UiO : Department of Informatics
University of Oslo

Effects of Clean Code on Understandability

An Experiment and Analysis

Henning Grimeland Koller

Master's Thesis Spring 2016

Abstract

Having understandable and readable code is important in the software development world as small changes can be made difficult and complex by poor code. Clean Code is a set of rules and principles meant to keep code understandable and maintainable, proposed by software engineer Robert C. Martin.

This thesis studied the effect of Clean Code on understandability. After consulting previous research on refactoring and the effect on understandability, we identified a need for a measurement other than calculated code quality metrics. An experiment was conducted where the time used to solve coding tasks was used as a measurement for understandability. For the experiment a system with characteristics of legacy code was refactored to be more "clean". Two groups of participants were asked to solve three small coding tasks on this system. An experimental group solved assignments on the "clean" code, and a control group on the legacy code.

We expected the Clean Code version to be more understandable, and that the experimental group would spend less time solving the tasks. Contrary to our expectations, two of the three assignments were solved faster by the control group. These results indicate that Clean Code does not have an immediate effect on understandability. However, we observed that developers working with Clean Code wrote higher quality code.

Acknowledgements

I proudly present my master thesis on the effect of Clean Code on understandability. It was written at the Department of Informatics, University of Oslo during the years 2015/2016. After two years of work I would like to thank the persons who made it all possible.

Firstly, I would like to thank my supervisors, Sigmund Marius Nilssen and Yngve Lindsjørn for their great help and advise. Especially Sigmund for his thorough and constructive feedback from start to end. Secondly, all of my family for their encouragement and kind words, in particular, my brother Martin for his great interest and our long discussions. Also, thank you Tom Erik for reading my thesis and giving me valuable feedback.

To all of my good friends, thank you for five incredibly fun and memorable years at Assembler. And last but not least my girlfriend Vilde, thank you for all your love and support.

Contents

List of Figures

Listings

Chapter 1

Introduction

Imagine a desk. The desk is messy; there are papers floating around and no structure in the documents whatsoever. Finishing the work started on this desk could prove to be difficult. To do so one would have to go through the mess, and try to make sense of it before even starting on the actual work. This could take hours, while the actual work only takes a couple of minutes.

This is a very common scenario in the world of a software developer. Short, easy tasks are made complicated by untidy and "dirty" code. The developer has to spend time understanding this code before he can confidently make changes. Or even worse, if he is in a hurry he may be tempted to create a piece of code that further increases the mess.

Back to the earlier example. What if the desk was uncluttered and neat, with papers organized and stored systematically in folders? This would make the assignment much easier. Now one could quickly pick up on the former worker's trail of thought and understand what needs to be done. The same goes for software development. Writing structured and tidy code makes it much easier for both the writer and the next developer to work with the code later on. But writing good and structured code is not that easy.

Robert C. Martin talks about "code-sense" in his book Clean Code[17, p,7]. Code-sense is not only the ability to look at a piece of code and immediately tell if it is good or bad, it is also the ability to see what can be done to make the code good. According to Martin, some are born with it, while others must work hard to achieve it[17, p,7]. But even with good code-sense we might end up writing bad code.

One of the largest threats to good code is time constraints. Whenever a programmer is in a rush he might be tempted to write bad code just to finish on time. "It is just a small piece of code, I can clean this up later". But he just "broke a window" in the project: Andrew Hunt and David Thomas discuss the "Broken Window Theory" in The Pragmatic Programmer[12, p,5]. A house with broken windows gives the impression that nobody cares about it. So others also stop caring. Sooner or later the house will be completely destroyed. The poor piece of code the programmer created is such a broken window. It can create the feeling that nobody cares about the code, and the next programmer working with it might not bother either.

Martin Fowler tells a story of another danger to good code in the preface of his book Refactoring: Improving the Design of Existing Code[11]. Even though programmers know about problematic code in their project, the management does not allow them to spend time on cleaning it up because it will not add to the features of the system. The customer cannot see the code, so it does not matter if it is troublesome. This way of thinking is why bad code is often tolerated; it may look like it is more profitable to spend time on adding new features instead of halting production and cleaning up. But in the long run, as in Fowler's story, the poor code can bring the whole project to its knee due to how difficult it is to work with. Several practices have been proposed to cope with these challenges. Clean Code by Robert C. Martin is arguably one of the most renowned.

1.1 Research Question

There has been conducted studies about how to measure or evaluate code quality[4, 5]. These studies establish key-points in code quality and a set of metrics used to measure the quality of software. They also discuss how to use these metrics to do measurements. What they do not discuss in much detail is how to achieve a higher code quality. To find out what might increase quality, we read studies on refactoring and its effect on software quality[2, 9, 14, 19, 21, 22].These studies conclude differently about the effects of refactoring on the quality of the code.

After reading these studies we wanted to focus more on understandability. We read studies about software understandability and how to measure it[8, 15, 16, 20]. These studies propose different models and methods for measuring and evaluating the understandability of software, and of course what software understandability really is. But when we tried to find studies on how to improve software understandability there was a lack of studies. So we decided to study Clean Code and its effect on understandability.

We wanted to study the effect of Clean Code because it is a widely accepted method of writing good maintainable code. Since we could not find any studies on the actual effect of Clean Code we decided to use Clean Code as our guide to write understandable code, and see if it does improve the understandability of code.

All of the previously mentioned studies use calculation to determine if refactoring has a positive effect on code quality or productivity. Even though these studies include understandability, our opinion is that understandability cannot be measured the same way as code quality. We agree that a low complexity and few lines of code could indicate that the code is understandable, but it might not be the case. The understandability of code is what enables developers to comprehend its meaning.

Therefore, we decided to use time as a measurement of understandability. The reason is that we want to find empirical evidence by observing participants understand and solve small coding tasks. The argument we want to make is that understandable code requires less time to become familiar with, and coding can begin earlier. Hence, less time is used solving the assignments. Clean Code also comprises rules and principles that are less calculable: how to make code descriptive and informative to the programmer.

We focus on small coding tasks as it is easier to determine the understandability of the code with smaller tasks. The reason being that with greater coding tasks more time is used planning and implementing as one are making great changes in the system. In small tasks on the other hand, time is mostly spent on understanding the system and the surrounding parts dependent on the change, instead of implementing.

- **This leads to our main research question:**
  Will Clean Code improve the time used to solve small coding tasks?

To determine whether small coding tasks are solved faster on clean code, we first need to narrow our question to specific coding tasks. We decided to focus on three of the most commonly tasks performed by developers: Implementing new significant functionality, changing current functionality and bug fixing. This gives us three sub-research questions:

- **Sub-research question 1:**
  Will Clean Code improve the time used to implement new significant functionality?

- **Sub-research question 2:**
  Will Clean Code improve the time used to change current functionality?

- **Sub-research question 3:**
  Will Clean Code improve the time used to find and solve a bug?

Only after answering the sub-research questions do we have enough knowledge to answer our main research question.

To answer our research questions we conducted a quantitative, empirical experiment. Volunteer participants were placed in two groups and asked to solve three different small assignments on the system. One group worked on the original version of the system, while the other worked on a refactored version following the principles of Clean Code. The assignments are designed to answer our sub-research questions, and focus on one question each. Our research method is discussed more in detail in chapter 3.

1.2 Outline

Chapter 2 first gives an overview of understandability in the software development world, before it describes Clean Code and code quality. It finishes with discussing the effect of Clean Code on code quality and related work.

Our research method is presented in chapter 3. Here we also discuss the implementation of our experiment, the assignments and the participants.

Next we present the system used in the experiment and the refactoring. In section 4.1 we give an introduction to the system, and we identify different problems with the code and justify why it is a fitting system to use. In section 4.2 we discuss the refactoring done to the system to make it more clean.

In chapter 5, the results from the experiment are presented.

The results are discussed and analyzed in chapter 6. Limitations with the experiment are also identified. After discussing the results we conclude our research in chapter 7 and list our thoughts for future work.

Chapter 2

Background

2.1 Understandability

We want to examine the effects that Clean Code can have on understandability. However, we first need to establish what understandability means in the field of software development. Boehm defines understandability the following way: *"Code possesses the characteristic understandability to the extent that its purpose is clear to the inspector[5]."* He goes on to say that this implies that variable names are used consistently and modules are self-descriptive. This description has a lot in common with the ideas Robert C. Martin discusses in Clean Code. We discuss this more in detail in section 2.2.

Understandability is part of a system's maintainability[5] because a developer needs to understand the system in order to be able to maintain it. This makes understandability crucial when it comes to software development. If a system is not understandable, maintaining it will prove difficult for new developers. One could say that the original developers have not properly communicated the purpose of the code if the system is hard to understand. If we think of it that way, source code can be a thought of as a form of communication between developers[20]. So, for a system to be maintainable and evolvable, developers need to communicate with each other through the code. As with any other kind of communication, code can be misunderstood or misinterpreted, and if a new developer misunderstands code there is a higher chance that errors are introduced.

Several metrics and models to measure understandability have been proposed[15], [16]. These metrics focus on the size of the system, number of operands and operators or the distance between usage of variables:

1. Code spatial complexity(CSC) is a measurement of the effort needed to understand the logic in a module. CSC is defined as the average distance, in

terms of lines of code, between the definition and use of modules[8]. The higher the number, the more cognitive effort is required to understand.

2. Data spatial complexity is defined as the average distance between the usages of a variable, and is a measurement of the effort needed to understand the use of a variable[8]. As with CSC, a higher number indicates more cognitive effort needed to understand.

What can we do to make sure our code is understandable? Understandability is a very subjective matter which makes it hard to come up with a good answer. Robert C. Martin proposes his answers to this difficult question in his book Clean Code: A Handbook of Agile Software Craftsmanship. In section 2.2 we discuss his ideas and rules which, in his opinion, create better and more understandable code.

2.2 Clean Code

> *"Clean code is simple and direct. Clean code reads like a well-written prose."*
> -Grady Booch[17, p,8]

In 2008 Robert C. Martin published the book *Clean Code: A Handbook of Agile Software Craftsmanship*[17]. It describes the principles, patterns, and practices of writing Clean Code. Clean Code in a nutshell is as Grady Booch describes it: Simple, elegant, direct and reads like a story/article[17, p,8].

> *"You know you are working on clean code when each routine you read turns out to be pretty much what you expected."*
> - Ward Cunningham[17, p,11]

Understandability is a major part of Clean Code. The code should clearly show what it is trying to do, and not hide anything from the reader. There should be no surprises. Martin suggests several rules throughout the book, for how clean code should be written. He touches upon everything from classes to comments. Some of the most important rules mentioned in Clean Code are:

- **Meaningful Names** [17, p,17]
  Names are incredibly important when writing code. We name everything, and it helps describe what is going on in the code. A good name states the intention of the variable, class or function. It should also be clear from the names what the context of the code is. The names are what gives the code meaning.

- **Functions should do one thing** [17, p,35]
  A function should do only one thing. It should do what the name states, and nothing more. If a function is doing multiple things it gets confusing very fast, and the potential for side effects increases. Side effects are dangerous and can lead to unwanted incidents. Every statement in the function should be nothing more than one abstraction level below the name of the function.

- **Functions should be small** [17, p,34]
  Functions should be as small as possible. This way one is forced to further abstract the functions, which in turn strengthens the previous rule.

- **Don't repeat yourself** [17, p,48]
  Never repeat code. This indicates a lack of abstraction. With duplication bugs are more likely to be introduced. Whenever a change is needed, the duplication forces the change to be implemented in multiple areas. If an area is forgotten, it could lead to errors and bugs. Extracting the duplicated code to a single function removes this problem as the change is only needed in one place.

- **Single Responsibility Principle** [17, p,138]
  The Single Responsibility Principle, as the name implies, means that a class/module should only have one responsibility. One could also say that a class should only have one reason to change.

Martin discusses other rules, but we will focus on the ones in the list above. Code following these rules is what we will refer to as being *clean* for the rest of this thesis.

These rules are simple and yet one might not have thought about them earlier. They are all trying to make the code more understandable and easier to work with. In "Toward a Measurement Framework for Example Program Quality"[7] Börstler et al. compare two implementations of a calendar program to investigate desirable properties of programming examples for students. They argue that "a good example must effectively communicate the concept to be taught". This is similar to what Martin is saying about code clearly showing its intent[17, p, 55]. Börstler et al. named one program the Beauty and the other the Beast. They find The Beauty beautiful for several reasons very similar to the rules we just listed:

1.*"The key concepts in the problem domain are modelled as separate classes"[7].* Instead of having one large class, they have created several classes each with a single responsibility. The Month class has only one responsibility, and that is to know how many days there are in each month. The implementation of classes are following the Single Responsibility Principle.

2.*"The implementations are very simple and small"[7].* The functions and the classes in The Beauty are small and they do only one thing. This is similar to Martin's rules about functions.

3.*"Carefully chosen identifier names enhance the readability"[7].* The names of the variables, functions and classes in The Beauty describe very well why they are there and their functionality, very much like Martin's rule about meaningful names.

4.*"The decomposition supports independent development and testing"[7].* When modeling all the key concepts as separate classes, each individual part can be tested more easily.

The Beast, on the other hand, is the complete opposite. It is mainly one long function where all the logic is done. This long function is also highly nested, making it even more complex. As Börstler points out, this approach makes it difficult to include identifiers to help with the comprehension. There are similarities between Börstlers reasons for why the Beauty is beautiful, why The Beast is a beast and what Martin describes as "clean code" and "bad code".

2.2.1 Clean Tests

Börstler speaks of the benefit of being able to test each of the classes separately in The Beauty. Martin discusses the same, only he refers to the Single Responsibility Principle. Testing is not something exclusive to Clean Code, but it is nevertheless an important topic. Unit tests are what makes the code safe to change. *"Test code is just as important as production code. It requires thought, design and care. It must be kept as clean as production code."*[17, p, 124]

Unit tests should evolve with the system, and if the tests are dirty it will be hard to keep them up to date with the system. To keep unit tests clean they need to be precise, simple and readable. Much like the code they are testing. Many of the same rules applies to unit tests to achieve this, but Martin adds to these with additional rules.

Single Concept Rule

For a test to be precise it should focus on one thing. When a test contains many assertions it starts to get confusing and difficult to see what the test is actually testing. The problem is not necessarily the amount of asserts but the number of concepts being tested. With several concepts in the same test, the reader has to use more effort to understand what each is testing[17, p,131]. Therefore Martin proposes the "Single Concept Rule": Minimize the number of asserts and only one concept per test. The rule forces tests to be written as short and understandable as possible.

F.I.R.S.T

The acronym F.I.R.S.T is another five different rules clean tests should follow[17, p,132]:

- **Fast**
  Tests should be fast. If a test is slow one might be reluctant to run it after a change. The faster a test runs, the more frequently it will be used. One makes a change and run the test. Failing to have fast tests could lead to errors not being discovered at an early stage, and repairing them might be more costly when they finally are discovered.

- **Independent**
  A test should only depend on itself. It should be able to be run independently and not rely on other tests being run for it to work. If tests are dependent on each other only one test needs to break to break the others. This creates a series of errors making debugging harder.

- **Repeatable**
  A test should be able to run repeatably anywhere and whenever. It should not depend on the environment or any external services. This means that it should run without the Internet or outside the development environment. Without this, uncertainty is introduced as one cannot know if the test fails or passes because of other factors than wrong code.

- **Self-Validating**
  For a test to be self-validating it needs to have a boolean output, pass or fail. If a test is not self-validating it is hard to make it either of the above. If one has to compare two files to see if the test passes, it is not very fast or independent.

- **Timely**
  A test should be written before the code it is supposed to test. This way the code must be designed in a way that makes it testable. Otherwise one might find the code hard to test because it is not designed to be tested, and decide to not test it at all.

