Tampere University

Sara Brentini

MANAGING CODE QUALITY IN A SOFTWARE COMPANY WITH REFACTORING

Faculty of Information Technology and Communication Science
Bachelor's thesis
April 2020

ABSTRACT

Sara Brentini: Managing Code Quality in a Software Company with Refactoring
Bachelor's thesis
Tampere University
Degree Programme
April 2020

There has been an increasing interest in code quality, how to maintain it and its effect on a software company. One quality attribute is the maintainability of the source code. Research has shown that software companies use a lot of time and money on software maintenance, thus the question rises how software companies can decrease the use of their time and resources on maintenance. This study aims to determine what is code quality and how the use of refactoring can improve or maintain a code's quality. Refactoring is the practice of editing the external structure and making the code more readable and easier to modify in the future.

The analysis of literature shows that in theory refactoring should increase code quality. The results, however stated that further testing in practice needs to be done on refactoring to be able to measure its impact on code quality.

Keywords: code quality, refactoring, readability, testability, maintainability

TIIVISTELMÄ

Sara Brentini: Koodin laadun ylläpito refaktoroinnin avulla ohjelmistoyrityksessä
Kandidaatintyö
Tampereen yliopisto
Tutkinto-ohjelma
Huhtikuu 2020

Ohjelmistoyritykset ovat kiinnostuneita koodin laadusta, sen ylläpitämisestä ja sen vaikutuksesta yrityksen toimintaan. Yksi koodin laadun ominaisuus on lähdekoodin ylläpidettävyys. Tutkimukset ovat osoittaneet, että ohjelmistoyritykset käyttävät paljon aikaa ja rahaa ohjelmistojen ylläpitoon, joten herää kysymys, kuinka ohjelmistoyritykset voivat vähentää aikansa ja resurssiensa käyttöä koodin ylläpidossa. Tämän tutkimuksen tarkoituksena on selvittää, mitä tarkoittaa koodin laatu ja kuinka refaktorointia käyttämällä voi parantaa tai ylläpitää koodin laatua. Refaktorointi tarkoittaa käytäntöä, jolla muokataan koodin ulkonäköä ja tehdään siitä helppolukuisempaa sekä helpommin muokattava.

Kirjallisten tutkimusten mukaan refaktorointi parantaa koodin laatua ainakin teoriassa. Tutkimustuloksissa kuitenkin todettiin, että vielä on tehtävä lisätestausta refaktoroinnille, jotta sen vaikutus koodin laatuun voidaan mitata.

Avainsanat: koodin laatu, refaktorointi, ylläpidettävyys, testattavuus, luettavuus

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
CC	Cyclomatic Complexity
HC	Halstead Complexity
IDE	Integrated Development Environment
LOC	Lines of Code
MI	Maintainability Index
TDD	Test-Driven Development

1 INTRODUCTION

A software company uses at least 70 % of its time and budget on the maintenance of a software project [4] [10]. With such a high percentage it is ideal for a software company to put effort into how to lower these costs and the time spent on maintenance. Maintenance is highly affected by the quality of the source code, therefore, finding ways to improve this quality can help in saving time and money.

Chapter 2 observes code quality and its importance to a software company. There are three different quality attributes that can contribute to the quality of the source code. These attributes are *readability, testability* and *maintainability*. Readability means how well the source code can be read by a programmer who is not the original author [5]. Testability ensures that changes can be made to the source code without breaking the software [12]. Maintainability indicates the effectiveness, how well modifications can be made to improve, fix or change the software [9]. These attributes can be measured by different quality measurement metrics that can show the current quality of the source code.

To improve or maintain quality of source code, refactoring can be used. This is inspected in chapter 3. Refactoring is a way of making changes to the source code without changing its functionality [6]. Refactoring can improve readability, thus improving the maintainability of the source code. Testability also plays an important role in refactoring, since it allows for safe modifications to be made [6]. There are certain scenarios when refactoring can be applied and when it should be avoided. The end of the thesis covers different practical ways how to implement refactoring and the usefulness of automated refactoring.

2 CODE QUALITY AND ITS ROLE IN A SOFTWARE COMPANY

The biggest factors affecting the success of any software company's projects are quality, time and cost [14]. It can be argued that quality of the source code adds to the quality of the software project. Additionally, it can be disputed that the higher the quality of the code the less time will be spent on irrelevant matters related to the source code, and more time can be put into the essentials. In short, a software company might benefit greatly, if they focus on writing quality code. This chapter covers the meaning and importance of code quality and how it can be measured.

