Investigating Whether Clean Code Helps Developers in Understanding a New Project

Master's Thesis

Rowan Bottema

Investigating Whether Clean Code Helps Developers in Understanding a New Project

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Rowan Bottema
born in Rotterdam, the Netherlands



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl



ALTEN Nederland B.V.
Fascinatio Boulevard 582
Capelle a/d IJssel, the Netherlands
www.alten.nl

©2019 Rowan Bottema. All rights reserved.

Investigating Whether Clean Code Helps Developers in Understanding a New Project

Author: Rowan Bottema
Student id: 4112458
Email: rowan.bottema@gmail.com

Abstract

When developers enter a project, often a vast amount of existing code exists for them to understand. Improving the understandability of the code should help them in getting up to speed. This study researches two methods that could improve the understandability of the code for newcomers: Refactoring the code to adhere to Clean Code guidelines and providing an introductory document. The effect is measured by performing a controlled experiment in which the participants are given small tasks to complete. The results show an increase in productivity for the participants working in the refactored code, and no effect for the participants who had received the introductory document. This suggests that refactoring the code can be used to aid new developers in projects.

Thesis Committee:

Chair:	dr. A.E. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	ing. V. van Gool, ALTEN Nederland B.V.
Committee Member:	dr. M.T.J. Spaan, Faculty EEMCS, TU Delft

Preface

Software developers sometimes refer to the '80-20-rule' as "The first eighty percent of the project takes eighty percent of the time and the remaining twenty percent of the project takes the second eighty percent of the time", jokingly referring to software projects often exceeding budget and time estimations. In my experience during my studies and in writing this thesis, I have found that the joke contains an undeniable kernel of truth. Nonetheless, after 'multiple eighty percents' of work, I proudly present my master's thesis "Investigating Whether Clean Code Helps Developers in Understanding a New Project". This empirical research, in the form of a controlled experiment, on the relations between Clean Code and understandability was conducted on behalf of fulfilling my Master of Science degree in Computer Science, at the Delft University of Technology.

This work would not have been present, and certainly not in its current form, without the help and support of multiple people. First of all, I would like to thank Andy Zaidman for his supervision and guidance from start to finish. Without his expertise and thorough feedback, this research could not have happened. At ALTEN, many people have supported me with indispensable advice, wisdom and expertise that have lifted this research to a higher level. In particular, I want to thank Vincent, André, Danny, Marten and Tomas for committing some of their scarce time to supporting me and this research. Last, but not least, I thank Manon for her unconditional support in often stressful times.

In this research, I was able to exercise my passion for software engineering, and learned a lot in the process. I hope you enjoy reading this thesis.

Rowan Bottema
Rotterdam, the Netherlands
May 29, 2019

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 ALTEN . . . 1
- 1.2 Research Questions . . . 2
- 1.3 Thesis Outline . . . 3

2 Background 5
- 2.1 Software quality . . . 5
- 2.2 Maintainability and understandability . . . 8
- 2.3 Quantitative assessment . . . 9
- 2.4 Qualitative assessment . . . 13

3 Experiment Design 19
- 3.1 Code base . . . 19
- 3.2 Project knowledge document . . . 20
- 3.3 Participant selection . . . 21
- 3.4 Participant distribution . . . 21
- 3.5 Resulting data . . . 22
- 3.6 Tasks . . . 25
- 3.7 Pilot experiment . . . 27

4 Refactoring 29
- 4.1 Project description . . . 29
- 4.2 Non-refactoring changes . . . 30
- 4.3 `BarnLayoutHelper` class . . . 33
- 4.4 `Area` class . . . 47

4.5 `MainViewModel` class 54
4.6 `ManureRobot` class 59
4.7 Singleton classes 64
4.8 Other refactorings done 65

5 Results **69**
5.1 Participants 69
5.2 Experiment results 75
5.3 Overall 103
5.4 Summary 107

6 Discussion **109**
6.1 Data limitations 109
6.2 Analysis of participant activities 112
6.3 Differences between tasks 116
6.4 Experimental setting 119
6.5 Document specifics 121
6.6 Effects 121
6.7 Threats to validity 123