The most important concept with tests, as Martin stresses several times, is that they provide safety when changing the code[17, p,124]. If the tests are dirty and hard to change, safety will be lost as the tests cannot evolve with the system. Both F.I.R.S.T and the Single Concept Rule aim to help with this problem.

2.3 Refactoring

As a system grows and ages it might need to be cleaned up. This cleaning is what we call refactoring. Martin Fowler defines refactoring the following way:

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure"[11, p, xvi].

In his book Refactoring: Improving the Design of Existing Code[11], Fowler presents several techniques to improve the internal structure of software systems. These techniques can be used to reduce redundancy, divide responsibility to reduce unit size or to make the code more readable.

What is important to note from Fowler's definition is that the external behavior should not be altered. In other words, refactoring does not give value to the customer immediately. Studies do however show that it can lead to improved code quality[9, 19, 22], which in turn can improve the development process and allow faster releases.

2.4 Code Quality

ISO/IEC 25010[13] is the international standard for evaluation of software quality. The standard defines the quality of a system in the following way:
"The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value".
"Needs" refers to performance, maintainability, security, etc[13].

So far we have discussed several principles and techniques that are supposed to make code more understandable and readable. Understandability and readability are both very subjective, which makes it hard to determine if implemented changes for understandability increases code quality. There are other, more objective, metrics which give an understanding of aspects of code quality other than understandability

We will focus on the metrics used to determine software maintainability[4]: *Maintainability represents 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*[13].

Baggen et al. supports this definition by claiming that: "when a change is needed in software, the quality of the software has an impact on how easy it is[4]":

1) To determine where and how that change can be performed.
2) To implement the change.
3) To avoid unexpected effects of the change.
4) To validate the changes performed.

The Software Improvement Group uses 6 different properties to calculate the maintainability quality in code[4]:

- **Volume**
  As the system grows it gets less maintainable due to the size and amount of information in the system. A possible metric is lines of code.

- **Redundancy**
  It takes more time to make a change in redundant code than non-redundant code. A possible metric is percentage of duplicated code.

- **Unit size**
  The smaller the size of the lowest-level units the easier it is to understand them. Lines of code per unit can be used as a metric.

- **Complexity**
  A system with high complexity is hard to maintain. Cyclomatic complexity is a good metric.

- **Unit interface size**
  A unit with many parameters indicate bad encapsulation. Number of parameters per unit can be used as a metric.

- **Coupling**
  Tightly coupled components introduces difficulty when change is needed. Number of calls incoming per component can be used as a metric.

These scores are important for developers because they give an indication of the quality of their software, and if it possesses desirable characteristics. Developers can use them to spot problems in the code before it grows too large and solving the problem becomes too expensive or difficult to handle. A problem that could occur with this approach is that undesirable characteristics might be discovered late in the development process.

To calculate these metrics accurately the development needs to be at a certain level of maturity, and at this point rewriting or refactoring is not necessarily something one would want to spend time on. The worst case would be discovering that the system needs a full rewrite due to high redundancy or coupling scores. To avoid this problem guidelines should be followed when writing code. Clean Code offers a guideline to write understandable code, but also code with increased quality as we will discuss in the following section.

2.5 The Effect of Clean Code on Code Quality

Clean Code is a book about the craft of programming and how one can become a better programmer, and if we look closer we can see that Martin's rules not only aim to make code more understandable but they also increase the quality. This means that by writing clean code the quality of the software should increase. Let us look closer at some of the properties and how Martin's rules help to achieve a good quality score:

2.5.1 Redundancy

Don't repeat yourself is one of the rules we discussed in section 2.2, and following it will lead to code with less redundancy. This rule addresses the property redundancy directly. One could say that the property measures how good of a job one has done following Martin's rule about duplication.

2.5.2 Unit Size

There is a clear correlation between the property unit size and Martin's rule *functions should be small*. The more the rule is followed, the smaller the average function size. Martin even argues that a function should be no longer than a couple of lines[17, p, 34].

2.5.3 Complexity

When measuring complexity, cyclomatic complexity is a very common metric.[4]. Cyclomatic complexity was developed by Thomas J. McCabe[18] and it is a quantitative measure of the number of linear independent paths through a control flow graph of a program. To calculate a function's cyclomatic complexity we use the following formula:

$$M = E - N + 2$$

E is the number of edges in the graph and N is the number of nodes. Nodes translate to decision points, such as if statements. If statements with multiple predicates count as one for each predicate. McCabe also shows that cyclomatic complexity in programs with one entry point and one exit point is equal to the number of decision points plus one[18]. For functions with multiple exit points it can be calculated with the following formula:

$$M = D - S + 2$$

D is the number of decision points and S is number of exit points.

Another advantage of calculating the cyclomatic complexity of a function is that it gives the necessary test cases to achieve full test coverage. The score gives the upper-bound to achieve full branch-coverage, and the lower-bound to achieve full path-coverage of the function.

Martin does not discuss how to avoid a high cyclomatic complexity, but as we shall see, following his rules leads to code with better complexity spread. Most important are the rules *functions should do one thing* and *functions should be small*. If a function has a high cyclomatic complexity it may be a sign that it is trying to do more than one thing, its size also naturally grows as a side effect. So by having a high cyclomatic complexity, two of Martin's rules are violated and the quality of the code is reduced. But if the decision points are extracted into separate smaller functions the complexity will spread across several functions, each with a low complexity. The overall complexity of the system is not reduced, but dealing with smaller less complex functions is better than dealing with large highly complex functions. This way Martin's rules are followed and the quality of the software is increased.

Martin's rule *meaningful names* is aimed at giving functions meaningful and descriptive names. If a function has many decision points it may prove difficult to describe all this in a name.

2.5.4 Volume

Clean Code does not help reduce the volume of a system, but if the rules are followed the size will not be as much of a threat to maintainability. The volume of the system might grow due to Clean Code, but the rules we discussed in section 2.2 are rules meant to help write code that is understandable and easy to work with. So while the volume increases the maintainability also increases.

2.5.5 Unit Interface Size

Martin discourages functions with more than two parameters[17, p, 40]. He prefers no parameters, and following this gives a good score on the property unit interface size.

2.5.6 Coupling

To achieve decoupled code, Martin talks about the Dependency Inversion Principle(DIP): "In essence the DIP says that our classes should depend upon abstractions, not on concrete details"[17, p, 150]. What this means is that high-level modules should not depend directly on low-level modules, there should be a level of abstraction in between that the high-level modules depend on instead. The low-level modules are then based on this abstraction layer. This way the modules are isolated from each other, and changes in the low-level module wont affect the high-level module. We achieve less coupling and a more flexible and testable system when following the DPI.

Now we have drawn similarities between Clean Code, which can be viewed as a subjective good way to write code, and the objective metrics for maintainability quality. This suggests that Clean Code is not just subjective but it also contributes to increase the objective quality.

The four points made by Baggen et al.[4] can also be compared to Clean Code. We can summarize the points into one sentence: *software with high quality is easy to work with and should not be surprising*. This is how Martin describes Clean Code.

2.6 Related Work

Now that we have established the necessary background knowledge, we can present and discuss some of the available related work. In particular, we want to look at research into understandability and the effect of refactoring, as this is closely related to what we have researched. Research on refactoring cover several different areas, such as how to refactor[11, 17], the effect on code quality, productivity etc. We did refactoring to increase understandability and quality, but we strictly followed Clean Code when doing so. Therefore it is very interesting to see what other found and concluded, when they have studied the effect of refactoring.

2.6.1 The Effect of Refactoring on Code Quality

There have been several studies on refactoring and its effect on code quality, but the results differ. Some conclude that refactoring improves code quality[9, 19, 22], while others conclude the opposite[2, 14, 21]. Despite differing results, all of the studies are carried out in a very similar manner.

First, they establish the refactoring techniques to be used. The techniques proposed by Martin Fowler in his book "Refactoring: Improving the Design of Existing Code"[11] is mainly used. For example: "Extract Method"[11, p,110], "Extract Superclass"[11, p,336], "Move Method"[11, p,142], "Replace Data Value with Object"[11, p,175].

Secondly, they calculate the quality score of one or more systems before refactoring and then calculating again. We discussed some of the metrics used to give the systems a score in section 2.4. Finally, the two scores are compared.

Why these very similar studies come to different conclusions is difficult to say. The original state of the systems they refactor could be a factor, or how skilfully the refactoring is done. But it shows that refactoring might not always increase code quality.

A common problem with these studies is the quantitative side. They refactor few systems, sometimes just one, and decide whether refactoring benefited code quality. The conclusions are also based solely on the calculated quality score. We argued in section 1.1 that time could be a better metric for understandability as it is difficult to measure how descriptive and informative code is for a developer.

As we mentioned, the refactoring techniques used by the research presented in this section is based on the techniques of Martin Fowler. These techniques resemble the techniques we discussed in section 2.2, such as the "Single responsibility principle" and "Don't repeat yourself". One difference though, is that Fowler's techniques is meant to improve the design of existing code. On the other hand, Clean Code describes how one should write code to attain understandable, maintainable and readable code. So the approach is different, but the results should be much the same. In section 2.5 we discussed how Clean Code could help increase code quality. The results from these studies suggest that our reasoning is correct, but as the results vary we cannot conclude anything.

2.6.2 The Effect of Refactoring on Productivity

Moser et al.[19] studied the effect of refactoring on productivity and quality. Their approach was very different from the one we described in section 2.6.1. By conducting a case study in a close-to-industrial environment, they found that refactoring increases quality and productivity.

Moser et al. follows a newly started agile development process over eight weeks, divided into five iterations lasting from one to two weeks each. As part of the development process the team had to solve two user stories focused on refactoring. The first story during the second iteration and the other during the fourth. This way they could compare productivity and quality between iterations. Productivity was measured as the number of lines of code added during each iteration. Moser et al. observed that the average productivity was higher in the iterations after refactoring than the other iterations.

The fact that this study is a case study in a close-to-industrial environment makes their findings extra special. Observing the effect of refactoring on a software project over a longer period gives a completely different type of data than the studies we discussed in section 2.6.1. From the data produced one could also tell if refactoring helps with maintaining the quality of software. Moser et al. suggests that "refactoring of a software system prevents in a long term its deterioration".

There are some problems with Moser et al.'s research, which makes it difficult for us to see how their study proves that understandability also is increased by refactoring. Firstly, they focused on productivity with lines of code added as measurement in a small newly started project. Even though many lines of code are added it does not necessarily mean that the code is more understandable after refactoring. As the project is new, high productivity is expected due to the project taking shape. Secondly, the same team of developers participated throughout the case study. An increase in productivity could then also mean that the developers are becoming more accustomed to the source code.

2.6.3 The Effect of Refactoring on Understandability

When we started working on this thesis we could not find any research that studied the effect of refactoring or Clean Code on understandability in any other way than calculating the code quality score. This led us to use time as measurement for our research. After we started our work however, Erik et al.[3] published a study on the effect of refactoring on understandability in which time was used as measurement. Unfortunately this research did not come to our attention until the later stages of our work. If it had come to our attention earlier, we could have built on their research.

The study performed refactoring on a system with characteristics of legacy code. They used refactoring techniques from both Martin Fowler[11] and Robert C. Martin[17], and categorized the refactorings based on the number of classed they added. They used three categories: small, medium and large. If no classes were added it would go into the small category, one class added would qualify as medium and finally two or more classes would be categorized as large. Then they conducted five different experiments where an experimental group solved assignments on the refactored code and a control group on the original code. For each experiment they created an assignment in the form of either bug fixing or changing functionality. The average time the groups used on an assignment were then compared to determine the effect of the refactoring.

The research is similar to ours in many ways, but on a larger scale. Their refactorings had larger changes to the system, as they extracted code and created several new classes. In their most substantial refactoring they split a class with 350 lines of code into multiple new classes. Furthermore they conducted several experiments and had more participants. Even so, as we discuss in section 6.5, our study supports their findings.

Chapter 3

Research Method

This chapter presents our research method. We discuss experimental research design, how we implemented our experiment and present the participants. After the first run of the experiment we found reason to make some adjustments to the implementation of the experiment for the second run. To make it clear why we made these changes to our method we present the problems we experienced in section 3.5. In section 3.5.4 we discuss what we learned and what actions we decided to take.

To answer our research questions we acquired a system that we found containing code-smells[11, 17], plus violating many of the different rules and principles described in Clean Code. We refactored a class in this system, removed the code-smells and made it more clean. This system and the refactoring we did is presented in chapter 4. Then we conducted an experiment.

The expected result of this experiment is that the refactored version of the system is more understandable. Therefore participants should require less time to solve the assignments.

3.1 Experimental Research Design

Experimental research is used to understand causal relationship by manipulating one or more variables, while controlling and measuring any change in other variables[6, p,290]. The variable manipulated is the independent variable, and its values are set by the experimenter. It is a variable independent of the participant's behavior. To manipulate and independent variable, it needs to have at least two levels that participants can be exposed to. When manipulating the independent variable, the goal is that it causes change in the recorded variable. This is the dependent variable: a variable dependent on the behavior of the participants. If there is a causal relationship, the value of the dependent variable will to some extent depend on the level of the independent value.

The most basic experimental design is exposing one group of participants to a level of the independent variable, while not exposing another group[6, p,109]. The group exposed is called the experimental group, and the other group the control group. The control group's behavior is the baseline, against which the experimental group's behavior is compared.

The goal of our research is to see if Clean Code has any effect on understandability. Clean Code is the independent variable, and understandability the dependent variable. As we are able to manipulate the independent variable, and want to measure and observe the dependent variable, an experimental research design is suitable[6, p,109]. To measure the effect of the independent variable on the dependent variable, we measure the time the participants spend on solving coding tasks. If Clean Code does increase the understandability the time participants use should decrease.

Experimental designs can be categorized into three basic types[6, p,291]:

- **Between-subjects design**

  Participants are randomly assigned to the different levels of the independent variable. In other words, they are part of the control group or the experimental group, but never both. The data are averaged across the subjects.

- **Within-subjects design**

  Participants are exposed to all levels of the independent variable. They are part of both the experimental group and control group. The data are averaged across the subjects.

- **Single-subject design**

  Participants are exposed to all levels of the independent variable. Instead of focusing on changes in groups, the focus is on single subjects or a small number of subjects.

In this study the independent variable only has two levels, whether the code is written following the Clean Code principles or not. This means we only need one experimental group and one control group. The single-subject design focus on the effect the independent variable has on individuals, whereas we are interested in the general effect of Clean Code. Therefore, between-subjects or within-subjects designs are more suitable.

Were we to use the within-subjects design, we need to refactor at least two different parts of the system, and create similar assignments for both.

Using only one part and one set of assignments would not work due to the carryover effect[6, p,306]: After finishing the first test, the participants would know all the answers and the second test would be meaningless. Refactoring two parts is time consuming and a job for future research. Therefore we found it better to use between-subjects design.

3.1.1 Randomized Two-Group Design

In this study we chose to use a randomized two-group design, as we found this to best fit our case due to its simple design.

When using the randomized two-group design, the first step is to randomly assign the participants into two groups. This randomization helps dealing with error variance such as individual differences. These groups are then exposed to different levels of the independent variable. When conducting the experiment, other variables should be kept as constant as possible between the groups. Finally, the means are calculated from the results, and analyzed[6, p,294].

The design is very simple and it has some clear advantages[6, p,296]:

- It is simple to carry out.
- No need to pretest or categorize subjects.
- It requires few subjects.
- Naturally randomizes subjects across the groups.

One of the downsides with this design is that it provides little information about the effect of the independent variable. The result only tells if the variable had an effect. We can see if Clean Code had an effect on understandability, but not in *what* way. We cannot tell if Clean Code decreased the time used because it makes code more maintainable, and so on. This out of scope for this study and something future research can study in more detail.

