspent on adding new capabilities.

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

This relationship between refactoring and adding features is vital to grasp, in order to understand the timing of each activity. This relationship is especially important to understand that planned refactoring are not always wrong. In particalur, if we comprehend 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.

When we compare it to our previous definition of small steps ... There is indeed a difference in scope, when it comes to the amount of small refactoring steps needed to be applied. With a certain projects doing continous refactoring during the development process might not suffice. In some cases however an overarching refactor is necessary.

3.2 Formalization of Design Patterns

Knowing just the importance and applicatons of refactoring does not suffice, as one still needs to understand when to do the 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. 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. 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.

Refactoring can then can be thought of as the process of getting rid of these 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

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 Sample Chapter

This chapter is an adopted version of a single chapter of **KeithThesis** thesis template **KeithThesis** in its version from 2011-12-11.

The reason why **KeithThesis** is not recommended to be used instead of this template is its more "traditional" LaTeX implementation. But the information contained regarding "How to write a thesis" is generally brilliant and worth reading.

Using this chapter here is meant as a teaser. If you do like this chapter, please go and download the full template to read its content: **KeithThesis**.

What was modified from the original chapter:

- strikethrough of bad examples
- minor typographical details
- technical modifications
  - moved citations from `\citet{}` and `\citep{}` to `\textcite{}` and `\cite{}`
  - changed quoting style to `\enquote{}`
  - created various commands and environments to encapsulate format

The classic reference for English writing style and grammar is **StrunkWhite**. The original text is now available for free online **Strunk**, so there is no excuse at all for writing poor English. Readers should consult it first, then continue reading this chapter. Another good free guide is **NASAGuide**.

Zobel-WritingCompSci and **BugsInWriting** are guides specifically aimed at computer science students. **Phillips-HowGetPhD** gives practical advice for PhD students.

The following Sections 7.3 and 7.4 are adapted from the CHI'94 language and writing style guidelines.

7.1 Some Basic Rules of English

There are a few basic rules of English for academic writing, which are broken regularly by my students, particularly if they are non-native speakers of English. Here are some classic and often encountered examples:

- *Never* use I, we, or you.
  Write in the passive voice (third person).

  Bad: ~~You can do this in two ways.~~

  Good: There are two ways this can be done.

- *Never* use he or she, his or her.
  Write in the passive voice (third person).

  Bad: ~~The user speaks his thoughts out loud.~~

  Good: The thoughts of the user are spoken out loud.

  See Section 7.4 for many more examples.
- Stick to a consistent dialect of English. Choose either British or American English and keep to it throughout the whole of your thesis.
- Do *not* use slang abbreviations such as "it's", "doesn't", or "don't". Write the words out in full: "it is", "does not", and "do not".

  Bad: ~~It's very simple to…~~

  Good: It is very simple to…

- Do *not* use abbreviations such as "e. g." or "i. e.".
  Write the words out in full: "for example" and "that is".

Bad: ~~. . . in a tree, e. g.the items. . .~~

Good: . . . in a tree, for example the items. . .

- Do *not* use slang such as “a lot of”.

  Bad: ~~There are a lot of features. . .~~

  Good: There are many features. . .

- Do *not* use slang such as “OK” or “big”.

  Bad: ~~. . . are represented by big areas.~~

  Good: . . . are represented by large areas.

- Do *not* use slang such as “gets” or “got”.
  Use “becomes” or “obtains”, or use the passive voice (third person).

  Bad: ~~The radius gets increased. . .~~

  Good: The radius is increased. . .

  Bad: ~~The user gets disoriented. . .~~

  Good: The user becomes disoriented. . .

- *Never* start a sentence with “But”.
  Use “However,” or “Nevertheless,”. Or consider joining the sentence to the previous sentence with a comma.

  Bad: ~~But there are numerous possibilities. . .~~

  Good: However, there are numerous possibilities. . .

- *Never* start a sentence with “Because”.
  Use “Since”, “Owing to”, or “Due to”. Or turn the two halves of the sentence around.
- *Never* start a sentence with “Also”. Also should be placed in the middle of the sentence.