7 Related Work **129**
7.1 General 129
7.2 Design Patterns 131
7.3 Clean Code 133
7.4 Conclusions 135

8 Conclusions and Future Work **137**
8.1 Conclusions 137
8.2 Future work 139

Bibliography **141**

A Project knowledge document **153**

List of Figures

2.1 The software quality model as defined in ISO/IEC 25010 [62] 6
2.2 Cost of change in relation to technical debt [58] 7
2.3 The TQI score categories [65] 11

4.1 Example Barn with four Areas 41
4.2 `BarnLayoutHelper` class before and the resulting classes after refactoring . . . 46
4.3 Classes `MainViewModel`, `BarnLayoutEditorModel` and `RibbonViewModel` before and after refactoring. Some members of the `RibbonViewModel` class were omitted for brevity. 58

5.1 Participants experience in professional and C# development 70
5.2 Correctness on questions on object-oriented knowledge. 73
5.3 Size preferences for class and method sizes 73
5.4 Testing frequency 74
5.5 Correctness and completion times for correct solutions for assignment 1A1. . . 78
5.6 Correctness and completion times for correct solutions for assignment 1A2. . . 79
5.7 General data for exercise 1 81
5.8 Correctness and completion times for correct solutions for assignment 2A1. . . 82
5.9 Correctness and completion times for correct solutions for assignment 2A2. . . 83
5.10 Correctness and completion times for correct solutions for assignment 2A3. . . 84
5.11 General data for exercise 2 86
5.12 Correctness and completion times for correct solutions for assignment 3A1. . . 87
5.13 Correctness and completion times for correct solutions for assignment 3A2. . . 88
5.14 General data for exercise 3 89
5.15 Correctness and completion times for correct solutions for assignment 4A1. . . 90
5.16 Correctness and completion times for correct solutions for assignment 4A2. . . 92
5.17 General data for exercise 4 93
5.18 Correctness and completion times for correct solutions for assignment 5A1. . . 95
5.19 Correctness and completion times for correct solutions for assignment 5A2. . . 96
5.20 Correctness and completion times for correct solutions for assignment 5A3. . . 98

5.21 General data for exercise 5 99
5.22 Correctness and completion times for correct solutions for assignment 6A1. . . 100
5.23 General data for exercise 6 102
5.24 Participants' opinion on code quality 102
5.25 Combined correctness and weighted completion times for the whole experiment. 105
5.26 Combined participant responses to questions after each exercise 106

6.1 The activities the participants performed while answering assignment 2A1. Only participants who answered correctly and had a recording available are shown. . 113

Chapter 1

Introduction

Writing good code is hard. Programmers do not only have to correctly translate a set of complex requirements into machine language, but also face the challenges of the result being hard (if not impossible) to visualize and being likely to have to be changed, due to shifting requirements [23]. Especially the nature of software to be ever-changing is impactful on the difficulty of software development.

While estimates vary based on differences in age and definitions, the consensus is that over half of the time spent in software development is in maintaining existing software [74, 77, 46, 18, 19, 98, 3, 45, 42, 87, 66]. A majority of this time is spent on reading and understanding existing code [46, 33, 94]. For this reason, a large portion of focus by programmers is in making their code design easy to maintain and understand, rather than implementing new features [125]. As Fowler put it: "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." [47].

When the design of the code is not cared for, the code may *decay*, meaning it becomes harder over time to make changes [106, 40, 43, 10]. Working in such 'decayed' code may drastically reduce developer productivity and morale [128] and has the potential to bring entire companies to a standstill [82]. It is therefore paramount that developers care and *clean* their code regularly.

1.1 ALTEN

ALTEN is a multinational technology consulting and engineering company. In the Netherlands, it employs more than 500 consultants, who are usually Software Engineers working at one of ALTEN's clients. There, they often work as developers on projects for a short time.

This poses a difficult challenge for the consultants. At the beginning of their assignment, they need to be able to quickly understand the project, in order to be productive as soon as possible. On the other hand, at the end of their assignment, they need to leave behind the project as clean as possible, so that the next person (be it another consultant from ALTEN, or another developer working at the client) is able understand and maintain it. These challenges are especially important for ALTEN, as failing to be productive or leaving behind a messy

code base will likely result in direct losses for the company, as consultants will not be rehired when their performance is unsatisfactory. Additionally, the consultants are the image of the company and bad performance can be detrimental to this image.