3.1.2 Dealing With Error Variance

There are other variables that could have an effect on our dependent variable. These variables create an error variance. Error variance could be a problem, as it affects our ability to determine the effectiveness of our independent variable[6, p,293]. The variables that are most threatening to this study are individual differences between participants and the implementation of the experiment.

We are dealing with individual differences by assigning the participants to groups at random. This way the individual differences should be evenly distributed across groups.

The implementation of the experiment could introduce error variance if it is not consistent. It is important that all runs of the experiment are as consistent as possible. All information given to participants should be standardized as to not affect the variables

3.2 Implementation of the Experiment

The participants were randomly divided into two groups and asked to solve three code assignments. One group was assigned to the original version, and the other group to the clean version of a system. This system and the refactoring we did to create the clean version is presented in chapter 4. The participants were not aware that there was two different versions.

To record the time each participant spent on each assignment they were given a form to report the start and finishing time. They then calculated the time spent on each assignment as they finished with all three. To control that the assignments were solved correctly the participants were instructed to raise their hand when they thought they had solved an assignment. The solution was then controlled and the participant told if the answer was correct or not. If the solution was incorrect they continued with the assignment.

Other approaches we thought of were to write several unit tests that they had to pass for the assignments to be approved. We decided against this because we felt that unit tests would point them too much in the right direction and perhaps spoil the results.

We did two separate runs of the experiment, to keep the groups small so we could be more available during the experiment. We wanted to be available so we could control solutions as fast as possible. If participants wait for our control and this adds to their time it would contribute to error variance. After the first run of the experiment we made some changes to the implementation for various reasons. We discuss this in more detail in section 3.5.4.

Unfortunately, the participants were not able to run the system while they were working on it. This is mostly because the system is not working properly since we have stubbed or completely removed many vital parts. We had to do this to be able to use this system for the experiment, as it is part of the agreement we have with the owner of the system. Another reason is that the system is an API, so just running it is not enough. One would have to execute different requests to test it, and this is something we did not want the participants to spend a great deal of time on.

3.3 The Assignments

For the experiment we designed three different assignments to answer our sub-research questions. Each assignment focuses on its own question. We tried to make the assignments simulate work one would experience as a developer. Implementing new functionality, changing functionality and bug fixing are tasks a developer would perform during a normal day of work.

3.3.1 Assignment 1: Implementing New Significant Functionality

"Right now this user-API does not support updating a user. Your first assignment is to implement this. Remember: The user must be authorized. A users username, email and uuid must all be unique."

The first assignment is to implement new functionality in the system, namely the *updateUser* function, see listing A.2 and B.2. We removed the function entirely from both versions along with the unit tests created for it in the refactored version. This is the largest assignment and the one we expect the participants to use most of their time on. The assignment tests the difference in how well the two code styles are understood when working with them for the first time, and how difficult it is to add new code into them. The result will reflect the ability to reuse code and how understandable the system is overall. As this is the first assignment it is expected that much of the time used on this assignment is used to learn the system and how it works.

The expected solution is the same in both versions of the code:

- The updated values in the user-object must be validated as legal values.
- If username or email has changed they must be validated as unique, for obvious reasons.
- The user doing the update must be authorized.

Updating a user is much like creating one, therefore looking at how a user is created can be of much help to the participants. They were not given this advice, but we expect them to see the similarities. Because of this the assignment can be solved by copying much of the logic from *createUser*, see listing A.1 and B.1. *createUser* and *updateUser* shares a lot of functions, so there is not very much coding needed, just the additional validating of unique username and email. This is true for both the original version and the refactored version. This makes the assignment consistent, as there is no clear advantage to either of the versions.

3.3.2 Assignment 2: Change Current Functionality

"Almost every function takes a parameter called "userId". Right now the API treats this as a UUID. Your assignment is to change the functionality so all of these functions also try to fetch a user by username if it fails to find a user by UUID. In other words, after this change "userId" could also be a username."

The second assignment aims to test much of the same as assignment 1, the understandability of the existing functionality in the two versions and how easy it is to identify reusable code. But there is a major difference from assignment 1: This assignment also test how changeable the code is. Currently almost every function in the API requires the UUID of a user to fetch it from the database. The assignment is to modify the functions to also be able to fetch users by username.

The expected solution is to write code that tries to find a user by username if it fails to find one by UUID. This is a short assignment but the amount of work needed differs between the versions. In the refactored version a function that does this already exists, see listing B.6, so the problem could be solved by using it. In the original version, on the other hand, one could write a similar function and use it, or change every function that fetches the user.

We expect the assignment to favor the refactored version, as it is provided with a function that can be used in the solution. This is intended, as having functions such as this to reuse is one of the benefits of writing Clean Code.

3.3.3 Assignment 3: Bug fixing

"There is a bug preventing users from being created with the "createUser" function. Your assignment is to find this bug."

The third and last assignment is about bug fixing. The first two focuses on working with the code and implementing new code into it, but now one needs to really understand what is going on in the code to reveal the bug.

Listing 3.1: Inserted Bug

```
if(dao.getUserByUsername(newUser.getUsername(), false) == null) {
    logger.info(append("method", "createUser"),
    "Username is already in use!");
    throw new ValidationException(
    new Error("duplicate_unique_property_exists",
    "Unique check failed", "This username is already in use!"));
    }
```

The bug is not part of the original system in any way; we have inserted it into the code. It is a logical error inserted into the validation of the username during creation. The error makes the system fail whenever someone tries to create a new user with a username that is not in use, but it will succeed if the username already is in use. See listing 3.1.

The expected solution is to identify the bug and change the conditional from equal to not equal. The simplest approach to finding the bug differs between the two versions. As the original code has all the logic of creating a user inside the *createUser* function, all one has to do is to go through the function carefully step by step until it is discovered. The refactored version's *createUser* has been split into several smaller functions doing all the work, so one has to go through each of them carefully until the bug is discovered.

There is no help in this assignment from logs or any other kind of error messages. The assignment only states that there is something wrong preventing users from being created. Because of this we do not expect the assignment to clearly favor any of the two versions. It might be a little easier to spot the bug in the original version due to the fact that the bug is inserted deeper into the code in the refactored version behind two function calls. It is also possible for the participants working on the refactored version to write unit tests to discover the logical error, instead of looking through the code.

3.4 The Participants

We had a total of 18 people participating in our experiment. Most of them are graduate students, except four who are experienced developers. These four volunteered when we asked for volunteers from a software company. We had hoped more would participate so we could get data from both students and experienced developers. This would have been interesting because a student might not have developed a preference for a specific code style yet. While an experienced developer might mean that understandable code is what he or she is used to work with. Had we had more developers it could have been possible to see if there were any noticeable difference in what students and experienced developers feel is understandable code.

Even though we had more students participating we still had fewer than we had hoped. Why it proved difficult to recruit participants is hard to say, but we noticed during the first run of the experiment we had to use graduate students due to a need of experience with larger systems. This greatly reduced the number of people we could use. In addition, the graduate students were all working on their own thesis and they might have felt that they did not have time for someone else's experiment.

3.5 First Run of the Experiment

During our first run of the experiment we experienced some problems that possibly created noise in the result. We deemed some of these issues harmful enough that we had to make some changes to the implementation of our experiment for the next run. We discuss the changes made in more detail in section 3.5.4. In the following subsections we go through each assignment and discuss flaws we discovered and feedback we received from the participants. The results of the experiment are presented in chapter 5.

In the first run of the experiment we had eight students participating. These eight were randomly divided into two groups. All eight participants were graduate students so the groups should be more or less even in skill and knowledge. What version the groups worked on were decided by a coin toss. The group that was assigned the original code was named group A, and the group assigned with the refactored code was named group B.

3.5.1 Assignment 1

As this is the first assignment most of the problems the participants experienced were things such as importing the system into an IDE, or finding the correct class to work in. These things added noise to the results as many participants added the time used to import and finding the class to the time they used to actually solve the assignment. This was an unforeseen problem, and we decided to fix it for the next experiment.

Other than that the assignment worked as expected.

3.5.2 Assignment 2

For the first run, the assignment text was slightly different from the one presented in section 3.3.2. The assignment text read: *"Right now the functions in our user-API uses the parameter "userId" to find a user by* *uuid, but userId could also be a username. Implement the functionality to support this."*

There were few problems with this assignment, just some minor misunderstandings about the assignment text. Some participants did not understand what the assignment was asking them to do and needed some extra explaining. Because of this the text needed to be more explanatory and go more into detail.

3.5.3 Assignment 3

Again, the assignment text was slightly different from the one presented in section 3.3.3 for this run. The assignment text read: *"There is a bug in createUser. Find it."*

We had some problems with this assignment text. Many of the participants from group B told us that they misread the text to mean "inside" the function *createUser*. They were only looking at the function calls, and not further exploring the functions called. So they based their answers on the order of the calls, and not what the other functions called do.

They also looked for other functions throughout the system that they thought the assignment referred to. Group A did not have this problem, as all the logic was inside *createUser*. They did as expected and read the function line by line and eventually discovered the logical error.

Whether group B misread the assignment because of their inexperience with Clean Code, larger projects or because of the text itself is hard to say. Either way it provided valuable feedback on the assignment text.

3.5.4 Learnings From the First Run

After the experiment we had a short discussion with the participants about the experiment. They felt the size of the project was problematic. With so many classes and files it was confusing and difficult to know where to start. The assignment text was deliberately written so as to not tell the participants where they should start. This was to create a more realistic situation. But this may also have damaged the results as the times used by participants to discover the problem area differed. The problem is that participants located the correct class at random times, so we cannot know if the time recorded represents locating the class or solving the assignment. A participant could spend a lot of time finding the correct class, but very little to solve the actual assignment. Or the other way around.

This creates questionable results as the time used to solve the assignment disappears in the time used to find the correct class. We are only interested in the effects of the refactored code. Therefore we decided that the assignment texts should clearly state where they should be solved.

Because the size of the system proved to be challenging for this group of graduate students, we decided that for the next run of the experiment we should only use participants on master level and higher. This meant acquiring volunteers would be more difficult, but it assured us that they are capable of solving the assignments.

The main points of what we learned after this run was that the assignment texts needed to be more precise and explanatory, and that we should point to the class in which all the assignments are to be solved. As we discuss in section 4.1 we only refactored this class in the system. Based on the feedback from assignment 3, we changed the text to be more explicit by stating that there is something wrong with the process of creating a user. We hoped this alteration would make the participants go through the function calls and debug them as well.

When rewriting the assignments to be more precise and explanatory, it is important that they are not written too explanatory. We do not want to guide the participant to the solution step by step, but we want to point them in the right direction. We want the text to make them understand that what they need to solve the assignment is already in the class, or needs to be created by them. Since we only refactored this one class, as discussed in section 4.1, we do not want participants to dig through other parts of the system. The new assignment texts are presented in section 3.3.2 and 3.3.3.

To control the solution of the participants, and decide if it passes or not is also something we decided against doing after the first run. The reason is that participants were given different hints and pointers when their solution did not pass. In other words, it was difficult to keep all variables consistent and standardized. It was also difficult to avoid other participants hearing other proposed solutions, leading them to the answer. To avoid this, we decided that the participants should themselves decide when the assignment is solved, and deliver their work when they finish. We then have to go through their solutions and control their answers. If their answer is wrong, or missing important logic, we have to take this into consideration when comparing the results. The error rate represents additional data we can use.

After making these changes to our experiment, we hoped that the size of the project would not be a as much of a problem, as the participants would not have to look through it to find the correct class. Furthermore the assignment texts should be clearer and not cause confusion or misunderstandings. Also, we hoped to eliminate as much noise as possible in the results, especially the time used to locate the problem areas.

Chapter 4

System and Refactoring

This chapter introduces the code base we used for our experiment, and why we found it suitable for our experiment. We point to different problems with the code in section 4.1, and in section 4.2 we discuss the refactoring done to make the system more clean.

To be able to make the system clean we had to study Robert C. Martin's book Clean Code[17]. We studied code quality and compared Clean Code to code quality and its different metrics to make sure that Clean Code also increases the objective quality of software. This is presented and discussed in chapter 2. We performed several types of refactoring, from just renaming variables and functions to extracting code to divide responsibility and to reduce duplication.

All of the functions we discuss in section 4.2 are presented in their original form in appendix A. The refactored versions are presented in appendix B.

4.1 The System

We have worked on a part of a larger system written in Java. It is a REST-API handling create, read, update and delete requests(CRUD-requests) from users of the system. Even though it is just one part of the entire system it is nonetheless a sizeable part. Apart from handling requests it also authorize users, queries the database, sends emails and so on. For it to be able to do all this, it is dependent on several different modules and libraries.

Because of this we decided to focus on the API for managing users. This API is not dependent on too many modules and it is straight forward in what it is doing. Typically it handles CRUD-requests and some other requests, such as exporting users and getting the roles a user have in the system. It is a sizeable class with long functions doing more than one thing. There is also much duplication of code and unnecessary complexity throughout the class.

In addition, this class has some other problems. It resembles "Legacy Code"[10], it contains "Code Smells"[11, 17] and it breaks several of Martin's rules and principles in Clean Code. In the following sections we briefly discuss what Legacy Code and Code smells means, and then why, in our opinion, this system suffers from these problems.

4.1.1 Legacy Code

The noun legacy is defined as: "something handed down by a predecessor"[1]. It is something old, but not necessarily bad. In the software development world, legacy can mean many different things, and it is mostly used as a negative adjective. "Legacy code" and "legacy system" is something most developers dread to hear.

Michael Feathers defines legacy code as "code without tests"[10, p,xvi]. Code without tests is the worst case scenario, as we discussed in section 2.2. Without tests one cannot control whether or not the code is changed for better or worse, thus change gets difficult and time consuming.

In the book he describes typical characteristics of legacy code[10]:

- Classes and functions in a legacy system are large and complicated.
- Figuring out what to do and where to do it in a legacy system can be difficult. Implementing the change is also a problem.
- Legacy code often has duplication. Whenever changes are introduced it has to be implemented in several places.

Legacy system is mostly used to describe outdated systems, written by a previous generation of developers. These systems can be difficult to change, and hard to understand[10, p,77]. Mostly due to their tangled and unintelligible structure and outdated technology. Even though these systems are outdated they are often still widely in use. Legacy systems are also highly cost inefficient, as making a small change is difficult and time consuming.

The term *legacy code* is used to describe code that has the same characteristics as legacy systems. It can be old code that is part of a system that is constantly updated and still evolving. Legacy code is that old hairy piece of code that no one dares to touch in fear of breaking something. This old code can be a burden on the evolution of the system, and slow down development.

Even though we speak of legacy code as old code, it is not always the case. If a company experiences that their product is popular and the users are requesting a great deal of new features and fixes, the developers may rush these out to keep their product popular and growing. This could lead to poorly written code with many of the same characteristics as legacy code. "The Beast", as we discussed in section 2.2, is a good example of such legacy code, with its long and complex functions.

4.1.2 What is Legacy in the System?

If we try to identify the characteristics given by Fowler, it is easy to see that the class matches the first in his list. The functions in our class are long. They are all above 20 lines, some are close to 100 lines. In addition, there are many if statements and multiple exit points.

One of the most notable reasons for all these if statements and exit points are duplication; number three on Fowler's list. Every function executes the same authorize-check. The check is not very complicated, but it introduces both an if statement and an exception. Since this is the user-API there is a lot of fetch operations for user-objects from the database, which are also executed by many of the functions. Duplication is one of the most substantial problems in the class.

The size and the duplication makes the class unnecessarily complicated. We expect this to make it more complicated to make changes, to debug and to find where to make the changes. If this proves to be true after the experiment, it also matches Fowler's second characteristic.

Furthermore, there are no unit test. By Fowler's definition, the class is legacy code based just on this. Since the class matches two of the three characteristics, and there are no unit tests, we classify the class as legacy code.