2.1 The Nature of Code Quality

In general, code quality is difficult to define. In one definition, code quality can be defined as code that is readable and easily modified by other developers, includes comprehensive tests, meaningful naming conventions, limited dependencies to ease maintenance, provides a coherent and minimal application programming interface (API) and has no duplication [12].

Code quality focuses mainly on the expressiveness of the code. This expressiveness can be called the readability of the source code. Another important attribute is the testability of the source code. Additionally, there is a code quality attribute that can also be defined as a software quality attribute. This is called maintainability. It is probably one of the only software quality attributes that can be approached through the actual source code of a program [14].

This thesis focuses on the following three quality attributes; readability, testability and maintainability. These attributes are covered in the next subsections to provide an understanding as to how these three increase code quality.

2.1.1 Readability

Readability can be seen as a very basic level attribute that increases code quality. It is a part of the visual aspect of the source code and shows how well the code is read by other developers. Readability touches a wide variety of different topics, such as naming con-

ventions, functions, commenting, formatting, objects and data structures. This subsection explains the contribution of the mentioned topics to code quality.

Code should be written in a clear and understandable way, allowing other developers to spend minimal time to understand it. [5]. Some programmers might focus on solving the current problem and write their code quickly without giving it too much thought. The code is acceptable to their standards if they can understand it and it solves the problem. Complications arise when the code is modified by someone else, and their time taken to understand the code becomes unnecessarily long due to unreadable code. Making changes, spotting bugs, and understanding the interaction with the rest of the code requires a complete understanding of the source code. [5].

One of the very basics that can increase code readability, is using meaningful naming conventions. According to Martin [12] the name of variables, functions and classes should tell the user why they exist, what they do and how they are used. If they require comments, then the given name is not descriptive enough.

Another topic that should be taken into consideration is functions. The author in 'Clean Code' [12] explains that keeping functions small, having them only do one thing, having the smallest possible amount of arguments given to a function and excluding all duplicate code can increase the readability of the code base tremendously.

Commenting can be divided into necessary and unnecessary commenting. The necessary types include comments, such as legal comments, informative comments that for example, explain regular expressions, clarification or explanation comments and TODO comments, which indicate changes that need to be made to the source code, but cannot be done currently due to some reason [12]. Comments become unnecessary when they share useless information, are outdated, redundant, poorly written or if they are commented-out code [12]. It can be difficult to distinguish between informative comments and pointless information. If the source code can be read by another developer without the need of additional comments, then there is no need to add them.

Additionally, the aesthetics of the code add to how readable the source code is. This means that using line breaks to create paragraphs for together fitting lines of code, keeping the same order consistently throughout (for example, the order of variables), aligning code into columns and using indentation can help with readability [5].

Lastly, the biggest factor affecting readability is the consistency of style used in a software project [14]. Using a consistent rule or following a set of coding standards is an efficient way to increase readability and quality control [4].

2.1.2 Maintainability

According to the ISO 25010 standard [9] maintainability is “the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment and in requirements”.

The effectiveness of performing maintenance on source code can be linked to multiple different factors. One of these factors is readability of the source code, since it allows other developers to work more efficiently with the source code. This subsection focuses on maintainability and its different attributes.

According to Ingeno [7], maintenance can be categorized into four different types found in the table below.

Table 1. *Types of maintenance*

Type of maintenance	**Description**
Corrective	Fixing defects in a software. The most common type of maintenance.
Perfective	Adding new requirements to the software. Altering the functionality.
Adaptive	Adapting the software system to changes in the software environment. For example, adapting the software to a new operating system.
Preventive	Improving quality of the software to prevent problems in the future.

All the maintenance types have in common the fact that they modify the program. Therefore, it is crucial that these modifications can be done easily to the program. Attributes enabling the ease of modification include: analyzability, how easily the location of the changeable or fixable element can be found, changeability, how much work needs to be done to the modification, stability, the amount of defects introduced after modification, and testability, being able to validate the modifications [14].

To conclude this section, a maintainable program contributes to all the factors that allow other developers to work on the source code with efficiency. Therefore, the maintainability of a program is linked to the quality of the source code. The higher the quality, the easier maintenance becomes.

2.1.3 Testability

In today's software development world, many companies have adopted the agile method to deliver software projects, allowing faster release of their newest versions. Since there is a continuous deployment of software with new features, an even bigger need has risen for a way to assure bug-free software. This can be achieved with testing.