One measure ALTEN takes to tackle this challenge is training. Consultants at ALTEN receive regular training to maintain and improve their programming skills. Two of these trainings relating to object-oriented design are of particular interest.

The first is called 'Design Principles' and aims to improve the software designing skills of the participants. It claims that software design is essential in keeping software products understandable and maintainable. The core of the training consists of teaching the participants the SOLID and GRASP principles[1].

The second is called 'Code Smells & Refactoring' and aims to improve the abilities of the participants to recognize patterns of bad design, and how to solve them. It claims that software design naturally degrades when not cared for regularly and that this threatens code quality and programmer productivity. The core of the training consists of teaching the participants the concepts of software rot, technical debt, code smells and refactoring[1].

The contents of these trainings are based on popular literature on object-oriented design and have been around for a long time. The SOLID principles were popularized in 2003 by Martin [81], the GRASP principles as early as 1998 by Larman [72]. The technical debt metaphor was coined by Cunningham in 1993 [34] and the concepts of code smells and refactoring are the contents of Fowler's book published in 1999. Additionally, principles laid out by Martin in 2009 [82] are used widely in the training on code smells & refactoring.

Another measure ALTEN takes is to encourage consultants to organize regular 'knowledge evenings', where knowledge is shared and discussed in an informal setting. One of these recurring events are 'Uncle Bob' sessions. In these, the attending consultants watch one of Robert C. Martin's Clean Coders videos[2] and discuss its contents afterwards. The content of these videos are very close to Martin's books. In particular, the contents are very similar to his book on Clean Code [82].

To summarize, developers at ALTEN need to have the skills and knowledge to perform well, which is especially noticeable at the start and end of their projects. ALTEN tries to achieve this by teaching the developers specific skills on object-oriented programming and design. While the taught concepts have stood the test of time, ALTEN wishes to know if there is empirical evidence for their benefits and if they can improve on this.

1.2 Research Questions

In this research, we focus on the challenge of knowledge transfer, made clear in the previous section: the developer needs to be able to leave behind a situation in which the next developer can quickly understand the software and be productive. The material used as the basis for ALTEN's training will also be the foundation of this research. In particular, we follow the guidelines of Martin's book on Clean Code, because of its popularity, compatibility with the contents of ALTEN's trainings and close match with a focus on understandability.

[1] These concepts will explained further in Chapter 2.

[2] `https://cleancoders.com/`

We study this with two different approaches. The first approach is to use the guidelines of Clean Code to refactor parts of the code to be more 'clean'. The second is to provide an introductory document that introduces the system and its architecture to the developer, aiming to provide a footing that allows the developer to easier understand the system.

This leads to the following research questions:

RQ1 Does refactoring according to the guidelines of Clean Code have a positive effect on understandability?

RQ2 Does adding project knowledge in the form of an introductory document have a positive effect on understandability?

RQ3 Does both refactoring and adding project knowledge have a positive effect on understandability?

RQ1 relates to the effect of the refactoring on the understandability. It is expected that the understandability is improved, as the Clean Code guidelines propose that following them improves understandability [82].

RQ2 relates to the effect of the document on the understandability. It is expected that the understandability is improved, as the added knowledge would improve the 'baseline' understanding of the project and code.

RQ3 relates to the effect of both the refactoring and the document on the understandability, particularly to the interaction between the two. It is expected that understandability is improved, but that the interaction causes a small decrease in understandability. This is because the effects may overlap and that the knowledge of the code may cause the novelty effect or resistance to change.

1.3 Thesis Outline

The thesis is structured as follows. Chapter 2 will describe the theoretical background of the study, exploring code quality, maintainability and understandability, among others. In Chapter 3, the set up of the experiment is detailed, along with the choices made for the approach used. Next, Chapter 4 describes in detail the refactorings done for the experiment, so the reader can judge their applicability and quality. Chapter 5 presents the results of the experiment, followed by a discussion of the results in Chapter 6. In Chapter 7 earlier work is discussed and compared to this research, before concluding with Chapter 8, giving an overview of the work and answering the research questions.

Chapter 2

Background

Before we can answer the research questions, we first present an overview of some relevant background material. The aim is to establish a good understanding and define the notions of code quality, understandability and refactoring in the context of this thesis.