4.1.3 Code Smells

Martin Fowler uses the term "Code Smells" in his book "Refactoring: Improving the Design of Existing Code"[11, p,75] to describe problems with code that points to a need for refactoring. It is a sign that there is something wrong with the system, possibly a deeper problem. These problems are not bugs or something that breaks the system, but rather signs of bad design and implementation. Fowler lists several smells in his book[11, p,75], smells that indicate a need for refactoring:

- Duplication
- Long Classes
- Feature Envy
- Comments

In addition to Fowler's definition of code smell, one could also say that code smells point to violations of principles and poor quality. Our definition is a mix of the two, as we point to different violations of the principles and rules described in Clean Code, and some of Fowler's smells, in our system.

4.1.4 The different code smells in the system

Functions Should Do One Thing

Martin states that functions should do one thing, and one thing only[17, p,35]. He is not saying that the function should only contain one line, but rather that the function should only operate on one level of abstraction. Specifically one below the functions name. Many of the functions in this class violate this by operating on several different levels of abstraction. A good example is the *createUser* function, listing A.1. This function does more than just creating a user:

- First of all it validates the values given by the user. If one of these proves to be invalid the function aborts by throwing an exception.
- The location of the user is located using the IP.
- After the user been created the function tries to log the user into the system.

Each of the points listed above operate on their own level of abstraction. Having them in the same function then obviously makes the function operate on more than one level of abstraction.

Descriptive Names

Most of the names used on variables and parameters are descriptive enough. They state their purpose and functionality. The problem is the lack of descriptive function names. This is mostly due to the problem that each function does all the logic by itself. There are no small helper functions with names that can help with understanding what these large functions are doing. This code smell warns about more code smells; the functions are not doing one thing, they have more than one responsibility, and they contain possible side effects.

Misplaced Responsibility

This code smell goes hand in hand with "functions should do one thing". If a function is doing more than one thing, it is taking on too much responsibility.

Duplication

We mentioned in section 4.1.1 that the class suffers from duplication. The duplicated code mainly performs authorization and fetching of user-objects from the database. These are not long parts of code, but they introduce noise and complexity.

4.1.5 Summary

After identifying both characteristics of legacy code, and different code smells we judge the chosen system to be suitable for our experiment. It is suffering from some problems but these are not making the system impossible to work with. If the problems were too large the experiment would not have the same credibility, as the assignments would not be fair. Comparing a very messy system, with a lot of problems, to a clean system would not prove much. Any kind of system with less problems would most likely prove to be easier to work with.

4.2 Refactoring

We discussed in section 4.1 that we decided to refactor one specific class of the system. The responsibility of this class is to receive and answer requests. Therefore it is fitting to abstract the logic that creates the answer out of the interface code and into an internal service. This new service is responsible for creating answers and executing processes for the REST-API. Fetching user-objects, saving user-objects, validation and authorization etc, is all part of the responsibility of the new service. We now have two layers of abstraction instead of one. Instead of one interface handling all by itself we have an interface handling the requests and a service layer beneath, handling the logic and creating answers. Each layer has its own responsibility thereby fulfilling the single responsibility principle.

4.2.1 Extracting Repeated Code

Many of the requests that the API handles require the same steps to create an answer. One example is validating if the user has the correct authority. Another example is to retrieve information about a user from the database. Extracting this functionality into separate functions is beneficial for several reasons. Most importantly, it greatly reduces duplication. If the way users are authorized changes, the work needed to implement the change is very time consuming and complex if the code is duplicated across the system. When extracted into a separate helper function this would only have to be changed in one place. There are other benefits of extracting. The logic can be explained in the form of function names. In addition, the error handling is hidden. Instead of having a confusing if statement we have a function call with a descriptive name.

Almost every function in the API does some work on a user-object. Therefore the functions try to fetch a user from the database. This little piece of code, listing A.4, was repeated in most of the functions. After realizing this we decided to extract it into a separate function.

Code listing B.4 shows the result of the refactoring. The new function retains the functionality: it does a null check and then throwing an exception if the user is not found. By extracting this logic, we get rid of the noisy null check in almost every function as well as the creation of the exception. We also have to modify the code only once if a change is needed. The function also brings more security into the system as it never returns null. We will discuss this further in section 4.2.4.

Duplication With Minor Differences

Sometimes when trying to extract repeated code one can come across repeated code with minor differences. We experienced this when we tried to refactor the validation of user fields when a user is created and updated. They both validate the same fields, but the validation of an updated user has some special cases. When we create a new user we always want to make sure that the username is not already in use, but when we update a user it is only a problem if the username has changed.

To eliminate all duplication, we tried to make a function that could serve both purposes: Validate the username whenever a new user is created. When updating a user, we only validate the username if the username has changed. The result was code listing B.7. This function takes two parameters, a user and a boolean used to determine if the user is new. It then declares two booleans, one for username and one for email, and initializes them to true. If the user is updated these booleans are set corresponding to whether the values have changed or not. Then, if the username has changed and there is a user with the same username we throw an exception. We felt there were multiple problems with this solution.

Most importantly this new function felt more complicated than necessary. During the refactoring we had to introduce more if statements, and make the existing ones more complicated to maintain the original behavior. This made it harder to understand what the function was doing with a quick look, especially since the function behaves differently whether or note the user is created.

After considering this we decided to try something different. This time we extracted the shared code into separate functions and created a validation function for both created users and updated users. This way the functions read much easier and one can understand the functions after a quick look. Even though we now have two fairly similar functions, see listing B.8 and B.9, they are each less complex and more descriptive than the single function we first created.

What Can We Learn From This?

What we did in our second attempt was to value code with good understandability, instead of low duplication. We lowered the duplication, but allowed some in order to achieve better understandability. The first attempt removed all duplication, but it came at the cost of understandability. There is a fine balance to what we did though. Much duplication in order to achieve understandability might not be worth it due to the high workload needed when change is introduced to the system. Low understandability to achieve low duplication has the same effect, implementing changes in a system that is difficult to understand can be time consuming.

But could we have seen the problems earlier when we started refactoring? Did we break any of the rules in Clean Code? First of all we introduced the boolean parameter *creatingUser*. This parameter is needed to trigger the validations to be done when the user is created. It is the parameter that decides the behavior of the function, and this can be confusing. Martin talks about how to not use flags as parameters, because it is an indication of the function doing more than one thing[17, p. 41]. We could have used this rule as an indication of going in the wrong direction, and divided the function into two different functions right away.

4.2.2 Better Names and Abstraction

Even though some of the code in the REST-API is not used by more than one function we have moved it into separate functions. These functions are given descriptive names, the benefit of this is discussed in section 2.2. By moving the code we also keep our functions small and focused on doing one thing. Furthermore, by doing this we also fulfill the criteria of The Stepdown rule[17, p,37]. The rule states that every function should be followed by a function at the next level of abstraction. The point is to make the functions read somewhat like articles, explaining their use.

Authorization

The authorization of users, see listing A.5, is a good example of code that is very easy to extract from the original code and where one can really see the benefits of doing this. First of all the code is basically just an if statement. It is very short and it does not do much. It can also be difficult to understand why it is there and what it is doing at first glance, especially when it is mixed in with plenty of other statements.

When extracting this logic into a new function we remove noise such as brackets and logging statements, but more importantly we introduce a name, a name that describes the function. Just by looking at the function call one immediately understands why it is there and what it does. The name also makes the function easier to debug as the name acts as a manual, telling us what the function is supposed to do. The result is listing B.5.

Update User

Let us look at the function *updateUser*. In the original code this function is cluttered with if statements and exceptions, as seen in listing A.2. Most of this is validation of the updated user object, checking whether the birthday is valid or if the username is in use, and there is also error handling. With all this going on it is difficult to get a clear overview of what the function is doing. One have to step through line by line to make sure nothing is overlooked. To make this function more in line with the clean code principles we have extracted more or less all of the logic into several smaller functions. All of these functions have one single responsibility and a descriptive name. We already discussed these functions, the authorize function *throwIfNotAuthorized*(listing B.5), *getUserByUUID*(listing B.4) and the validation of fields *validateUpdatedUserFields*(listing B.9).

The result is a much tighter and smaller function, see listing B.2. This function is much easier to read and follow than the original version. It authorizes the user, fetches the user from the database, validates the fields and then updates the user. This is pretty clear from the function calls thanks to their names. The function is also now easier to debug if something should go wrong. If there is something wrong with the validation, one knows immediately where to start looking.

4.2.3 Reducing Complexity

Trying to understand a function with a high cyclomatic complexity can be a mind-bending task. With all the different outcomes it is difficult to get a clear overview of what could potentially happen when one calls the function.

The function *getUser* is a function with many decision points. As we can see in listing A.3 there are five different if statements, where one consists of an or expression. This gives a total of six decision points, some of them are nested and some even duplicated. If we use the formula discussed in section 2.4 to calculate the cyclomatic complexity of functions with multiple exit points we see that the function has a score of six. Six decision points and two exit points gives us the calculation M = 6 - 2 + 2.

What creates problems in the function is the fact that the parameter *userId* can be both UUID and a username. It looks as this functionality was implemented at a later stage and the programmer who did it "broke a window", as we discussed in chapter 1. Instead of refactoring the function and create a function that tries to acquire a user-object by both username and UUID he changes the parameter to be the users UUID, if it is the username, to work with the existing code which expects the parameter to be the UUID. He introduced two decision points and it makes the function more confusing than it needs to be.

What we did to reduce the complexity of the function was to first extract all of the logic concerning finding the user-object, both by username and UUID. We created the function *findUser*, see listing B.6, which first tries to find a user-object by UUID, and then username. By doing this we can remove the logic that is changing the userId parameter, and the logic fetching the user on line 17, after the authorization in the original *getUser*(listing A.3). We are also using the authorize function discussed earlier, listing B.5.

The result is listing B.3. The function itself has a cyclomatic complexity of 1 and the functions it uses has a maximum of 2. Now that the function is smaller and it uses functions with descriptive names it is easier to understand what is going on. As a result, this new version of *getUser* also reads as an article.

4.2.4 Error Handling

Throughout the REST-API there are incredibly many null checks and exceptions thrown. There is also some inconsistency in how the functions handle the different errors. In *getUser*, listing A.3, if a user cannot be found null is returned, while in *updateUser*, listing A.2, an exception is thrown instead. Most of the time an exception is thrown, but there is no consistency in the messages sent with the exceptions.

Many of these null checks can be avoided if we handle the error where they occur instead of just returning them. According to the rule "Don't Return Null" in Clean Code, a function should not return null. It should deal with the null right away[17, p,110]. The reasoning is that if a function returns null the problem is only postponed. One would have to deal with it whenever the function is used, and this is something that can be forgotten. If it were to be forgotten only once it could lead to severe errors in the system. Therefore it is much better to deal with the problem right away in the function providing the null. So by throwing an exception instead of returning null there is a smaller chance that there is introduced errors into the system.

This is what we have done with *getUserByUUID*, listing B.4. After fetching a user-object we check if it is null. If that is the case we throw an exception instead of returning null. When a user-object is found we return it. Instead of letting a potential null value bubble upwards we deal with it right away. If we let it pass it is possible that when this function is used somewhere else or by someone not familiar with the system they forget to do a null check. Furthermore, as this function is used to retrieve a user-object that is about to be altered or manipulated, there is really no point in returning null if a user is not found as there is nothing to manipulate. As mentioned earlier there was inconsistency in how this problem was handled in the system. By doing what we have done we increase the consistency and remove a lot of null checks as every function now uses *getUserByUUID*.

4.2.5 Responsibility

Responsibility is an important point of Clean Code as we discussed in section 2.2. The single responsibility principle means that a class or function should only have one reason to change. There are several functions violating this rule in this system and one of them is the *exportActivatedUsers* function, listing A.6.

exportActivatedUsers builds a CSV-file with all the relevant information stored about activated users and other information. The function creates a BufferedWriter, loops through all of the activated users and appends data to the output. This leads to different reasons why this function might have to change.

- The function would have to change if the information exported changes. Say a new variable is added to the user that also should be exported, the function would then have to change to include this in the export. Also, if a variable is not relevant anymore for the export, it would have to change.

- The function depends on several other modules to work, such as BufferedWriter. If the implementation of the BufferedWriter changes the function would have to change to follow the changes made to BufferedWriter to keep the intentional functionality.
- If the requirement specifications on exporting users were to change, the function would have to change. Say a new requirement is to be able to choose the file type. This would force a change.

To make sure that our new service follows the single responsibility principle we tried to identify different responsibilities and delegate them to the correct object:

- The function uses a while loop and a for loop to loop through 200 users at a time. We created a new class, *UserPager* (listing B.10), and extracted this logic to the class to isolate the responsibility. This class fills an ArrayList with "pages" with 200 users in each. This list can be fetched as an iterator so one can iterate page by page of users. If there were to be changes in how all the activated user-objects are fetched, only this class would have to change.
- There are several lines of append statements in the function, and it is a little difficult to see what is appended and from where. We moved the responsibility of exporting the data to the object with the relevant data itself. In the user class we created a function that takes a writer as a parameter and then adds all of the relevant data to this, see listing B.11. If we now add more data to the user that we want to export, we only have to change the user and not the export function itself.
- Finally we create a new class, a *CSVExporter* (listing B.12), whose responsibility is to handle the exporting and creation of the CSV-files. This class takes the new *UserPager* iterator we created and iterates over all the pages and passes the writer to the user-objects so they append their information to the writer. If the exporting procedure should change there is only need to change this class.

4.2.6 Unit Tests

Long, large functions are hard to write unit tests for. The more the function does, the harder it is to test every single thing. This is the case for the original code. The functions are large and difficult to test, therefore the original version did not have any unit tests for the API. This makes it much harder to change the code. The question "will this break the code?" must be asked for every single change. In short, there is no safety net[17, p,124].

Listing B.13 shows a test which would act as a safety net whenever work is done to the function *updateUsersPassword*, or to what the function depends on. The test expects that a ValidationException will be thrown when a invalid password is passed to the function. If this does not happen the test fails.

To refactor code with no safety net is not a very good idea. Without unit tests one will not have any control over the changes made. Does the system behaves as expected, or has the behavior changed? Unit tests are needed to be able to answer questions such as this throughout the process.

With this in mind our first step in refactoring was to write small unit tests testing only one concept at the time. With these tests in place we could refactor while feeling safe about the changes as long as the tests did not break. These unit tests were written as clean as possible to keep them readable.

Much of the necessary mocking is done in the setup of the test suite, so the tests are mostly asserts with some mocking and setup to suit the need for the particular test. As with variables and functions naming is important, so the names of the tests describes what the tests are testing and what the expected result should be.

Listing B.14 shows one of the unit test written to test the function *getUser*. This particular test is written to make sure that the function also finds a user when it is passed a username as an argument. The test fulfills the rules discussed in section 2.2, both F.I.R.S.T and the "Single Concept Rule". It is short and does what the name says. The dao is a mock. This means that we have complete control over its behavior. We can then tell it to return a dummy user, which is created during setup, whenever the function *getUserByUsername* is called. By doing this we know that *getUser* should find a user. If *getUser* is changed and *getUserByUsername* is no longer called the test will fail.

Chapter 5

Results

In this chapter we present the errors made by the participants in the second run of the experiment, and the results from the two runs of the experiment we conducted. We discuss the results in chapter 6. We continue with the naming we established in section 3.5, group A worked on the original version, while group B worked on the refactored version.

5.1 Errors

In this section we present the errors made by the participants during the second run of the experiment. The run was conducted in the same manner as the first run, with the exception of the changes we made, as discussed in section 3.5.4.