Testing should be done on three different levels of abstraction. At the lowest level testing is done by using unit testing, which focuses on testing the program's modules. On the second level integration testing is used to ensure that the interaction between modules goes as planned. The highest level is system testing, which tests the compliance with the specified system requirements. [14]

Testing ensures that the modifications made to the code do not break other areas within the code. It also provides assurance that the modified code works as intended. It keeps the source code flexible, maintainable and reusable. With a range of automated tests to cover the production code it can be assured that the design and architecture stays clean. [12]

Similar to a source code, it is also important for tests to be written in a readable and clean way. Martin [12] suggests five rules how to have clean tests

1. **Fast** – Slow tests discourage frequent testing allowing for bugs to stay hidden. The longer they stay hidden the more difficult it becomes to fix them, causing the source code to decay.
2. **Independent** – A test should be able to run without the need for other tests. This allows the tests to be modular and specific modules can be tested separately. If tests rely on each other, the first failure causes everything to fail, making it difficult to spot the defect.
3. **Repeatable** – Tests should be able to run in any environment. If they end up being environment dependent, there might be situations where testing is unavailable due to being unable to use the needed environment. Being unable to run tests can keep the defects hidden.
4. **Self-Validating** – Keeping tests clear by a simple Boolean expression allows for simple understanding of the test result. It either passes or fails.
5. **Timely** – It is better to write tests before writing the production code.

Using these rules to write tests allows for frequent testing, catching independent bugs, easily understandable test results and encourages test-driven development (TDD). Testing also helps with maintainability and is later discussed in section 3.2 with how it can aid in refactoring as well.

2.2 The Importance of Code Quality

After understanding what code quality is, the real question for developers is whether it is important enough to be taken into consideration when writing code.

Section 2.1 explained three different attributes that contribute to code quality; readability, testability and maintainability. Readability minimizes the time to understand source code, allows for faster bug and faulty logic spotting and helps with software maintainability. Testability allows for clean architecture and design, allows for modifications to be made without breaking the software and also supports maintainability. Readability and testability are both important on their own, but their importance increases since they also support maintainability.

The cost and time for a software maintenance is said to be at least 70% [4] [10]. With so much time and money spent on maintenance, it is important that these resources are used efficiently. Maintaining a software is usually not done by the original author and the focus on code quality allows for developers to understand the source code faster and thoroughly [4]. If the quality is low, maintenance becomes time-consuming, forcing other tasks to pile up in the backlog and possibly cause delays in production. For a software company this situation is less than favorable and can cause stress in the relationship between the company and the customer.

Additionally, a natural part of a software project is the added complexity with every new version. Complexity usually increases with the addition of new requirements to the project; this increased complexity makes it harder to add other new requirements, which in turn causes delays. To some level, this rise in complexity is unavoidable, but it can help to keep the rise of the complexity as a constant. Having an easily maintainable source code allows for the constant rise of complexity. [2] This in return allows for better time management within the software project and decreases the chances of causing delays.

According to Catolino [3], source code quality affects the ratings of mobile applications. It is argued that inheritance and the complexity of a software increases bug-proneness, which in return causes crashes and unwanted behaviors in the application, which can leave the users dissatisfied. This study shows that quality is not only related to how effectively other software developers can work with the source code, but it also shows that there is a clear correlation with the satisfaction of users and the quality of the source code.

In conclusion, code quality is important because it can save time and costs for a software company. It can also increase the ratings of an application.

2.3 Code Quality Measurement Metrics

So far, this thesis has covered the meaning of code quality, what is meant by a readability, testability and maintainability of the source code, and why these are important to a software company. With this the question rises; how can code quality be measured? This section will focus on measurement metrics for each of the three quality attributes mentioned previously.

2.3.1 Measuring Readability and Testability

Readability and testability have simple ways for measuring their quality. Readability simply means how well other developers can read the programming code and testability how comprehensive the testing cases are. There are some metrics for both cases.

One way of measuring readability can be done during code reviews. Since readability depends on how well other developers can read the source code, having regular coding reviews on the project can aid in keeping the code readable.

Testability can be measured with a test coverage analysis. It measures the thoroughness of the testing and gives a percentage coverage answer in return. The measurement analyses how many lines of code get executed and thus shows if the tests cover every single angle of the code. This measurement can also be done for branch coverage, condition coverage and path coverage. [14]

2.3.2 Measuring Maintainability

There are a lot of different measurement metrics for maintainability. Additionally, maintainability is a very complex quality attribute and usually measurement metrics used to measure maintainability do so in a simplified way [14]. Since this thesis' main focus is not maintainability, this subsection only discusses one metric called the maintainability index (MI).