2.1 Software quality

Because software quality is a broad and unspecific term, many definitions exist. One such is the International Organization for Standardization (ISO) definition:

> The capability of software product to satisfy stated and implied needs when used under specified conditions [62]

In addition, a number of definition models [36] have been created that attempt to divide the notion of software quality into several characteristics [86, 20, 55, 38, 62]. They contain a fair amount of similarities, of which an overview and comparison is given by Al-Qutaish [2]. For the purposes of this thesis, we will focus on the model defined in ISO/IEC 25010 [62] (successor of ISO/IEC 9126 [63]), being an international standard.

An overview of the model is given in Figure 2.1. The 8 main quality characteristics are defined as follows [62]:

- **Functional Suitability**: The degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions.
- **Performance Efficiency**: The performance relative to the amount of resources used under stated conditions.
- **Compatibility**: The degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment.
- **Usability**: The degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

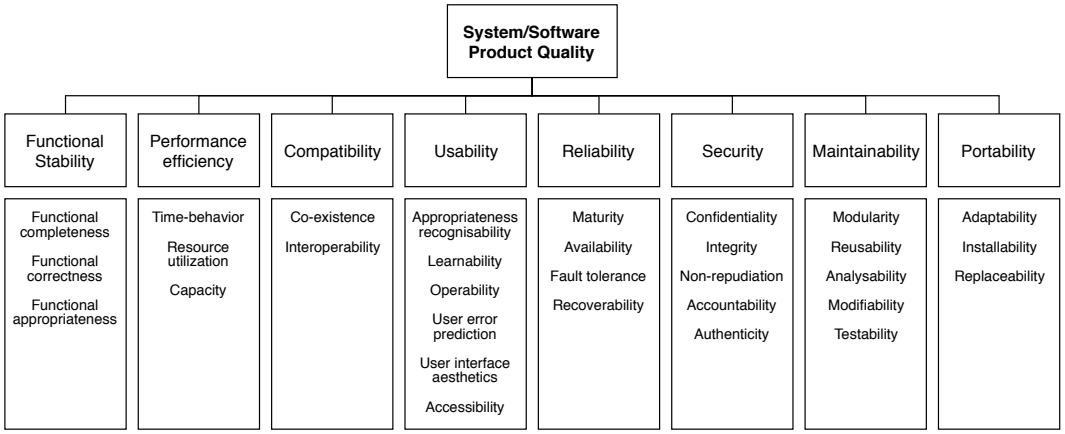


Figure 2.1: The software quality model as defined in ISO/IEC 25010 [62]

- **Reliability**: The degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.
- **Security**: The degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
- **Maintainability**: The degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers.
- **Portability**: The degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.

In this thesis, we are interested in the quality of the code. While software quality and code quality are similar terms, they do not have the same meaning. Software quality can be divided in two areas: external and internal quality. External quality regards the observable behavior of the system, while internal quality regards the code structure and architecture. The ISO/IEC 25010 definitions are given in Table 2.1. Code quality can be seen as internal software quality.

One way to illustrate the difference between external and internal software quality is to look at two metrics of software quality: faults and failures. Faults are defects in the code (i.e., internal), while failures are defects observed running the code (i.e., external). While related, they are not the same: not all faults are necessarily detected during testing as failures, and a single fault may give rise to multiple apparently different failures [62].

The impact of having low code quality can be large. In extreme cases, software failures attributed to low code quality can lead to huge financial losses, bankruptcy [82, 29] and even death [75, 70]. In general, lower code quality has a direct connection to an increased number of failures and lower developer productivity and morale [128]. When a decrease of

Table 2.1: Definitions of external and internal measures of software quality as defined in ISO/IEC 25010 [62]

Term	Definition
external measure of software quality	measure of the degree to which a software product enables the behaviour of a system to satisfy stated and implied needs for the system including the software to be used under specified conditions
internal measure of software quality	measure of the degree to which a set of static attributes of a software product satisfies stated and implied needs for the software product to be used under specified conditions