Out of the ten participants half of them made the same error. In assignment 1 when they are validating the updated values they are not checking if the username or email has changed. Instead they are assuming the values has changed and jump straight to controlling if this "new" username or email is in use. If the username has not changed this would fail the update as the username is in use by the current user. This is not a major error, and one can justify why they assume every value has been updated or why they did not think of it. First of all they do not know the system and they cannot run it. Running the system and experiencing how updating a user works on the frontend gives a great amount of information needed when writing the *updateUser* function. Because of this they have to make a decision, should it be possible to update a single value at a time, or is every value updated?

Three of the five who made the error was part of group A. One of the two from group B lacked more logic than just checking whether the username or email had changed. He was not validating anything. This is a vital part of the assignment. Due to this, we claim that the amount for errors in assignment 1 is evenly spread between the groups.

Only two failed to find the bug. One had a wrong answer while the other did not have the time to finish. Three participants in the first run of the experiment also failed to find the bug before they had to leave. All the participants who had to leave before they found the bug, worked on the refactored version. What is important to note is that these participants did not use more time on the first two assignments than the others. On the contrary, some of these participants solved the other assignments with some of the best times. This is important because it shows that it is not a lack of skills making them miss the bug. They spent on average 32 minutes looking for the bug before they had to leave, while group A participants found the bug with an average of 20 minutes.

Other than these errors no other faults were made.

5.2 First Run of the Experiment

The results from the first run of the experiment show that group B finished with the best average time on two of the three assignments. We now present the results from each assignment.

5.2.1 Assignment 1

Group A's best time was 42 minutes. They averaged around 49 minutes, while group B averaged at 46 minutes. The first participant to solve the assignment was part of group B. He solved it in 24 minutes, while the next person finished 7 minutes later. He was also part of group B.

As we can see from figure 5.4 the participants of group A finished with a more consistent result than the participants of group B. B had a much larger spread in the results.

5.2.2 Assignment 2

Assignment 2 had a great span between the groups. Group B finished with an average of 8 minutes, while group A finished with an average of 23 minutes. B finished 15 minutes faster than group A. The first participant to finish was from group B, after 5 minutes.

Figure 5.5 shows that the results of group A are once again very consistent, but so are group B's.

5.2.3 Assignment 3

This is the assignment with the most notable time difference. Group A had an average time of 8 minutes while group B had an average of 35 minutes. Group B used 23 more minutes than group A. Two of the best times came from group A, with three minutes. The closest participant from group B found the bug after 19 minutes.

Unfortunately three of the participants from group B had to quit the experiment before they could locate the bug. See figure 5.3. They had to leave for personal reasons. We decided to use the time they spent looking for the bug before they had to leave as their result, because they had already spent more time than any member of group A.

5.3 Second Run of the Experiment

In the second run of the experiment group A finished two of the three assignments with the best average time. We will now present the results from each assignment.

5.3.1 Assignment 1

Group A finished the assignment with an average time of 37 minutes. This is 10 minutes faster than group B's average of 47 minutes. Thanks to the changes we made to the experiment, see section 3.5.4, we now know that this result only reflects the time used to solve the assignment.

Even though group A had the best average time, three of the participants made the common error we discussed in section 5.1. In group B one made the same error as group A, while one made a greater error; not to validate anything. The best error free solution came from group B after 25 minutes.

5.3.2 Assignment 2

Group A completed the assignment with an average time of 21 minutes, 14 minutes behind group B's average time of seven minutes.

Figure 5.12 shows that all members of group B finished in 10 minutes, except one who spent 15 minutes. Group B's fastest participant used two minutes.

5.3.3 Assignment 3

Once again Group B had difficulties with this assignment, despite the actions we discussed in section 3.5.4. Group B finished with an average time of 36 minutes, 16 minutes behind A.

Even though group B had a worse average time, one of the best times came from their group. This participant finished in six minutes, exactly the same as the best member of group A. One of the participants in group B had to leave before he could start the assignment, so we have removed him from the average time. This means group B's average time for this assignment is calculated from one less participant than group A.

Figure 5.1: First Run of the Experiment: Table Legend

Yellow	Had to leave before finding the bug
Green	The fastest time in the run

Figure 5.2: First Run of the Experiment: Group A Data

	A	**B**	**C**	**D**	**Average Time**
Assignment 1	64	46	42	43	48,75
Assignment 2	21	38	17	17	23,25
Assignment 3	3	3	22	4	8,00
Total Time	88	87	81	64	80,00

Figure 5.3: First Run of the Experiment: Group B Data

	A	**B**	**C**	**D**	**Average Time**
Assignment 1	59	72	31	24	46,50
Assignment 2	10	5	13	6	8,50
Assignment 3	27	28	19	66	35,00
Total Time	96	105	63	96	90,00

Figure 5.4: First Run of the Experiment: Time used by participants on assignment 1

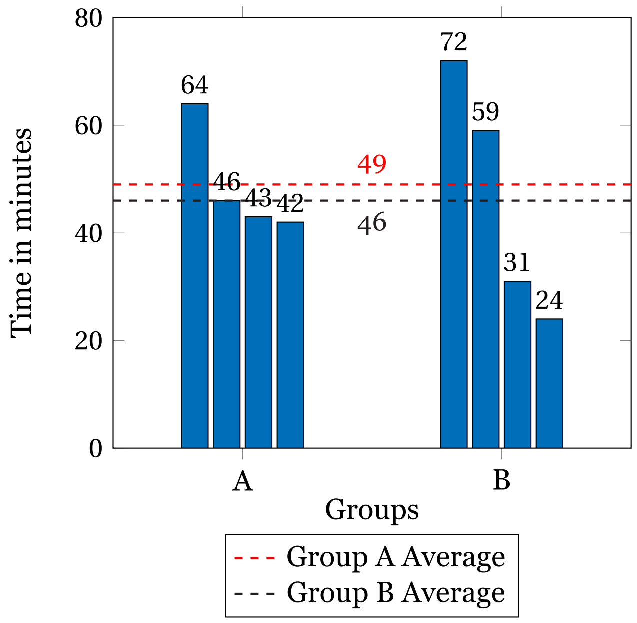


Figure 5.5: First Run of the Experiment: Time used by participants on assignment 2

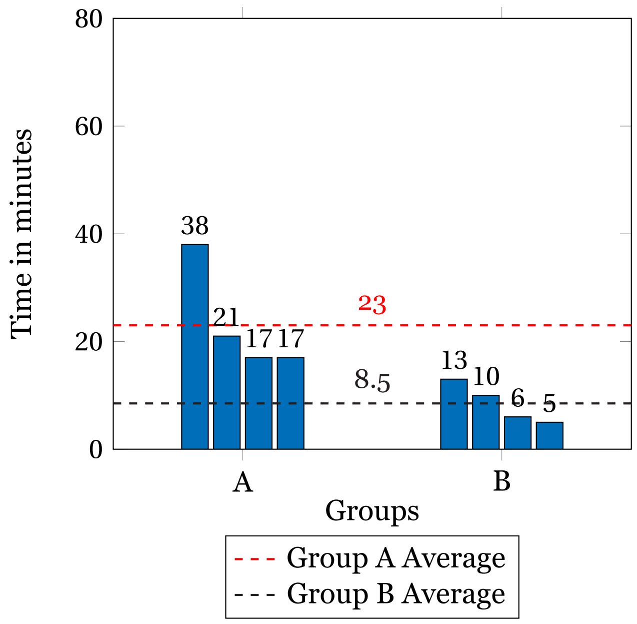


Figure 5.6: First Run of the Experiment: Time used by participants on assignment 3

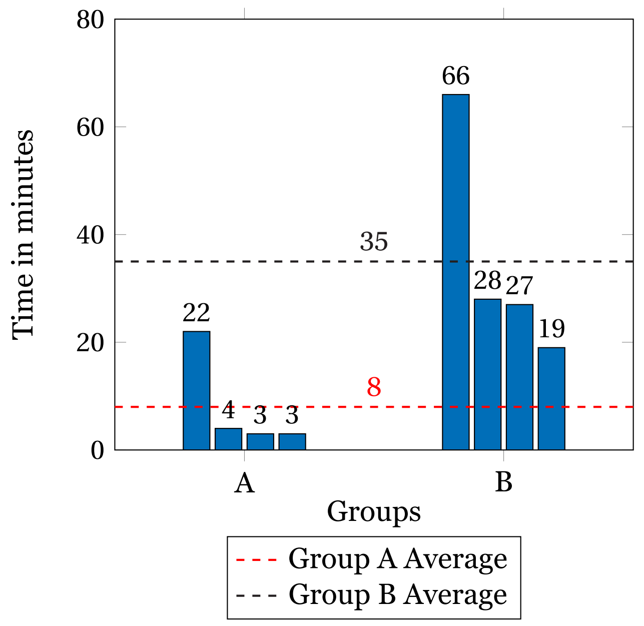


Figure 5.7: First Run of the Experiment: Average time pr assignment

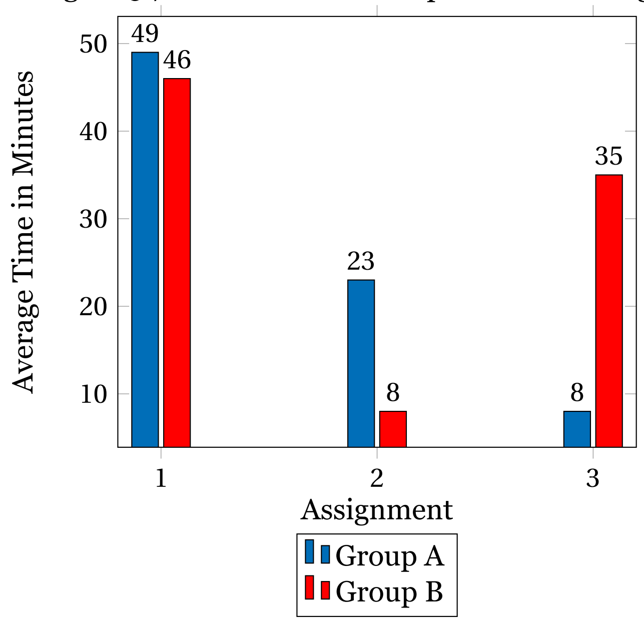


Figure 5.8: Second Run of the Experiment: Table Legend

🟥 **Red**	Wrong solution
🟧 **Orange**	Minor errors in solution
🟨 **Yellow**	Had to leave before finding the bug
🟩 **Green**	The fastest time in the run

Figure 5.9: Second Run of the Experiment: Group A Data

	A	**B**	**C**	**D**	**E**	**Average Time**
Assignment 1	40	48	🟧 24	🟧 21	🟧 52	🟩 37,00
Assignment 2	12	32	26	13	24	21,40
Assignment 3	11	🟩 6	15	🟥 32	53	🟩 19,80
Total Time	🟩 63	86	65	66	111	🟩 78,20

Figure 5.10: Second Run of the Experiment: Group B Data

	A	**B**	**C**	**D**	**E**	**Average Time**
Assignment 1	🟥 36	🟧 66	50	58	🟩 25	47,00
Assignment 2	6	6	15	8	🟩 2	🟩 7,40
Assignment 3	🟩 6	🟨 X	60	25	55	36,50
Total Time	48	72	125	91	82	83,60

Figure 5.11: Second Run of the Experiment: Time used by participants on assignment 1

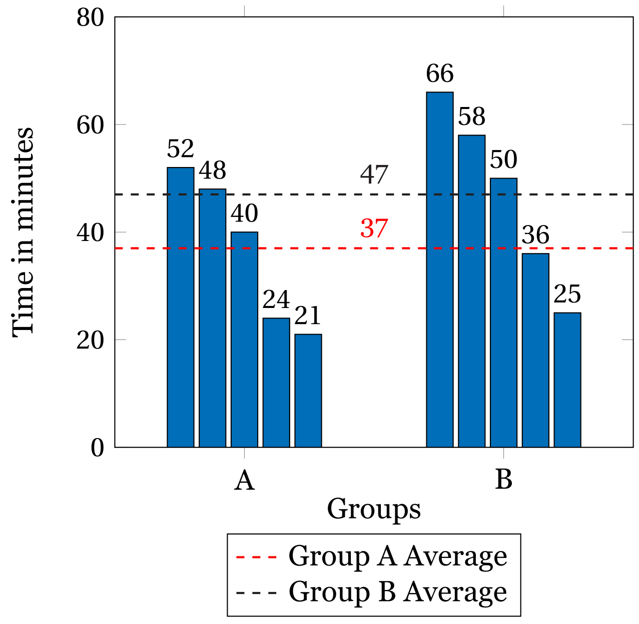


Figure 5.12: Second Run of the Experiment: Time used by participants on assignment 2

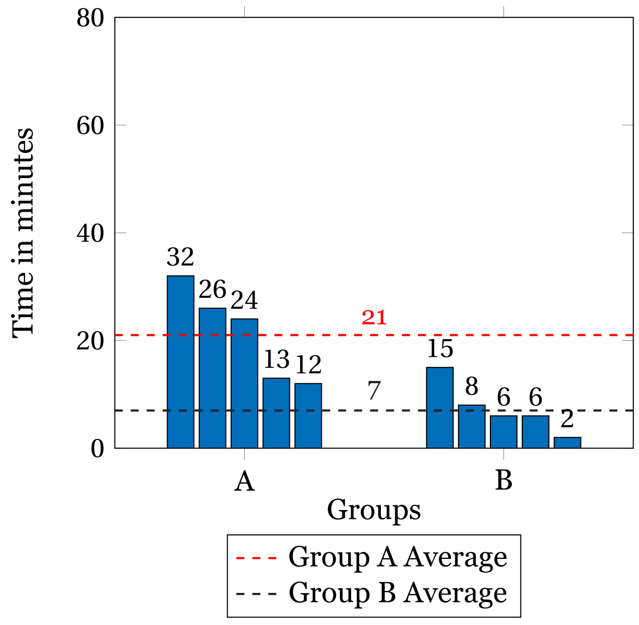


Figure 5.13: Second Run of the Experiment: Time used by participants on assignment 3

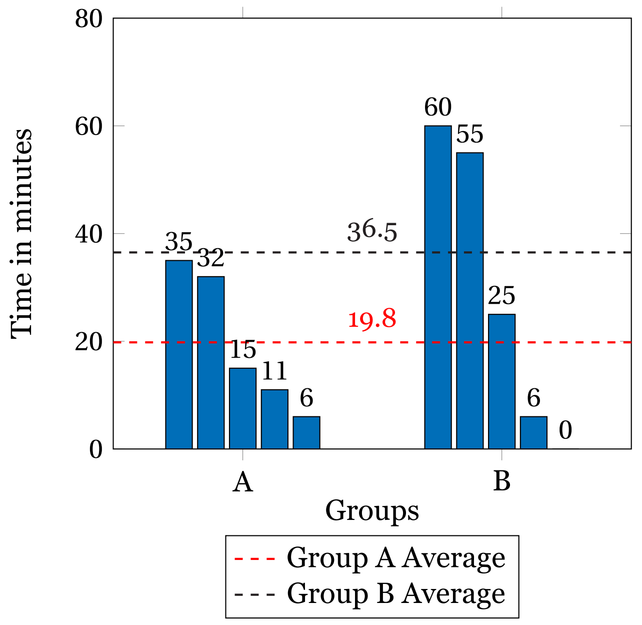
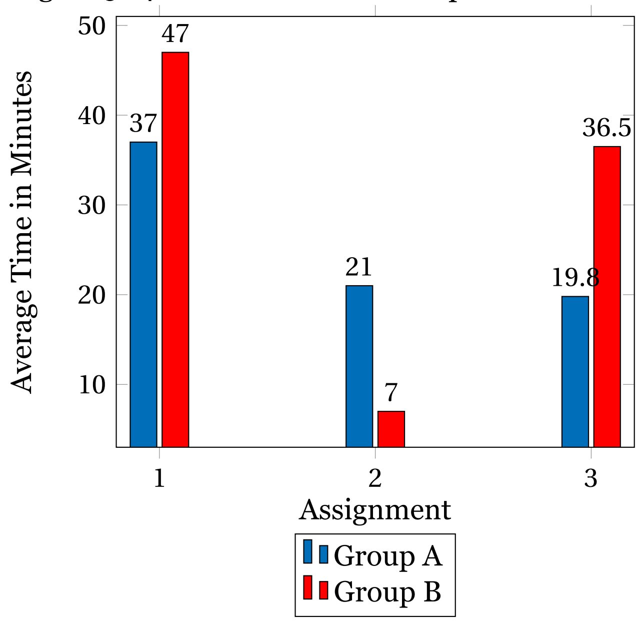


Figure 5.14: Second Run of the Experiment: Average time pr assignment



Chapter 6

Discussion

In chapter 5 we presented the results from the two runs of the experiment we have conducted. The results from the two runs vary, but the implementation of the two runs were slightly different as well. This gives us room for discussion and analysis.