MI uses four different averages of individual measurement metrics to come up with the result. The result lies between 200 and -100, with the higher value indicating better maintainability. The four metrics are the Halstead complexity (HC) that measures computational density, cyclomatic complexity (CC) that measures logical complexity, lines of code (LOC) that measures code size and lastly, the percent of lines of comments which measures human insight. [14]

HC calculation relies on the occurrences of operands and operators within the program. This calculation uses numbers related to operators and operands, which then calculates the volume to show the program's execution size. This volume indicates how many bits

are needed to create the program. [1] However, this metric is often criticized and is categorized as controversial [14].

CC measures all the independent logical paths that can be accessed through the source code. In addition, it can also establish all the possible test cases that are needed to ensure 100 % coverage of all the possible outcomes of a source code. [14] Therefore, using CC can also help when creating tests for the source code.

LOC is a simple way of measuring the amount of lines within the code. Typically, the bigger the number, the more difficult maintenance is and the higher the complexity. However, using LOC on its own to compare different programs might not necessarily give an accurate representation of which one is more complex. For example, if the comparison was done between two programs which have a difference of only a couple thousands of LOC, it cannot necessarily be concluded which one is more complex. If the comparison is done between programs that have a difference of more than ten thousand LOC, then it can be said that the program with the larger LOC will more likely be more complex. [7]

Calculating the average lines of comments does exactly what it says it does. However, in subsection 2.1.1 it was mentioned that comments can be divided into necessary and unnecessary comments. Therefore, this metric seems to be a bit conflicted without measuring the actual quality of the comment as well. It can be alarming if there are no comments [14], but having modules with comments that take up 12 % of the module and not checking the quality can surely impact this metric and give an inaccurate result.

MI uses a lot of criticized metrics for its calculations, but as a whole forms a very comprehensive metric to measure maintainability. Nonetheless, having a single calculation to measure maintainability should not be completely trusted. In the very least, it should be calculated over time to have historic data to compare the measurement with [14]. MI also helps with project planning and estimation, and also works as a guide for refactoring [14]. With this MI can work as a tool for multiple purposes and help a company to manage their project in a more calculated way.

3 IMPROVING CODE QUALITY WITH REFACTORING

As mentioned in the previous chapter, a big role in code quality is played by how the code itself reads to another programmer. This can aid in the maintainability of the program, which has the possibility of causing tremendous cost to a software company. Improving the readability of a project can therefore be extremely useful. One way of making source code more readable is by refactoring the code. Refactoring is also supported with comprehensive testing. Therefore, understanding code quality through the quality attributes readability, testability and maintainability and being able to measure these to find the areas where improvements need to be made can all increase the benefit gained from refactoring.

3.1 Refactoring

Refactoring means changing the external structure to increase readability and allowing for cheaper modifications to be made without changing the behavior of the code [6]. This means that the focus is on classes, variables, methods and class hierarchy to allow future adaptions and extensions to be made as painlessly as possible [13].

3.1.1 The Benefits of Refactoring

According to Fowler [6], there are four reasons why refactoring should be used:

1. it can improve the design of the software,
2. makes it easier to understand the source code,
3. aids in finding bugs,
4. programming becomes faster.

It was mentioned earlier, that continuous changes to the source code will decrease the clarity of the project's architecture. By using refactoring on a regular basis, the architecture of the program can be maintained allowing for a clearer structure. This helps programmers to preserve the architecture and thus avoiding common mistakes, such as creating duplicate code in multiple places [6].

Reason two mentions how it is easier to understand the source code if refactoring is used to maintain the code. This could also be translated to making the code more readable to the programmer. For example, the first reason mentioned that it is important to maintain the design of the software to decrease the number of duplicates. The more duplicates in the code the more indecipherable the source code becomes.

Reason three greatly helps in corrective maintenance. There are two reasons as to why refactoring helps find bugs. Firstly, refactored code increases readability. This makes the code more comprehensible and allows easier spotting of faulty logic. Secondly, according to Fowler [6] refactoring code while finding a bug allows for deeper understanding of how the code should work and clarifies the structure of the program. This makes it easier to spot bugs.