Bad: ~~Also the target users are considered.~~

Good: The target users are also considered.

- Do *not* use “that” as a connecting word.
  Use “which”.

  Bad: ~~. . . a good solution that can be computed easily.~~

  Good: . . . a good solution which can be computed easily.

- Do *not* write single-sentence paragraphs.
  Avoid writing two-sentence paragraphs. A paragraph should contain at least three, if not more, sentences.

7.2 Avoid Austrianisms

I see these mistakes time and time again. Please do not let me read one of them in your work.

- “actual” $\neq$ “current”
  If you mean “aktuell” in German, you probably mean “current” in English.

  Bad: ~~The actual selection is cancelled.~~

  Good: The current selection is cancelled.

- “allows to” is not English.

  Bad: ~~The prototype allows to arrange components. . .~~

  Good: The prototype supports the arrangement of components. . .

- “enables to” is not English.

  Bad: ~~it enables to recognise meanings. . .~~

  Good: it enables the recognition of meanings. . .

- "according" ≠ "corresponding"

  Bad: ~~For each browser, an according package is created.~~

  Good: For each browser, a corresponding package is created.

- "per default" is not English.
  Use "by default".

  Bad: ~~Per default, the cursor is red.~~

  Good: By default, the cursor is red.

- "As opposed to" is not English.
  Use "In contrast to".

  Bad: ~~As opposed to C, Java is object-oriented.~~

  Good: In contrast to C, Java is object-oriented.

- "*anything*-dimensional" is spelt with a hyphen.
  For example: two-dimensional, three-dimensional.
- "*anything*-based" is spelt with a hyphen.
  For example: tree-based, location-based.
- "*anything*-oriented" is spelt with a hyphen.
  For example: object-oriented, display-oriented.
- "*anything*-side" is spelt with a hyphen.
  For example: client-side, server-side.
- "*anything*-friendly" is spelt with a hyphen.
  For example: user-friendly, customer-friendly.
- "*anything*-to-use" is spelt with hyphens.
  For example: hard-to-use, easy-to-use.
- "realtime" is spelt with a hyphen if used as an adjective, or as two separate words if used as a noun.

  Bad: ~~. . . using realtime shadow casting.~~

  Good: . . . using real-time shadow casting.

  Bad: ~~. . . display the object in realtime.~~

  Good: . . . display the object in real time.

7.3 Clear Writing

The written and spoken language of your thesis is English as appropriate for presentation to an international audience. Please take special care to ensure that your work is adapted to such an audience. In particular:

- Write in a straight-forward style, using simple sentence structure.
- Use common and basic vocabulary. For example, use "unusual" for "arcane", and "specialised" for "erudite".
- Briefly define or explain all technical vocabulary the first time it is mentioned, to ensure that the reader understands it.
- Explain all acronyms and abbreviations. For example, the first time an acronym is used, write it out in full and place the acronym in parentheses.

  Bad: ~~. . . When using the GUI version, the use may. . .~~

  Good: . . . When using the Graphical User Interface (GUI) version, the use may. . .

- Avoid local references. For example, not everyone knows the names of all the provincial capitals of Austria. If local context is important to the material, describe it fully.
- Avoid "insider" comments. Ensure that your whole audience understands any reference whose meaning you do not describe. For example, do not assume that everyone has used a Macintosh or a particular application.
- Do not "play on words". For example, do not use "puns", particularly in the title of a piece. Phrases such as "red herring" require cultural as well as technical knowledge of English.
- Use unambiguous formats to represent culturally localised things such as times, dates, personal names, currencies, and even numbers. 9/11 is the 9th of November in most of the world.

- Be careful with humour. In particular, irony and sarcasm can be hard to detect if you are not a native speaker.
- If you find yourself repeating the same word or phrase too often, look in a thesaurus such as **Roget**; **RogetII** for an alternative word with the same meaning.