In this chapter we go through and discuss the results from each assignment and the limitations of our experiment. We continue with the naming we established in section 3.5 for the groups. Group A worked on the original version, while group B worked on the refactored version.

6.1 Assignment 1

Assignment 1 is the largest assignment, and it focuses on implementing new functionality. The goal of the assignment is to see if there is a difference in the understandability and reusability of the two versions. The assignment text given in the experiment was:

"Right now this user-API does not support updating a user. Your first assignment is to implement this. Remember: The user must be authorized. A users username, email and uuid must all be unique."

Assignment 1 is explained more in detail in section 3.3.1.

6.1.1 First Run of the Experiment

The groups started very even, both finishing assignment 1 on 49 and 46 minutes respectively. Group B finished a little faster than group A as we had expected. We have to keep in mind that these results reflects both the time used to solve the assignment and the time used to find the problem area. Whether group B was better at finding the correct class or solving the assignment is impossible to say. In other words, we cannot claim that this result shows that the refactored version is more understandable. But as the results are similar, it seems likely that it is at least as understandable as the original version.

6.1.2 Second Run of the Experiment

In the second run, group A had the best average time on assignment 1 with 37 minutes. Group B used 10 more minutes to finish, even so, the fastest solution came from group B, with 25 minutes. We expected B to finish first, as in the first run, but this time we had a different outcome. Unlike the first run, this result is only showing the time used to solve the assignment, which gives it more credibility. This result suggests that it is easier to add new features to the original version.

6.1.3 Discussing Assignment 1

Our experiment had different results for this assignment. The results from the first run suggests that the refactored version is more understandable. But as we have discussed earlier, the result reflects both finding the correct class and solving the assignment. This lowers the quality of the data, and we cannot prove that clean code made the system more understandable with this data.

In the second run of the experiment we fixed this problem, and now the original version had a better finishing time. This result has more quality and credibility to it than the results from the first run, as this only reflects the time used to solve the assignment.

So why is this? Why did participants finish faster on the version we thought to be more complicated and less understandable?

One reason could be that the participants working on the original code feel that they do not have to spend time writing good code, due to the state of the existing code. This makes them write a quick and dirty solution, that matches the quality of the code already in the system. Group A also had more solutions with errors, which could point to the fact that the solutions are not as thought through as they should be. This mentality resembles the characteristics of the "Broken Window Theory" we mentioned in chapter 1.

This mentality works in this experiment, as the whole point is to solve the assignment as fast as possible and then never work with the code again.

But in a project where the developer works with the same code over months or years, these quick and dirty fixes hurt the maintainability of the system.

The refactored version on the other hand shows signs of someone caring about the code. This might make the participants more reluctant to create a quick and dirty solution, and in the long run this will lead to more maintainable code. In the refactored version, one wants to make sure that there is a function available that could be used to solve the assignment, before creating it. When the code is built as it is with a lot of functions solving different problems, one would want to use as many as possible so code is not duplicated.

This makes the participants spend more time looking around in the system and the class they are working in. They also might care more about their code, putting in some extra minutes to make their code look good. Since we collected the code from the second run of the experiment, we can clearly see that most of the participants in group B tried to write their code in the same manner as the system. In other words, Clean Code encourages the participants to write code with higher quality, and more maintainable code, as we discussed in section 2.5.

Another reason for the participants finishing faster on the original version could be experience. Most of our participants were students, and they might not be experienced with larger systems or Clean Code. Without this experience they could be more accustomed to long functions. If this is the case, they should be more effective on the original version, while working on the refactored version could be challenging. Unfortunately we have no data on the participants background and experience. If we have had a questionnaire about background and preferences we could have been able to tell if this is a factor.

To conclude, our research suggests that Clean Code does not decrease the time used to implement new significant functionality. It does however lead to code with higher quality as defined in section 2.4. So in larger projects where maintainability is important, the increase of implementation time when writing Clean Code could be beneficial due to the increase in both quality and maintainability. While in smaller projects, with tight schedules and less need for maintainability, Clean Code might not be necessary.

Everyone writes code differently. Developers have their own individual preferences and styles when writing code. If our participants were unaccustomed to the style of Clean Code this could also explain why it increased the time to implement functionality. Since this research focuses on the immediate effect of Clean Code, we cannot know if development time decreases when a developer becomes more comfortable with Clean Code. A study over a longer period could be more suited to research this.

6.2 Assignment 2

Assignment 2 changes the focus over to changing current functionality. The result of the assignment reflects how changeable and understandable the code is. The assignment text given in the experiment was:

"Almost every function takes a parameter called "userId". Right now the API treats this as a UUID. Your assignment is to change the functionality so all of these functions also try to fetch a user by username if it fails to find a user by UUID. In other words, after this change "userId" could also be a username."

Assignment 2 is explained more in detail in section 3.3.2.

6.2.1 First Run of the Experiment

Group B finished assignment 2 with an average time of 8 minutes. 15 minutes faster than A with an average time of 23 minutes. This result suggests that Clean Code is easier to change, as we had expected.

6.2.2 Second Run of the Experiment

Assignment 2 in the second run of the experiment had a similar result as the first run. Group B finished much faster than A. B spent 7 minutes on average, while A spent 21 minutes on average. The difference is almost as big as the difference in the first run. The fastest solution came from group B, with 2 minutes.

Group B's fastest participant must have noticed the *findUser* function during assignment 1 and done a quick search and replace. The result is as we expected, group B finishing faster than group A.

6.2.3 Discussing Assignment 2

In this assignment group B already has a function doing exactly what the assignment asks for, see listing B.6. This gives them a slight edge, but not a considerable one as it should not take group A too long to write it. After group A writes the function solving the assignment they are on equal grounds, even so the time difference is rather large. This means that there might be more to why there is such a great gap.

So what is causing this great difference? That group B does not have to write the function is of course a factor, but what might matter more is that the function is giving away the answer. When they see the function they immediately understand what the assignment is asking for, as the function answers all of their questions about the assignment. So instead of thinking about a solution or how to solve it, they only have to do a search and replace. Group A on the other hand have to read the assignment and comprehend what it is actually asking them to do, and then come up with a solution. This could be where time is spent, and group B skips this entirely when they have a solution available.

Another challenge in the original version is to identify where the change is needed. The way the functions fetch a user is not standardized and it is done in different parts of the functions. This makes changing the functions more complicated and time consuming as they have to go through each function carefully, understand and find where the user is fetched. Sometimes they also have to change some of the surrounding code to solve the assignment.

In the refactored version on the other hand, this is not a problem. Here Clean Code solves all of the problems one might experience in the original version. The function clearly shows where a user is fetched because of the name of the function, and because the logic is already separated from the rest of the function they do not have to worry about the surrounding code.

Would the result have been different if group B did not have the *findUser* function? This would have forced them to spend more time on understanding the assignment, and then creating a solution. If we assume the groups are completely even on skill, this should take equally long. Next is then to implement the solution into all the functions fetching a user. As we discussed earlier, this is much easier in the refactored version and group B should spend less time on this.

In conclusion, our research suggests that Clean Code decreases the time used to change current functionality, when the logic needed is available. But as we just discussed, even without the logic Clean Code should decrease the time.

6.3 Assignment 3

Assignment 3 is about bug fixing. There is no implementation of new code, so the assignment tests how readable and understandable the two versions are. The assignment text give in the experiment was:

"There is a bug preventing users from being created with the "createUser" function. Your assignment is to find this bug."

Assignment 3 is explained more in detail in section 3.3.3.

6.3.1 First Run of the Experiment

In the final assignment group A finished with an average time of 8 minutes, while B spent 35 minutes on average. We were not expecting this major difference. We were prepared for the groups to be fairly even, but that was just not the case.

As we discussed in section 3.5.4 the participants in group B experienced some problems with the assignment text, which is one of the reasons they spent so much more time solving the exercise. The first participant to start on assignment 3 was from group B, and he had only worked for 30 minutes, 24 minutes on the first assignment and 6 on the second. But then he spent 66 minutes looking for the bug, 30 minutes longer than anyone else, before he had to quit. He was one of the participants who misread the assignment and only looked "inside" *createUser*, and did not explore the function calls.

This is not the result we expected, but the problems we experienced with the assignment text could have affected our result.

6.3.2 Second Run of the Experiment

Up until this assignment both groups have had the best average times on one assignment each. Group A finished assignment one fastest, while group B finished assignment two first. This assignment proved to be a lot more difficult and time consuming for group B than we had initially thought after changing the assignment text. Group B finished with an average time of 36 minutes, 16 minutes behind A.

The fastest solutions came from both groups, with 6 minutes each. This Group B did better than the group B we had in the first run of the experiment, so changing the assignment text might have made it clearer for them where to look. Even so, the result is not as we had expected.

6.3.3 Discussing Assignment 3

In the first run of the experiment the assignment text caused some problems, and made the results questionable. Many of the participants working on the refactored version misunderstood the text and thought that there was a bug in just the *createUser* function. At first they did not explore the functions called, they only looked inside *createUser*. This did not lead them to a correct answer, as the bug is inside one of the functions called. The result mostly reflects the time they spent on this before realizing that the bug was not "inside" *createUser*, and started going through the calls. Unfortunately three of the participants had to leave before finding the bug.

They spent the majority of assignment 3 looking at the ordering of the function calls.

In the second run, the assignment had a new description and none of the participants mentioned that the text was confusing or misleading in any way. Even so it is fair to suspect that group B spent some time looking, and thinking about the ordering of calls in *createUser*. This applies to all the other functions they go through as well. Essentially this means that group B naturally spent more time looking for the bug because they had to understand the logic and spot possible logical errors, while also considering the order of calls. Group A does not have this problem, because for them *createUser* is only one large function.

Another advantage of looking for the bug in the original code is that all the logic is inside the function. Because of this, the participants know the bug is somewhere between line X and Y. This allows them to start at line X and slowly go through each line carefully looking for anything that looks out of place, wrong or something that does not make any sense. Even though the function is rather large, it is not overwhelming when utilizing this kind of approach to solve the problem.

The refactored version of the function on the other hand is nothing but function calls. This might give the participants a feeling of being lost, because they do not know where to start. They may also feel they have to go through a lot of code since some functions call other functions bringing them deeper into the code. Some might lose track of the logic, and get confused or feel overwhelmed.

In the original code the bug is out in plain view, it is right there in the open. While in the refactored version it is hiding much deeper into the code. The participants have to explore two function calls to get to the right function. Even so, one of the best times on this assignment is six minutes, and it was achieved on the refactored version. What we discussed in section 6.1.3 about experience and preference of code style also applies to this. If the participants were more accustomed to Clean Code, they would have more experience with debugging this code style. The fact that one of the fastest solutions was from the refactored version could suggest this.

Debugging a system can be done in many ways, and developers usually have their own ways of doing it. Some might like to use a debugging tool, others rely on running the system, forcing the bug to occur and then read logs for information. Other approaches might be equally viable. Because the participants are not able to run the system, some might be taken out of their comfort zone, and find it difficult to claim that something is a bug. They are however able to run the unit tests created in the refactored version, but this is something no one took advantage of. Some mentioned that since they were told not to build the system they did not think of unit tests at all. This was an unfortunate misunderstanding. When we said there was no point in *running* the system, we did not mean to imply that they could not *build* it.

As we discussed in section 6.1.3, individual preference and experience affects this assignment as well. Participants accustomed to Clean Code will find it easier to locate the bug than the ones who are not. A study over a longer period of time could also be conducted to research this.

If we assume that all the participants were accustomed to Clean Code, our result indicates that Clean Code does not improve the time to find and solve bugs. On the other hand, if we assume the opposite, that the participants were unaccustomed to Clean Code, our results suggests the same but with practice it could improve the time. As most of our participants were students it is a possibility that they had little experience with Clean Code.

6.4 Implications For Practice

Our results suggests that Clean Code does not improve understandability, but what do the results *mean*? We acquired the results through conducting an experiment, where the time spent on each assignment was measured. The participants were also aware of their time being recorded. In addition, they knew they were never working with this code again. This creates a competitive environment, where the participants want to finish as fast as possible. This in turn promotes quick and dirty solutions, and this was not our intention. We wanted an environment where the participants felt like they were working on something important, an environment close to a development project.

Taking this into consideration, we can say that our results mean: When solving assignments and the goal is to finish as fast as possible, and never work with the code again, it is faster to do so on code resembling the one we used. That being code with large and long functions.

So perhaps the effect Clean Code has on understandability varies according to the size of the project. In smaller, shorter projects the effort needed to keep the system clean might not be worth it as it already is understandable enough. With all the logic in one place, one does not have to understand how all the smaller functions, as in Clean Code, work together. Abstraction is also kept at a minimum, making it less complex. The results from assignment 1, and 3 suggest this.

In larger, lengthier projects, where maintainability is important, the effect of Clean Code might be more apparent and valuable. Clean Code also encourage writing good code, so the quality of the project is kept. So in the long run, Clean Code could be valuable. Over a longer period developers also become more accustomed to Clean Code which could further increase the value. The results from assignment 2 suggest this, but more research would have to be done.

6.5 Comparing our Result to Previous Study

In section 2.6.3 we presented a newly published study[3], by Erik Ammerlaan, Wim Veninga and Andy Zaidman, very similar to ours. The major difference is the size of the study. It is larger in every aspect, with larger refactorings, multiple experiments and more participants. Despite having a smaller study, we have reached similar conclusions.

When discussing the results from one of the assignments on a smaller refactoring, they argue that: "The flow of method arguments and return values might have been more difficult to understand than a linear flow in a large method."[3] This is very much in line with what we read from the result of assignment 3, where the original code proved to be more understandable.

They further discuss this by suggesting that the habits of the participants could explain why the original code is more understandable. What lends their argument weight is that all of the participants are employed in the same software company, and work with the original code on a daily basis. In our case we argued that experience with Clean Code and larger systems could be a factor. Our participants were mostly students and they might not be comfortable with larger code bases and the style of Clean Code.

From the results of assignment 1 we experienced that the participants on the refactored version produced solutions with higher quality than the other group. We proposed that Clean Code encourages developers to write code with higher quality, leading to better maintainability. Erik et al. points to similar findings in one of their experiments.

Unlike our study, they acquired data on the benefits of unit tests. Two of the participants wrote unit tests to identify the bug in the assignment. As expected they quickly found the bug and solved the assignment. It is only two out of thirty participants, but it still shows the power of having testable code.

Even though our study is smaller we have gathered data suggesting much of the same as the larger study of Erik et al. This shows that a long term study is needed to clearly see the effects of refactoring or Clean Code on understandability. A similar study to what we presented in section 2.6.2, where instead the time used on user stories is measured could be ideal.

6.6 Limitations

We have identified some limitations to our experiment. Here we present these limitations and discuss what they mean for our results.