The last reason can be viewed in a conflicted manner. When implementing refactoring, it can take a lot of time out of the programmer's work hours. This can lead to delays in production or maintenance. However, a program with a clear structure and architecture allows the developer to quickly find where changes need to be made. This decreases the development time dramatically in the long run. [6] This statement is also supported in a study, where software architects were asked about refactoring. Every interviewee stated that time spent on refactoring usually paid itself back in future development [11].

To conclude, refactoring has multiple benefits that can increase readability and allows ease of maintenance. Therefore, it can increase code quality and should be considered by software companies, if regular refactoring should be included in their programming routine. However, there are certain situations when implementing refactoring would be of most use. These are covered in the next subsection.

3.1.2 When to use Refactoring

In the previous subsection, it was discussed why exactly refactoring should be included in code development. Even if the benefits are worthwhile, there are certain situations when refactoring is highly valuable and at other times, refactoring should even be avoided. Fowler [6] gave an example of multiple different types of refactoring. They each have their own ideal timing when they should be incorporated. These types can be found in the table below.

Table 2. *Different types of refactoring*

Type of Refactoring	**Description**
Preparatory	Refactoring when adding a new feature or while fixing a bug.
Comprehension	Making code easier to understand by reading through the code and considering if the code can be written in a more readable way.
Litter-Pickup	A variation of comprehension, that allows for small quick changes, for example dividing bigger functions or renaming. Over time the code will become cleaner.
Opportunistic	Refactoring as a side product from programming. All three types above can be seen as opportunistic.

Fowler puts great importance in the fact that refactoring and programming go hand in hand. A programmer should not take time out of the day to randomly start refactoring the source code. Instead when the need arises to refactor, then is the most ideal time to do it. This is called opportunistic refactoring.

In section 2.3 one of the ways to measure code quality was during a code review. Code reviews are also a great time to consider refactoring [6]. Being able to have another set of professional opinions can allow refactoring to be implemented then and there. This ensures that the code can be as readable as possible. In addition, it is a great time to teach programmers with less experience, as it has been established that refactoring allows for a deeper understanding of the source code. However, it might be more useful to use code reviews for major refactoring operations rather than small ones [11].

As it was previously mentioned, there are also times when refactoring should be avoided. Fowler [6] mentioned two occasions when it should be avoided. Sometimes there is ugly code, but if the ugly code can be understandable, then it might be better to not refactor it. For example, when calling an API, there can be a lot of lines that set parameters and operate with the connection. These lines can be seen as scattered around and pushing the line character limit, which usually is considered ugly code. Even if the API call is written in this manner, a programmer understands the code. The second occasion is that

when rewriting is a better option than wasting time on refactoring.

Furthermore, according to Fowler [6] there can be more problems that need to be considered when refactoring. For example, the code might be owned by someone else making it difficult to refactor, merging multiple weeks old refactored branches to the mainline can cause multiple merge conflicts, making merging difficult and taking a lot of time. Additionally, refactoring is time consuming. If there are deadlines for new features, applying refactoring while adding the new feature can slow down production. It should therefore be considered if there is enough time to refactor.

Moreover, when working with customers it can be very difficult to convince them to put funds into refactoring. Since refactoring does not alter the functionality of the program, it can be difficult to show the customer how it is beneficial to them. [11] This can be avoided by working with customers that have a middle man who also understand the software world, but not every company has such a person.

In most cases, refactoring can be implemented without problems. There are some cases when refactoring however should be considered, and the benefits weighted against the problems it may cause. There is no real way of measuring when refactoring can be safely implemented, especially when racing against deadlines. Having a lot of experience and making small changes by refactoring the code while programming can help the programmer a lot.

3.2 Refactoring in Practice

Before taking a deeper look into how refactoring can be done in practice, some precautions need to be taken. To ensure that changes made to the code base do not break anything, a wide range of tests need to be available that check the made changes for errors. These tests should be self-checking and run on a regular basis after every change made to the software [6]. It can help if refactoring is done in small steps, allowing for any bugs to be found quickly with the support of tests [6].

This section follows the refactoring guidelines that are suggested by Fowler [6]. The upcoming subsections consist of different ways how to refactor, when a refactoring method should be applied, a step-by-step guide and then a picture of a simple code example. These pictures are lines of codes taken from a bigger program. Additionally, there are a variety of different refactoring methods, such as refactoring conditional logic, API's or simplifying inheritance. However, due to the scope of this study, five most useful ways of refactoring are presented.

Lastly, automated refactoring is mentioned, since a lot of powerful integrated development environments (IDE) have included it in their software application.