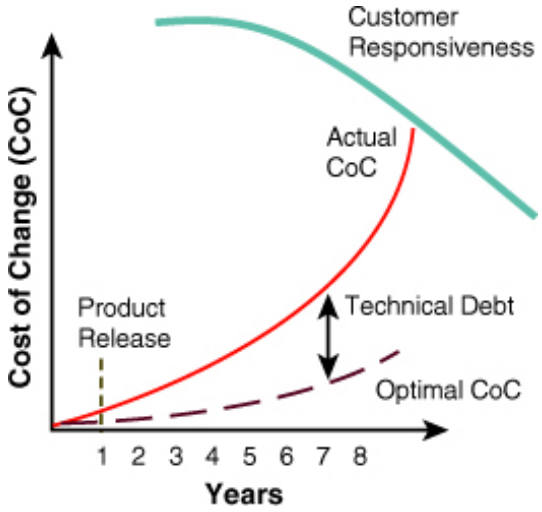


Figure 2.2: Cost of change in relation to technical debt [58]

code quality happens over time, this is referred to as software decay [106, 40] or software rot [81, 43].

An often used metaphor to illustrate low code quality is the notion of technical debt. The term was first coined in 1993 by Cunningham, who describes it as follows [34]:

> Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. (...) The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation.

The technical debt metaphor can be illustrated with a 'cost of change curve', as shown in Figure 2.2. In this image, the effect of 'stand-still' is displayed as an ever-increasing cost of change up to the point that adding new functionality or performing maintenance becomes infeasible. The difference between the actual cost of change and the optimal cost of change

is characterized as technical debt. However, unlike the image suggests, technical debt does not always have such negative effects, and may be beneficial in the short term [4].

While the metaphor originally only considered technical debt at the code level, its definition has since been expanded beyond that to include broader aspects such as architecture, design, documentation and testing [24, 128, 76]. This thesis will focus on code technical debt. In general, many studies agree that technical debt has a negative effect on maintainability [76].

2.2 Maintainability and understandability

The focus of this study is maintainability and understandability. The ISO/IEC 25010 model provides the following sub-characteristics of maintainability [62]:

- **Modularity**: The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- **Reusability**: The degree to which an asset can be used in more than one system, or in building other assets.
- **Analysability**: The degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more.
- **Modifiability**: The degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- **Testability**: The degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

Interestingly, the ISO model does not contain a definition for understandability. This exposes a gap in current research: While it is clear that good understandability is an important factor in the maintainability of software, the term lacks a definition that is agreed on by researchers. Many studies aiming to assess understandability exist, but this is rarely accompanied by a clear definition of what the authors believe entails understandability. Perhaps this is due to the apparent intuitiveness of the term: understandability describes to what extent the system can be understood, and ‘to understand’ is a common verb. This can, however, lead to small but noteworthy differences in interpretation. For example, the Merriam-Webster dictionary contains different definitions of the verb ‘to understand’ in varying degree of strength, ranging from “to grasp the meaning of” to “to be thoroughly familiar with the character and propensities of” [102].

Boehm et al. give a definition of understandability in their model [20]:

> Code possesses the characteristic understandability to the extent that its purpose is clear to the inspector.

They further identify some sub-characteristics: "[Understandability] implies that variable names or symbols are used consistently, modules of code are self-descriptive, and the control structure is simple or in accordance with a prescribed standard, etc." [20].

The closest definition in the ISO model is analysability [64]. An important distinction however, is that analysability in the context of the ISO model can refer to both manual or automatic analysis, while the notion of understandability generally applies to manual analysis only. An example where there is a significant difference between the two is in the case of reflection, which is generally hard to analyze automatically, but human analysis is not impacted as much.

Another similar term is that of program comprehension. One definition, given by Maalej et al., is as follows: "program comprehension is the activity of understanding how a software system or a part of it works." [80]. While the definition seems very similar to the definition given by Boehm et al., an important distinction can be made: program comprehension is the human process of understanding the software, while understandability is a property of the software indicating to what extent the human can understand it. Thus, it seems the terms differ in their perspective: one is human-centered, while the other is code-centered. However, this distinction might not be clear at first glance, and it is unclear if all research use the same definition.

The existence of different terms for the same concept, in addition to the lack of common definition, is a problem for research in this area. Researchers looking for earlier research might have trouble finding all related previous work, due to different terminology. In addition, the lack of common definition hurts the comparison between studies as it is not always clear if what was studied was the same.