6.6.1 Quantitative Problem

In section 2.6.1 we mentioned that previous studies on the effect of refactoring on code quality suffer from a quantitative problem. This experiment suffers from the same problem.

We have a small number of assignments and only one system. When we only have three different assignments on one system, we cannot necessarily generalize the results. These results only represent Clean Code's effect on this particular system.

To avoid this problem we would have had to refactor multiple systems instead of just one. Create new assignments for each system, and finally conduct an experiment for every system refactored. We felt this was too much work for a master thesis and decided instead to focus on only one system.

6.6.2 The Participants

We had a total of 18 participants, 9 in each group. This low number is something that can affect the results, as individual skill has a greater impact on the average time when the sample size is small. We divided the participants randomly to their groups, in an attempt to evenly spread the individual skills. Judging from our results it looks like this helped, as the participants have fairly consistent results within each group. However, all of our participants could be significantly better or worse than the norm.

Furthermore 14 of our 18 participants were students. This means that our sample mostly represents students, and not all developers in general. Even so, the students had similar results to the four developers who participated.

6.6.3 Is Assignment 2 Fair?

Assignment 2 can be viewed as unfair and that it favors the refactored version too much to be used as an assignment. The argument is that the refactored version contains a function which does exactly what the assignment asks for. All the participants have to do is to implement the usage of this function.

The original version on the other hand does not have a function solving the assignment ready to be used. But the logic required is still present. The only difference is that it is part of a larger function, the *getUser* function(listing A.3). In other words, it is just not that visible in the original version compared to the refactored version, where the logic is extracted into its own function. And this is one of the reasons to follow the Clean Code principles, so that logic can easily be reused and found when needed.

Because of this we believe that the assignment is completely fair and can be used.

6.6.4 Not a Real Working Environment

The system cannot run. This makes it more difficult for the participants to understand the system and what the code is trying to do. To see the system run, to see what the code is doing helps a lot when working with something unfamiliar. As the participants are unable to do this, all they have is the code itself. This creates a working environment which is not very realistic. An environment such as this could make the participants feel like they are out of their comfort zone, and their performance could suffer as a side effect. If this were the case, both groups had the same handicap and it should not affect the results.

Chapter 7

Conclusion

The goal of this research was to investigate the effect of Clean Code on code understandability. To determine this, we acquired a system with many of the same characteristics as that of legacy code, and code smells. A class in this system was refactored to follow the principles of Clean Code. We then conducted an experiment, where two groups of participants solved assignments on separate code versions. One group worked on the original code, the system in its original form, and the other group on the version we refactored.

We continue with the naming we established in section 3.5 for the groups. Group A worked on the original version, while group B worked on the refactored version.

7.1 Answering the Research Questions

In this section we conclude this thesis. Before we conclude we have to answer our research-questions. We start by answering the sub-questions before we finish with the main research question.

Will Clean Code improve the time used to implement new significant functionality?

According to our results Clean Code does not improve the time used to implement new significant functionality. The first run of the experiment showed that Clean Code improved the time, but the result contains noise and cannot be relied upon to answer the question. The second run on the other hand produced more trustworthy results, which can be used. Group

A finished the assignment with an average time of 37 minutes. 10 minutes faster than group B.

It is these results we use when concluding that Clean Code does not improve time used. Indeed, it seems to increase it.

Will Clean Code improve the time used to change current functionality?

Our results clearly point to this being true. In both runs of the experiment, group B finished with a much better average time than group A. In run 1, 15 minutes faster, and in run 2 14 minutes faster. Clean Code seems to improve time used when changing current functionality.

Will Clean Code improve the time used to find and solve a bug?

The results indicate this to not be true. Group A finished with a better average time in both runs of the experiment. In run 1 they finished 27 minutes faster, and in run 2 16,7 minutes faster. Clean Code does not improve the time used to find and solve a bug. On the contrary, it seems to increase it.

Now that we have answered our sub-research questions, we can use these answers to answer our main research question.

Will Clean Code improve the time used to solve small coding tasks?

Before starting this research we expected the results to show that Clean Code improves the time used to solve small coding tasks, thus the code being more understandable. The reason for these expectations is that understandability is one of the acclaimed benefits of Clean Code. Code following the principles should be self explanatory with small functions, descriptive names and good abstraction.

Despite our expectations, the results from the experiment show that we were wrong. Only one of three sub-research questions were answered in favor of Clean Code. Both implementing new functionality, and finding and solving a bug were completed faster on the original system. With this result we cannot claim that Clean Code does improve the time used to solve small coding tasks. In other words, there seems to be no immediate benefit of Clean Code in form of understandability.

Our research points to other benefits than understandability being achieved, such as maintainability and code quality. The participants working with the refactored version in our experiment continued to write code following the Clean Code principles, maintaining the quality of the code. Because of this it is possible that with time, as developers become accustomed to Clean Code, the time used on small coding tasks will decrease.

The size of the system could also have an impact on Clean Code's effect on understandability. The effect of Clean Code could be more apparent in larger systems, as there is more code, modules, classes and interfaces to comprehend and understand. Our research was only done on a small part of a system where this was not something the participants had to consider.

7.2 Future Work

Now that this thesis is concluded we list some thoughts on further work:

- In our research we refactored a class in a system and compared this to the original. This limited the assignments to just this part, and other parts of the system were not touched by the participants. It would be interesting to compare an entire refactored system to the original. This way the assignments could spread over several parts of the system. Data from such a research could show if Clean Code is more valuable in larger systems.

- We discussed how experience with Clean Code can be a factor in section 6.1.3, and a long time study could answer this. Doing the study we described above over a longer period would give developers time to become accustomed to Clean Code. We believe that with experience developers would be able to utilize the benefits of Clean Code better. A study such as this could also tell us if Clean Code does maintain the code quality, and the effect Clean Code has on larger, more complex assignments.

Appendix A

Original Code

All of the code presented in this appendix is presented with permission from the owner. To reuse or copy the code, permission must be granted by the owner.

A.1 Create User

Listing A.1: Original: Create User

```
public User createUser(@AuthLimit(required = false) Subject subject, User newUser,
        @HeaderParam("X−Forwarded−For") String ip) {
    logger.debug(append("method", "createUser"), "subject={}", subject);

    checkUserFields(newUser);

    String password = null;

    if (newUser.getBase().has("password")) {
        password = newUser.getBase().get("password").getAsString();
    } else {
        logger.info(append("method", "createUser"), "Request without a password");
        throw new ValidationException(new Error("Validation", "ValidationException",
                "Sorry, you are missing a password!"));
    }

    Validator.throwIfInvalidPassword(password);

    UUID uuidGenerated = UUID.randomUUID();
    String uuid = uuidGenerated.toString();

    newUser.setUUID(uuid);

    long currentTime = System.currentTimeMillis();

    newUser.setCreated(currentTime);
    newUser.setModified(currentTime);
    newUser.setActivated(true);
    newUser.setEmail(newUser.getEmail().toLowerCase());

    logger.debug(append("method", "createUser"), "createUserOk subject={}, email={},
        username={}",
        new Object[] {subject, newUser.getEmail(), newUser.getUsername()});

    if (newUser.getBirthday() != null && newUser.getBirthday().after(new Date())) {
      logger.info(append("method", "createUser"), "Invalid birthday, is larger than
          today!");
      throw new ValidationException(new Error("Birthday is not valid!"));
    }

    if (dao.getUserByEmail(newUser.getEmail(), false) != null) {
      logger.info(append("method", "createUser"), "Email is already in use!");
      throw new ValidationException(new Error("duplicate_unique_property_exists",
          "Unique check failed", "This email is already in use!"));
    }

    if (dao.getUserByUsername(newUser.getUsername(), false) != null) {
      logger.info(append("method", "createUser"), "Username is already in use!");
      throw new ValidationException(new Error("duplicate_unique_property_exists",
          "Unique check failed", "This username is already in use!"));
    }

    AuthUser au = new AuthUser();
    au.setUuid(newUser.getUuid());
    au.setCreated(currentTime);
    au.setModified(currentTime);

    JsonElement je = newUser.getBase().get("password");
    newUser.getBase().remove("password");

    au.updatePassword(je.getAsString());

    if (ipLocationLookup != null) {
      ip = IPLocationLookup.splitIPHeader(ip);
      JsonObject locationObject =
          LocationMetadata.createLocationMetadata(ipLocationLookup.lookup(ip));
      newUser.addMetadata(LocationMetadata.METADATA_NAME, locationObject);
    }


    dao.saveAuthUser(au);
    dao.createUser(newUser.getUuid(), newUser);

    logger.debug(append("method", "createUser"), "return id={}", newUser.getUuid());

    try {
      AuthenticateResponse access = logInUser(newUser, password, false);
      String token = access.getAccessToken();
      newUser.setLoggedInToken(token);
    } catch (Exception e) {
      logger.warn(append("method", "createUser"), "Logging in user after creation
          failed!", e);
    }

    newUser.setShowMetadata(MetadataVisibility.OWNER);
    return newUser;
  }
```

One of the largest functions in the API, and a good example of a function doing more than one thing.

A.2 Update User

Listing A.2: Original: Update User

```java
public User updateUser(@PathParam("userId") String userId, @AuthLimit Subject
        subject, User updated) {

  String uuid = (String)
        subject.getSession().getAttribute(CouchbaseRealm.USER_UUID_PROP);

  logger.debug(append("method", "updateUser"), "uuid={}, userId={}", uuid, userId);

  Validator.throwIfInvalidUUID(userId);

  if (updated == null) {
    logger.info(append("method", "updateUser"), "Missing user entity, returning
          blank");
    throw new ValidationException(new Error("User entity missing!"));
  }

  if (uuid.equals(userId) || subject.hasRole("admin")) {
    User oldUser = dao.getUserByUUID(userId);

    if (oldUser == null) {
      logger.info(append("method", "updateUser"), "User trying to edit non-existing
            user!");
      throw new ValidationException(new Error("User not found!", "User not found!",
            "User not found!", 404));
    }

    User tempUpdated = dao.getUserByUUID(userId);
    tempUpdated.clone(updated);

    updated = tempUpdated;

    if (updated.getBirthday() != null && updated.getBirthday().after(new Date())) {
      logger.info(append("method", "updateUser"), "Invalid birthday, is larger than
            today!");
      throw new ValidationException(new Error("Birthday is not valid!"));
    }

    if (!oldUser.getUuid().equals(updated.getUuid())) {
      logger.warn(append("method", "updateUser"), "User trying to change uuid!");
      throw new ValidationException(new Error("Invalid action!"));
    }

    if (oldUser.getUsername() == null ||
          !oldUser.getUsername().equals(updated.getUsername())) {
      logger.debug(append("method", "updateUser"), "changed username");
```

```

        User existingUser = dao.getUserByUsername(updated.getUsername(), false);
        if (existingUser != null) {
          logger.info(append("method", "updateUser"), "Username not available!");
          throw new ValidationException(new Error("Requested username is already in
               use!"));
        }
      }

      logger.debug(append("method", "updateUser"), "uuid={}, userId={}", uuid, userId);

      dao.updateUser(updated);

    } else {
      throwUnauthorizedException();
    }

    updated.setShowMetadata(MetadataVisibility.OWNER);
    return updated;
  }
```

One of the largest and most complex functions in the API. This is mostly due to the fact that it does more than just updating a user. It validates several values and authorize before it actually updates the user.

A.3 Get User

Listing A.3: Original: Get User

```
  public User getUser(@PathParam("userId") String userId, @AuthLimit Subject
       subject) {

    String uuid = (String)
         subject.getSession().getAttribute(CouchbaseRealm.USER_UUID_PROP);

    logger.debug(append("method", "getUser"), "uuid={}, userId={}", uuid, userId);

    if (!Validator.validUUID(userId)) {
      User u = dao.getUserByUsername(userId, true);

      if (u == null)
        Validator.throwIfInvalidUUID(userId);

      userId = u.getUuid();
    }

    if (uuid.equals(userId) || subject.hasRole("admin")) {
      User user = dao.getUserByUUID(userId);
      logger.debug(append("method", "getUser"), "user by uuid={}", user);
      if (user == null) {
        user = dao.getUserByUsername(userId, true);
        logger.debug(append("method", "getUser"), "user by userId={}", user);
      }
      if (user != null) {
            return user;
        } else {
            logger.info(append("method", "getUser"), "Could not find user with userId={}",
                    userId);
        }
    } else {
        throwUnauthorizedException();
    }

    return null;
}
```

A fairly simple function cluttered with if statements and null checks due to the fact that the parameter *userId* can either be UUID or username.

A.4 Get a User By UUID

Listing A.4: Original: Get a User By UUID

```
User oldUser = dao.getUserByUUID(userId);
if (oldUser == null) {
  logger.info(append("method", "updateUser"), "User trying to edit non−existing user!");
  throw new ValidationException(new Error("User not found!", "User not found!",
       "User not found!", 404));
}
```

One of the most repeated pieces of code in the API and one of the most easy to refactor into its own function. Almost every function in the API operates on a user, so fetching one is necessary.

A.5 Authorize User

Listing A.5: Original: Authorize User

```
 if (uuid.equals(userId) || subject.hasRole("admin")) {
   //Some logic
 } else {
   throwUnauthorizedException();
 }
```

The authorization check done in almost every function to control that the current user in session matches the user manipulated, or that the session user is an admin.

A.6 Export Users

Listing A.6: Original: Export Users

```
public Response exportActivatedUsers(@AuthLimit(roles = "admin") Subject subject) {
  StreamingOutput stream = new StreamingOutput() {
    @Override
    public void write(OutputStream os) throws IOException,
         WebApplicationException {
      Writer writer = new BufferedWriter(new OutputStreamWriter(os));

      writer
          .append("Full name;Username;Email;Org;Primary usage;Created;Last use
               (builder);games created;games favourited;games shared;games been
               favourited;games been shared;games been played;Players on users
               games;Answers on users games\n");


      DateFormat df = new SimpleDateFormat("yyyy−MM−dd");

      UserList users = null;

      boolean finished = false;
      String lastCursor = null;

      while (users == null || !finished) {
        users = dao.getAllUsers(200, lastCursor, false);

        lastCursor = users.getCursor();

        for (User u : users.getUsers()) {
          String name = u.getName();
          if (name != null)
            writer.append(u.getName().replace(';', ' '));

          writer.append(';');
          writer.append(u.getUsername());
          writer.append(';');
          writer.append(u.getEmail());
          writer.append(';');
          String org = u.getOrganisation();
          if (org != null)
            writer.append(u.getOrganisation().replace('"', ' '));
          writer.append(';');
          writer.append(u.getPrimaryUsage());
          writer.append(';');

          Date created = new Date(u.getCreated());
          writer.append(df.format(created));
          writer.append(';');

          Date modified = new Date(u.getModified());
          writer.append(df.format(modified));
          writer.append(';');

          GameUserAnalytics gua = dao.getAnalyticsForUser(u.getUuid(), null);

          writer.append("" + gua.getMyGames());
```

```
                writer.append(';');
                writer.append("" + gua.getMyFavouritedGames());
                writer.append(';');
                writer.append("" + gua.getMySharedGames());
                writer.append(';');

                writer.append("" + gua.getMyGamesFavourites());
                writer.append(';');
                writer.append("" + gua.getMyGamesShares());
                writer.append(';');

                writer.append("" + gua.getPlays());
                writer.append(';');
                writer.append("" + gua.getPlayers());
                writer.append(';');
                writer.append("" + gua.getAnswers());
                writer.append("\n");
              }

              if (lastCursor == null) {
                finished = true;
              }
            }

            writer.flush();
          }
        };
        return Response.ok(stream).type("text/csv")
            .header("Content-Disposition", "attachment; filename=\"users.csv\"").build();
    }
```

A function used to export users. The function builds a CSV file with all activated users and other data connected to them.