3.2.1 Extract Function

There are a handful of situations when function extraction should be done to the source code. Firstly, the length of the function is a good indicator when a function should be cut into smaller pieces. There are multiple opinions on when a function is too long. According to Fowler [6], a function should fit on the screen. Secondly, duplicate code lines should be extracted into their own functions. Lastly, if the function does more than one thing, the function should be divided into separate functions that all execute one separate thing. For small functions to improve code readability, a well thought out name is needed to indicate what the function does, not how it does it. A simple example can be seen in Picture 1. A simple guideline can be followed to successfully extract a function:

1. Create a new function and give it a meaningful name
2. Copy lines of code that need to be extracted and paste them into the new function
3. Look over the copied lines of code and see if there are any variables that are declared in the original function. If yes, pass them as parameters to the extracted function. If the variable is only used in the extracted function, move the variable declaration into the extracted function.
4. Compile
5. Replace the extracted lines of code with a function call
6. Test

7. Check if the extracted code is duplicated somewhere else and replace the code with a function call.

```
function printOwing(invoice){
    printBanner();
    let outstanding = calculateOutstanding();

    // print details
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
}

// In the above function the print details can
// be extracted
function printOwing(invoice){
    printBanner();
    let outstanding = calculateOutstanding();
    printDetails(outstanding);

    function printDetails(outstanding) {
        console.log(`name: ${invoice.customer}`);
        console.log(`amount: ${outstanding}`);
    }
}
```

Picture 1. *Function extraction*

The original function executes two different tasks. In the refactored version the details printing is extracted from the original function and put into its own nested function called printDetails(). Since the function uses the parameter that was declared in the original function, the parameter needs to be passed to the extracted function.

Similarly to function extraction, variable extraction can be done. Sometimes expressions can form complex logic. Cutting this complex logic into local variables can aid in clarifying the logic by naming it. Additionally, this also aids in debugging, since debugging allows to look into what value a variable holds.

3.2.2 Inline Function

Inline function is the opposite of extract function. There are situations where a short function's content can be as clear as the function's name. Extracting the code body into its own function does not add any additional clarity to the code, meaning it should be left alone. If there are functions with unnecessary delegations, then inline function can be used.

To successfully inline a function a simple guideline can be followed:

1. Check that the function is not a polymorphic method.

2. Find all the callers of the function
3. Replace calls with function body
4. Test after each replacement
5. Remove the function

```
function getRating(driver) {
  return moreThanFiveLateDeliveries(driver) ? 2 : 1;
}
function moreThanFiveLateDeliveries(driver) {
  return driver.numberOfLateDeliveries > 5;
}


// The two functions above can be refactored into one function
function getRating(driver) {
    return driver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

Picture 2. *Function inline*

Picture 2 shows how inline function can be used efficiently. The refactored function allows for less code lines, which can save time when reading through the code. A problem might rise, when the return value becomes too long, in which case a different refactoring actions might be more suitable.

There is also a refactoring method for inline variable. This works similarly to inline function, where variable names do not add any additional information to the expression, thus it is better to leave it alone or inline it.

3.2.3 Rename Function

A function should have a name that is descriptive enough to let the programmer know what its purpose is without having to read the function's implemented code. If the name is not clear, then it is advisable to rename the function. Renaming can be done by writing a comment describing what the function does and from the comment rewrite a fitting name to the function.

If the declaration and all the function callers can be changed easily, then renaming the function, and finding all the references and renaming those should not be problematic. After the renaming is complete, testing should be done.

However, there are cases when renaming can be difficult. For example, when the function has many callers, accessing these callers is difficult or in general the declaration is complicated to change. In this case a migrative type of guideline can be followed:

1. Use extract function to create a new function. Give the new function a temporary name

2. Let callers call the new function. This should be done in an iterative way and not change all callers at the same time.
3. Test
4. Apply inline function to the old function
5. Restore the temporary name with the original name
6. Test