Clear writing experts recognise that part of writing understandable documents is understanding and responding to the needs of the intended audience. It is the writer's job to maintain the audience's willingness to go on reading the document. Readers who are continually stumped by long words or offended by a pompous tone are likely to stop reading and miss the intended message.

7.4 Avoiding Gender Bias

Part of striking the right tone is handling gender-linked terms sensitively. Use of gender terms is controversial. Some writers use the generic masculine exclusively, but this offends many readers. Other writers are experimenting with ways to make English more neutral. Avoiding gender bias in writing involves two kinds of sensitivity:

1. being aware of potential bias in the kinds of observations and characterisations that it is appropriate to make about women and men, and
2. being aware of certain biases that are inherent in the language and of how you can avoid them.

The second category includes using gender-specific nouns and pronouns appropriately. Here are some guidelines for handling these problems:

- Use a gender-neutral term when speaking generically of people:

man	the human race
mankind	humankind, people
manpower	workforce, personnel
man on the street	average person

- Avoid clearly gender-marked titles. Use neutral terms when good ones are available. For example:

chairman	chairperson
spokesman	speaker, representative
policeman	police officer
stewardess	flight attendant

- If you are speaking of the holder of a position and you know the gender of the person who currently occupies the position, use the appropriate gender pronoun. For example, suppose the "head nurse" is a man:

  Bad: ~~The head nurse must file her report every Tuesday.~~

  Good: The head nurse must file his report every Tuesday.

- Rewrite sentences to avoid using gender pronouns. For example, use the appropriate title or job name again:

  Bad: ~~Interview the user first and then ask him to fill out a questionnaire.~~

  Good: Interview the user first and then ask the user to fill out a questionnaire.

- To avoid using the third person singular pronoun (his or her), recast your statement in the plural:

  Bad: ~~Each student should bring his text to class.~~

  Good: All students should bring their texts to class.

- Address your readers directly in the second person, if it is appropriate

to do so:

Bad: ~~The student must send in his application by the final deadline date.~~

Good: Send in your application by the final deadline date.

- Replace third person singular possessives with articles.

  Bad: ~~Every student must hand his report in on Friday.~~

  Good: Every student must hand the report in on Friday.

- Write your way out of the problem by using the passive voice.

  Bad: ~~Each department head should do his own projections.~~

  Good: Projections should be done by each department head.

- Avoid writing awkward formulations such as "s/he", "he/she", or "his/her". They interfere when someone is trying to read a text aloud. If none of the other guidelines has been helpful, use the slightly less awkward forms "he or she", and "his or hers".

Remember, the goal is to avoid constructions that will offend your readers so much as to distract them from the content of your work.

7.5 Titles and Headings in Initial Caps

7.6 Use a Spelling Checker

In these days of high technology, spelling mistakes and typos are inexcusable. It is *very* irritating for your supervisor to have to read through and correct spelling mistake after spelling mistake which could have been caught by an automated spelling checker. Believe me, irritating your supervisor is not a good idea.

So, use a spelling checker *before* you hand in *any* version, whether it is a draft or a final version. Since this is apparently often forgotten, and sometimes even wilfully ignored, let me make it absolutely clear:

Use a spelling checker, please.
Use a spelling checker!
Use a spelling checker, you moron.

7.7 Use a Dictionary

If you are not quite sure of the meaning of a word, then use a dictionary. **DictionaryCom** is a free English dictionary, **DictChemnitz** and **DictLeoOrg** are two very good English-German dictionaries.

7.8 Use a Thesaurus

If a word has been used several times already, and using another equivalent word might improve the readability of the text, then consult a thesaurus. **Roget** and **RogetII** are free English thesauri.

8 Conclusion

Declaration of Authorship

"I hereby declare

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

Date	Signature

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

Appendix

Bibliography

Fowler, M. (2018, November 30). *Refactoring: Improving the Design of Existing Code* (2nd edition). Addison-Wesley Professional. (Cit. on pp. 3, 5–7).

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 p. 7)

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)