Appendix B

Refactored Code

All of the code presented in this appendix is presented with permission from the owner. To reuse or copy the code, permission must be granted by the owner.

B.1 Create User

Listing B.1: Refactored: Create User

```
public User createUser(User newUser, String ip) {
   String password = newUser.getPassword();

   validateUserFields(newUser);
   setUserFields(newUser, ip);
   AuthUser authUser = createAuthUser(newUser);

   dao.saveAuthUser(authUser);
   dao.saveUser(newUser);

   firstTimeLogin(newUser, password);
   return newUser;
}
```

This is the result after extracting almost all of the logic in the original createUser, listing A.1. The result is a much smaller, tidier function, clearly showing the process of creating a user.

B.2 Update User

Listing B.2: Refactored: Update User

```
public User updateUser(String userId, AuthorisableUser authorisableUser, User
        updatedUser) {
    String uuid = authorisableUser.getUserUuid();
    logger.debug(append("method", "updateUser"), "uuid={}, userId={}", uuid, userId);
    throwIfNotAuthorized(uuid, userId, authorisableUser);

    User oldUser = getUserByUUID(userId);
    validateUpdatedUserFields(oldUser, updatedUser);

    oldUser.clone(updatedUser);
    dao.updateUser(oldUser);

    oldUser.setShowMetadata(GenericBaseWithMetadata.MetadataVisibility.OWNER);
    return oldUser;
}
```

After extracting almost all of the logic in the updateUser function this is the result. The original code is listed in listing A.2.

B.3 Get User

Listing B.3: Refactored: Get User

```
public User getUser(String userId, AuthorisableUser authorisableUser) {
    String uuid = authorisableUser.getUserUuid();
    logger.debug(append("method", "getUser"), "uuid={}, userId={}", uuid, userId);
    User user = findUser(userId);

    throwIfNotAuthorized(uuid, user.getUuid(), authorisableUser);

    return user;
}
```

When both the authorization and finding a user is extracted into their own functions this is what is left of getUser.

B.4 Get a User By UUID

Listing B.4: Refactored: Get a User By UUID

```java
private User getUserByUUID(String userId) {
    User user = dao.getUserByUUID(userId);

    if(user == null) {
        logger.info(append("method", "getUserByUUID"), "Could not find user with
                userId={}", userId);
        Validator.throwIfInvalidUUID(userId);
        throw new ValidationException(new Error("User not found!", "User not found!",
            "User not found!", 404));
    }

    logger.debug(append("method", "getUserByUUID"), "user by uuid={}", userId);
    return user;
}
```

This is the result of extracting the much used logic of trying to fetch a user-object from the database. The original is listing A.4. This refactored version is a simple method, with a null check to see if a user was found. If no user was found the function throws an exception instead of returning null.

B.5 Authorize User

Listing B.5: Refactored: Authorize User

```java
private void throwIfNotAuthorized(String sessionUUID, String userUUID,
        AuthorisableUser authorisableUser) {
    if(sessionUUID.equals(userUUID) || authorisableUser.isAdmin()) {
        logger.debug(append("method", "throwIfNotAuthorized"), "userId={} is
                authorized", userUUID);
    } else {
        logger.debug(append("method", "throwIfNotAuthorized"), "userId={} is NOT
                authorized", userUUID);
        throw new AuthenticationException(
            "You are not authenticated or missing privileges, please try again!");
    }
}
```

Fetching a user-object and authorization is done in almost every function. This is the result of extracting the authorization into a seperate function.

B.6 Find User

Listing B.6: Refactored: Find User

```java
  private User findUser(String userId) {
    User user;

    try {
      user = getUserByUUID(userId);
    } catch (ValidationException e) {
      user = getUserByUsername(userId);
    }
    return user;
  }
```

The function *getUser*, listing A.3, tries to find a user by both UUID and username. We extracted this logic into its own function. This function takes advantage of the exception thrown if a user is not found by UUID. This exception is caught and the function tries to find a user by username instead. If this also fails the exception bubbles upward.

B.7 Mixed Validation

Listing B.7: Refactored: Mixed Validation

```java
private void validateUserFields(User newUser, boolean creatingUser) {
  logger.debug(append("method", "checkUserFields"), "user={}, creatingUser={}",
       newUser, creatingUser);

  boolean changingUsername = true;
  boolean changingEmail = true;
  User oldUser = null;
  if (!creatingUser) {
    oldUser = findUser(newUser.getUuid());
    changingUsername = !newUser.getUsername().equals(oldUser.getUsername());
    changingEmail = !newUser.getEmail().equals(oldUser.getEmail());
  }

  Validator.throwIfInvalidEmail(newUser.getEmail());
  Validator.throwIfInvalidUsername(newUser.getUsername());

  if (creatingUser) {
    if (newUser.getPassword() == null) {
      logger.info(append("method", "validateUserFields"), "Request without a
           password");
      throw new ValidationException(new Error("Validation", "ValidationException",
          "Sorry, you are missing a password!"));
    }
    Validator.throwIfInvalidPassword(newUser.getPassword());
  }

  if (!creatingUser && !oldUser.getUuid().equals(newUser.getUuid())) {
        logger.warn(append("method", "validateUpdatedUserFields"), "User trying to change
            uuid!");
        throw new ValidationException(new Error("Invalid action!"));
    }

    if  (changingUsername && dao.getUserByUsername(newUser.getUsername(), false) !=
        null) {
        logger.info(append("method", "validateUpdatedUserFields"), "Username not
            available!");
        throw new ValidationException(new Error("Requested username is already in use!"));
    }

    if(changingEmail && dao.getUserByEmail(newUser.getEmail(), false) != null) {
        logger.info(append("method", "validateUpdatedUserFields"), "Email not available!");
        throw new ValidationException(new Error("Requested email is already in use!"));
    }

    if  (newUser.getBirthday() != null && newUser.getBirthday().after(new Date())) {
        logger.info(append("method", "validateUpdatedUserFields"), "Invalid birthday, is
            larger  than today!");
        throw new ValidationException(new Error("Birthday is not valid!"));
    }
}
```

This is our first attempt at refactoring the validation of fields when a user is updated or created. It is an attempt at reducing duplication as much as possible. Therefore it needs the boolean parameter *creatingUser* to decide its behavior. In the end we deemed the function too complex and decided to split it into two separate functions.

B.8 Validate User Fields After Creation

Listing B.8: Refactored: Validate User Fields After Creation

```
  private  void validateUserFields(User user) {
     if  (user. getTitle ()  !=  null)
       Validator.throwIfInvalidTitle(user. getTitle ());

     if  (user.getPassword() == null) {
       logger.info(append("method", "validateUserFields"), "Request without a
            password");
       throw new ValidationException(new
            no.cleanCoders.rest.resources.entities.Error("Validation",
            "ValidationException",
           "Sorry,  you are  missing a  password!"));
     }

     Validator.throwIfInvalidPassword(user.getPassword());
     Validator.throwIfInvalidBirthday(user.getBirthday());
     throwIfEmailInUse(user.getEmail());
     throwIfUsernameInUse(user.getUsername());
  }
```

One of the two functions we created for validation of fields. This is the one used after a user has been created.

B.9 Validate User Fields After Update

Listing B.9: Refactored: Validate User Fields After Update

```java
private void validateUpdatedUserFields(User oldUser, User updatedUser) {
  boolean changingUsername =
       !updatedUser.getUsername().equals(oldUser.getUsername());
  boolean changingEmail = !updatedUser.getEmail().equals(oldUser.getEmail());

  if (!oldUser.getUuid().equals(updatedUser.getUuid())) {
    logger.warn(append("method", "validateUpdatedUserFields"), "User trying to
        change uuid!");
    throw new ValidationException(new Error("Invalid action!"));
  }

  if (changingUsername) {
    throwIfUsernameInUse(updatedUser.getUsername());
  }

  if(changingEmail) {
    throwIfEmailInUse(updatedUser.getEmail());
  }

  Validator.throwIfInvalidBirthday(updatedUser.getBirthday());
}
```

One of the two functions we created for validation of fields. This is the one used when a user has been updated.

B.10 User Pager

Listing B.10: Refactored: User Pager

```java
public class UserPager {
  public List<UserList> pages;

  public UserPager(DataAccessInterface dao) {
    pages = new ArrayList<UserList>();
    loadPages(dao);
  }

  public void loadPages(DataAccessInterface dao) {
    boolean finished = false;
    String lastCursor = null;
    UserList users;

    while(!finished) {
      users = dao.getAllUsers(200, lastCursor, false);
      lastCursor = users.getCursor();
      pages.add(users);

      if(lastCursor == null) {
        finished = true;
      }
    }
  }

  public Iterator<UserList> getPageIterator() {
    return pages.iterator();
  }
}
```

We created this class to separate responsibility from the *exportActicatedUsers* function and other functions looping over users in the database. The function *loadPages*, loads users into an ArrayList and returns this as an iterator.

B.11 Users Exporting Own Data

Listing B.11: Refactored: Users Exporting Own Data

```
  public String exportUserData(Writer writer) throws IOException{
    DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    String data = "";

    if(getName() != null) {
      writer.append(getName().replace(';', ' '));
    }
    writer.append(";");
    writer.append(getUsername());
    writer.append(";");
    writer.append(getEmail());
    writer.append(";");

    if(getOrganisation() != null) {
      writer.append(getOrganisation().replace('"', ' '));
    }
    writer.append(";");
    writer.append(getPrimaryUsage());
    writer.append(";");
    writer.append("" + getCreated());
    writer.append(";");
    writer.append(dateFormat.format(getModified()));
    writer.append(";");
    return data;
  }
```

To separate more responsibility from the *exportActivatedUsers* function we moved the writing of data to the user-object.

B.12 CSV-Exporter

Listing B.12: Refactored: CSVExporter

```
public class CSVExporter {

  DataAccessInterface dao;

  public CSVExporter(DataAccessInterface dao) {
    this.dao = dao;
  }

  public void exportActivatedUsersToWriter(Writer writer, Iterator<UserList>
        userPageIterator) throws IOException {
    writer.append("Full name;Username;Email;Org;Primary usage;Created;Last use
        (builder);Games created;Games favourited;Games shared;Games been
        favourited;Games been shared;Games been played;Players on users
        Games;Answers on users Games\n");

    UserList users;
    while (userPageIterator.hasNext()) {
      users = userPageIterator.next();

      for (User user : users.getUsers()) {
        user.exportUserData(writer);
        GameUserAnalytics gameUserAnalytics =
              dao.getAnalyticsForUser(user.getUuid(), null);
        gameUserAnalytics.exportData(writer);
        writer.append("\n");
      }
    }
  }
}
```

We moved the entire responsibility of exporting data to a new class we called *CSVExporter*. This class can be expanded to export other data than just user-data to CSV.

B.13 Test UpdatePassword

Listing B.13: Refactored: Test UpdatePassword

```
@test(expected = validationexception.class)
  public void testUpdatePasswordThrowInvalidNewPassword() {
    password.setnewpassword("1");
    userservice.updateUsersPassword(dummyuser.getuuid(), authorisableuser,
        password);
  }
```

One of the unit test we wrote to test the new *updatePassword* function. This particular test is to make sure that users cannot update their password to an invalid password.

B.14 Test getUserWithUsername

Listing B.14: Test getUserWithUsername

```
@Test
 public void testGetUserWithUsername() {
    when(dao.getUserByUsername(dummyUser.getUsername(),
          true)).thenReturn(dummyUser);

    User user = userService.getUser(dummyUser.getUsername(), authorisableUser);
    assert(user.getUsername().equals(dummyUser.getUsername()));
 }
```

One of the unit tests we wrote when creating the *getUserByUsername* function.

Bibliography

[1] "Legacy". Def. 1.1. *Oxford Dictionaries*. URL: http://www.oxforddictionaries.com/definition/english/legacy (visited on 05/01/2016).

[2] Mohammad Alshayeb. ‘Empirical Investigation of Refactoring Effect on Software Quality’. In: *Inf. Softw. Technol.* 51.9 (Sept. 2009), pp. 1319–1326. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.04.002.

[3] Erik Ammerlaan, Wim Veninga and Andy Zaidman. ‘Old habits die hard: Why refactoring for understandability does not give immediate benefits’. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Mar. 2015, pp. 504–507. DOI: 10.1109/SANER.2015.7081865.

[4] Robert Baggen et al. ‘Standardized Code Quality Benchmarking for Improving Software Maintainability’. In: *Software Quality Control* 20.2 (June 2012).

[5] Barry W. Boehm, John R. Brown and Myron Lipow. ‘Quantitative Evaluation of Software Quality’. In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE ’76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 592–605.

[6] Kenneth S. Bordens and Bruce B. Abbott. *Research Design and Methods: A Process Approach*. 8th ed. New York, NY, USA: McGraw-Hill, 2011.

[7] Jürgen Börstler, Michael E. Caspersen and Marie Nordström. *Beauty and the Beast–Toward a Measurement Framework for Example Program Quality*. Tech. rep. 06.31. Umeå University, Computing Science, 2007.

[8] Jitender Kumar Chhabra, Krishan Aggarwal and Yogesh Singh. ‘Code and data spatial complexity: two important software understandability measures’. In: *Information and Software Technology* 45.8 (2003), pp. 539–546. ISSN: 0950-5849.

[9] Bart Du Bois, Serge Demeyer and Jan Verelst. ‘Refactoring - improving coupling and cohesion of existing code’. In: *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. Nov. 2004, pp. 144–151. DOI: 10.1109/WCRE.2004.33.

[10] Michael Feathers. *Working Effectively with Legacy Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004. ISBN: 0131177052.

[11] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2.

[12] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-61622-X.

[13] 'ISO/IEC 25010:2011 - System and Software Quality Requirements and Evaluation - System and software quality models'. In: Geneva, Switzerland: International Organization for Standardization, 2011.

[14] Sandeepa Kannangara and W. M. J. I. Wijayanayake. 'An Empirical Evaluation of Impact of Refactoring On Internal and External Measures of Code Quality'. In: *CoRR* abs/1502.03526 (2015).

[15] Kari Laitinen. 'Estimating Understandability of Software Documents'. In: *SIGSOFT Softw. Eng. Notes* 21.4 (July 1996), pp. 81–92. ISSN: 0163-5948. DOI: 10.1145/232069.232092.

[16] Jin-Cherng Lin and Kuo-Chiang Wu. 'A Model for Measuring Software Understandability'. In: CIT '06 (2006), pp. 192–. DOI: 10.1109/CIT.2006.13.

[17] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0132350882, 9780132350884.

[18] Thomas J. McCabe. 'A Complexity Measure'. In: *Software Engineering, IEEE Transactions on* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[19] Raimund Moser et al. 'Balancing Agility and Formalism in Software Engineering'. In: ed. by Bertrand Meyer, Jerzy R. Nawrocki and Bartosz Walter. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pp. 252–266. ISBN: 978-3-540-85278-0. DOI: 10.1007/978-3-540-85279-7_20.

[20] Kazuyuki Shima, Yasuhiro Takemura and Kenichi Matsumoto. 'An Approach to Experimental Evaluation of Software Understandability'. In: *Proceedings of the 2002 International Symposium on Empirical Software Engineering*. ISESE '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 48–. ISBN: 0-7695-1796-X.

[21] Konstantinos Stroggylos and Diomidis Spinellis. 'Refactoring–Does It Improve Software Quality?' In: *Proceedings of the 5th International Workshop on Software Quality*. WoSQ '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 10–. ISBN: 0-7695-2959-3. DOI: 10.1109/WOSQ.2007.11.

[22] K. Usha, N. Poonguzhali and E. Kavitha. 'A quantitative approach for evaluating the effectiveness of refactoring in software development process'. In: *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on*. Dec. 2009, pp. 1–7. DOI: 10.1109/ICM2CS.2009.5397935.