```
function circum(r) {
    return 2 * Math.PI * r;
}

// renaming the function and parameter
function circumference(radius) {
    return 2 * Math.PI * radius;
}
```

Picture 3. *Renaming unclear functions and variables*

Picture 3 shows a very common way of naming functions. Shortening longer words, such as circumference to circum is unnecessary. It is advisable to use the full word instead. This will not leave the programmer in a guessing state if circum really means circumference. Similarly, parameters should not be a single word, or an abbreviation of something, such as r for radius.

3.2.4 Encapsulate Variable

Having raw data in a program is not ideal and can cause problems when wanting to access the data from different places or if the data needs to be moved around. If there is a raw data variable and it is not a temporary variable, then encapsulation should be considered. Encapsulation means that the information that the variables shares is limited or even hidden [6]. This can be achieved by routing the access of the data through functions. By encapsulating the variable, it can be moved around more freely, and modifications can be made with ease.

Encapsulating a variable can be done with the following steps:

1. Create functions that access and update the variable.
2. Run static checks.
3. Find where the variable is used and replace the usage with the appropriate encapsulated function. Test after each replacement.
4. Restrict the visibility of the variable
5. Test

```
// global variable
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};

// referencing the global variable
spaceship.owner = defaultOwner;

// updating the global variable
defaultOwner = {firstName: "Rebecca", lastName: "Parsons"};


// instead the variable can be encapsulated
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function defaultOwner()    {return defaultOwnerData;}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}

// now referencing and updating can be done as follows
spaceship.owner = defaultOwner();
setDefaultOwner({firstName: "Rebecca", lastName: "Parsons"});
```

Picture 4. *Encapsulation of a global variable*

Picture 4 shows an example of a global variable defaultOwner that can be encapsulated. The refactored code provides a clear way to access and change the data. In addition, adding any form of validation or conditional logic to the updates is easier now. However, there might be a problem if the code wants to change shared data, since changes made by clients are not reflected in the shared data. This can be avoided by encapsulating raw data into a class instead, which is covered in 3.2.5.

3.2.5 Combine Functions into a Class

If there are functions that work together with a common data structure, then these functions can be refactored to form a class. A class shows clearly how these functions use the data, removes unnecessary arguments, and allows passing the object to other parts of the system. This is more ideal in object orientated languages, such as JavaScript or Python. However, refactoring functions into a class is not simple and requires encapsulation.

When there is a set of functions that can be refactored into a class, the first step is to encapsulate the common data structure used by the functions. This can be done by following these guidelines:

1. Encapsulate the variable that holds the data structure, with the method used in 3.2.4
2. Replace the content with a class that wraps the data structure. Create an accessor inside the class that fetches the raw data.

3. Test
4. Create a function that returns an object of the class. These include functions, such as getters and setters that set data to an object and retrieve data of an object
5. Replace the original code that uses the data structure to use the function that calls the object instead. If the data structure is complex, then make the changes in small steps testing after every change.
6. Remove the raw data accessor.
7. Test

Picture 5 shows an example of how a data structure can be encapsulated by following steps 1-7. Creating a class creates more LOC, but it allows the programmer to access the data from anywhere in the system if the class Reading is imported. Using classes allows for clear design and adds to readability, as can be seen in the Picture 5.

```
1  // Data Structure
2  reading = {customer: "frank", quantity: 10, month: 5, year: 2017};
3
4  // The above data structure can be encapsulated
5  // into a class called Reading
6  class Reading {
7    constructor(data) {
8      this._customer = data.customer;
9      this._quantity = data.quantity;
10     this._month = data.month;
11     this._year = data.year;
12   }
13   get customer() {return this._customer;}
14   get quantity() {return this._quantity;}
15   get month()    {return this._month;}
16   get year()     {return this._year;}
17 }
```

Picture 5. *Encapsulation of a raw data structure*

Once the encapsulation has been completed, all the functions that use the data structure should be moved inside the new class. Additionally, functions that manipulated the data structure should be extracted with the extract function method mentioned in subsection 3.2.1 and relocated into the new class.

```
// a set of variables that use the data structure values
// to calculate base charge and tax charge
const aReading = acquireReading();
const base = (baseRate(aReading.month, aReading.year) * aReading.quantity);
const taxableCharge = Math.max(0, base - taxThreshold(aReading.year));


// instead methods can be created inside the Reading class
// that calculate the base charge and tax charge
get baseCharge() {
  return baseRate(this.month, this.year) * this.quantity;
}

get taxableCharge() {
  return  Math.max(0, this.baseCharge - taxThreshold(this.year));
}