In this thesis, understandability is measured by both the completion times and correctness percentages of the tasks during the experiments. This is further elaborated on in Chapter 3.

2.3 Quantitative assessment

With the definitions and impact of code quality clear, we are interested in assessing the quality of the code. Preferably, we do this using quantitative, objective measures. Quantitative measurement of code quality is done using metrics. Since they first saw use around the 1970s [44], many metrics have been defined over the years with more being proposed every year [99].

One of the simplest and arguably one of the oldest metric is Lines of Code (LOC). While the size of a code unit or system can be seen as surrogate measure for effort, functionality, and complexity, more advanced metrics are needed to better measure this [44]. Early examples of more advanced metrics, which are still actively used today, are the Cyclomatic Complexity by McCabe [85] and the complexity measures by Halstead [56].

The Cyclomatic Complexity (also called the McCabe Complexity) was defined by McCabe by representing programs as graphs, where every statement can be considered a node and code paths can be considered edges [85]. For example, an `if` statement can be represented by a single node with two outgoing edges, representing the paths the program will

take when the evaluation of the statement is `true` or `false`. Defining the graph G with n vertices, e edges and p connected components[1], the Cyclomatic Complexity $v(G)$ becomes:

$$v(G) = e - n + 2p$$

An alternative interpretation of the Cyclomatic Complexity is to regard its value as "the number of binary decisions in a program plus 1" [67].

The complexity measures defined by Halstead are commonly referred to as the Halstead metrics. Halstead defines the following primitive measures on code, where operators are operations such as + or ++ as well as keywords such as `if` and `for`, and operands are the data the operators operate on, such as identifiers and constants [56]:

$$\begin{aligned}
\eta_1 &= \text{Number of distinct operators} \\
\eta_2 &= \text{Number of distinct operands} \\
N_1 &= \text{Total number of operators} \\
N_2 &= \text{Total number of operands}
\end{aligned}$$

Halstead then defines a set of measures based on these primitives, such as the program length ($N = N_1 + N_2$), vocabulary ($\eta = \eta_1 + \eta_2$), volume ($V = N \times log_2 \eta$) and difficulty ($D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$).

In 1994, Chidamber and Kemerer presented a suite of metrics specifically tailored for object-oriented design [30], commonly referred to as the 'CK metrics' [99]. These metrics have seen widespread adoption and are the most referenced metrics in literature, along with Lines of Code and the Cyclomatic Complexity [111, 64, 99]. Examples of CK metrics are Weighted Methods per Class ($WMC = \sum_{i=1}^{n} c_i$, where n is the number of methods in the class, and c_i is the complexity[2] of method i), Depth of Inheritance Tree (DIT = The length of the path from the class to the root of the inheritance tree) and Lack of Cohesion in Methods (defined below, where C is the set of methods in the class and I_m is the set of instance variables used by method m):

$$\begin{aligned}
P &= \{(I_m, I_{m'}) \mid m, m' \in C, I_m \cap I_{m'} = \emptyset\} \\
Q &= \{(I_m, I_{m'}) \mid m, m' \in C, I_m \cap I_{m'} \neq \emptyset\} \\
LCOM &= \begin{cases} |P| - |Q| & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

While the metrics are indicators of specific properties of the code, in order to make a judgement about the code as a whole, combining them is necessary. A classic example is the Maintainability Index (MI) [101]. This metric has been developed in the late 1990s as

[1] In most applications, such as when considering a single code unit, the graph will be a single connected component, meaning $p = 1$, simplifying the formula. The p is included to be able to calculate the complexity of a collection of programs [85].

[2] Chidamber and Kemerer deliberately do not give a specific definition for complexity [30]. The McCabe complexity is most frequently used as the complexity measure in calculating WMC [22].

Category	Name	TQI Score
A	Outstanding	>= 90%
B	Good	>= 80%
C	Fairly Good	>= 70%
D	Moderate	>= 50%
E	Weak	>= 40%
F	Poor	< 40%

Figure 2.3: The TQI score categories [65]

one of the first metrics for maintainability [31]. Over time many variations have been created [136]. One such still in use and shipped with the latest versions of Microsoft Visual Studio [93] is defined as follows [91]:

$$\begin{aligned} MI = max\Big(0, \big(171 &- 5.2 * ln(\text{Halstead Volume}) \\ &- 0.23 * (\text{Cyclomatic Complexity}) \\ &- 16.2 * ln(\text{Lines of Code})\big) * 100/171\Big) \end{aligned}$$

The formula was derived by taking a number of systems written in C and Pascal, both calculating a large set of metrics on them and asking experts to give each a maintainability rating between 0 and 100. Using statistical regression, a selection of the best metrics was made and fitted to the expert opinions [31].