// with the methods inside the class, these methods
// can be called once the object has been created with 'new'
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const baseCharge = aReading.baseCharge;
const taxableCharge = aReading.taxableCharge;
```

Picture 6. *Moving functions inside the class Reading*

Picture 6 shows how functions can be moved inside the new class called Reading. This allows for all the logic that uses the data structure to be placed inside the class. For a programmer this helps greatly in writing more code that uses the class Reading and if more features need to be added, they can be added with ease.

3.2.6 Automated Refactoring

A variety of different IDEs support automated refactoring. There is usually some level of refactoring capabilities available in any editors, but they vary immensely on the level of support they offer. [6] However, a powerful IDE, such as IntelliJ IDEA (IntelliJ) by JetBrains or Eclipse can offer a lot of support in all the previously mentioned refactoring methods. IntelliJ supports refactoring for the three most common types: renaming, extracting and deleting. [8]

When renaming a function or variable the common search-and-replace tool can be used. This can be error-prone, since some editors use text manipulation to achieve this [6], meaning items that use the same word in their name will unintentionally be changed as well [8]. With a powerful IDE such as IntelliJ, this error can be minimized, since the tool operates on the syntax tree of the code. IntelliJ's renaming refactoring offers a variety of different tools to support renaming. These tools have the following functionalities [8]:

- suggests a list of possible replacement names based on the class name and other aspects,
- methods that use the renamed elements' field are renamed accordingly
- rename all the calls for the methods and overridden/implemented methods in sub-classes
- renames non-code uses of the name
- allows for a preview of what the user wants to rename

Similar tools are offered when extracting or deleting elements. With this level of support on refactoring, a software company should make it a priority to use a powerful IDE, such as IntelliJ, to make refactoring easier and more attractive to use. Another suggestion would be to encourage the use of automated refactoring and have a set of guidelines available how to use automated refactoring with the IDE they use for their work.

4 CONCLUSION

This thesis aimed to identify code quality and the importance of it to a software company. In addition, it examined how code quality can be managed by using refactoring. A link can be found between the three quality attributes mentioned in Chapter 2. Maintainability seems to be one of the most important quality attributes, since it has an influence on a software company's time and budget. Maintainability is highly affected by how readable the source code is and by the coverage of tests implemented in the software.

A company can implement measurement metrics on these quality attributes to see if their source code quality should be improved. If needed, refactoring can be used to improve maintainability and readability, while testability is used to back up refactoring. Since it is natural for the quality of the source code to decay over time, it is advisable to implement opportunistic refactoring whenever possible. There are a variety of different advantages to refactoring, but during certain situations refactoring can prove to be more problematic than beneficial. It is important for a company to weigh the advantage against the disadvantage.

Further research is needed to determine the actual impact of refactoring on code quality. Refactoring could be applied on a company's software project by taking quality measurement metrics pre- and post-refactoring to show its impact. This could prove to a software company if they truly need to use refactoring to improve their source code quality.

REFERENCES

[1] M. Alfadel, A. Kobilica and J. Hassine. Evaluation of Halstead and Cyclomatic Complexity Metrics in Measuring Defect Density. (2017), 1–9. DOI: 10.1109/IEEEGCC.2017.8447959.

[2] A. Axelrod. *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*. Apress, 2018.

[3] G. Catolino. Does Source Code Quality Reflect the Ratings of Apps?: *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)* (2018), 43–44.

[4] X. Fang. Using a coding standard to improve program quality. (2001), 73–78. DOI: 10.1109/APAQS.2001.990004.

[5] T. Foucher and D. Boswell. *The Art of Readable Code*. O'Reilly Media, Inc, 2011.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. 2nd ed. Addison-Wesley, 2019.

[7] J. Ingeno. *Software Architect's Handbook*. Packt Publishing, 2018.

[8] *Introduction to refactoring*. URL: https://www.jetbrains.com/help/idea/tutorial-introduction-to-refactoring.html# (visited on 04/14/2020).

[9] *ISO/IEC 25010*. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/57-maintainability (visited on 03/16/2020).

[10] P. Jain, A. Sharma and L. Ahuja. The Model for Determining Weight Coefficients of Maintainability Criteria in Agile Software Development Process. (2019), 1–4. DOI: 10.1109/IoT-SIU.2019.8777609.

[11] M. Leppänen, S. Mäkinen, S. Lahtinen, O. Sievi-Korte, A.-P. Tuovinen and T. Männistö. Refactoring-a Shot in the Dark?: *IEEE Software* 32.6 (2015), 62–70.

[12] R. C. Martin. *Clean Code: Handbook of Agile Software Craftsmanship*. Prentice Hall, 2015.

[13] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30.2 (2004), 126–139.

[14] D. Spinellis. *Code Quality*. Addison-Wesley Professional, 2006.