While the MI is still in use, its effect has not properly been determined on modern, object-oriented systems. The validation found in literature (as mentioned in e.g., [136]) was done mostly on systems that were not object-oriented. In contrast, more recent experimental evidence [120] suggests that there might be no meaningful relation between the Maintainability Index and software maintainability (when controlling for size [41]).

In addition, the MI can be seen as having many important limitations. Heitlager et al. suggest that the calculation is expensive and flawed, and that values of the metric are not easily explained and do not make it clear what steps can be taken to improve the score [57].

A larger model is the TIOBE Quality Index (TQI) by Tiobe [65]. Using ISO/IEC 25010 as a starting point, they define 8 software quality metrics [65]:

- **Code coverage**, measured by the test coverage percentage.
- **Abstract interpretation**, measured by the amount of violations found by an abstract interpreter.
- **Cyclomatic complexity**, measured by the average McCabe complexity [85].
- **Compiler warnings**, measured by the amount of compiler warnings returned by all compilers on the highest setting.

- **Coding standards**, measured by the amount of violations found by a code checker.
- **Code duplication**, measured by the number of semantically equivalent chains of 100 tokens.
- **Fan out**, measured by the amount of imports per module.
- **Security**, measured by the amount of security violations found by a code checker.

For all of these metrics, a function is applied to produce a score between 0% and 100%. These scores are then combined into a single indicator by computing a weighed average of those scores. This is called the TQI score, and is again divided in multiple categories A to F, shown in Figure 2.3. Tiobe recommends a score of A for domains such as Avionics and Defense, but a score of D is considered good enough for Administrative software [65].

Another such model is the model created by the Software Improvement Group (SIG) [57, 13]. Also using the ISO model as a basis, they compute 7 quality properties [13]:

- **Volume**, measured by the estimated rebuild value.
- **Duplication**, measured by the percentage of redundant code.
- **Unit size**, measured by the lines of code per unit.
- **Unit complexity**, measured by the cyclomatic complexity [85] per unit.
- **Unit interfacing**, measured by the number of parameters per unit.
- **Module coupling**, measured by the number of incoming calls per module.

These properties are given a rating between 1 and 5 stars, depending on the performance relative to a benchmark repository of systems analyzed by SIG. For example, 5 stars means the system belongs to the top 5% 'most maintainable' systems in their corpus. Interesting to note is that while the SIG and the TQI models are similar in nature, they contain some key differences in the metrics used. For example, the SIG model does not explicitly take test code into account. Instead, they aim to measure testability as an aggregate of unit complexity, component independence and volume [132].

The use of metrics for predicting software quality has been widely validated [12]. However, doubts on their predictive value exists. For example, most of the predictive power of metrics seems to disappear when correcting them for size [41, 142, 54]. Additionally, models are often not generalizable across different projects [143]. In a systematic review, Riaz et al. concluded that "Our results suggest that there is little evidence on the effectiveness of software maintainability prediction techniques and models." [112].

In a set of studies regarding four functionally identical, but differently implemented, software systems, metrics seem to disagree with empirical results on maintenance effort. The metrics concluded that one system was the worst in maintainability [16, 11], while actual maintenance activities suggest that that system was actually the best maintainable of

the four systems [120, 121]. Sjøberg et al. concluded that from the studied metrics only size was a good predictor [120].

While the two larger models mentioned above, the TQI and SIG model, come from experiences in practice, their validity is also not without question. In a study by Steneker, the author was "not (...) able to establish the validity of the TQI as an indicator of software quality." [124]. On the other hand, in a study by Bijlsma et al., the authors were able to demonstrate "a significant, positive statistical correlation between the quality of software products as measured by the SIG quality model and the speed at which issues are solved by the development and/or maintenance teams of these products." [17].

Looking at understandability specifically, little relation can be found between metrics and understandability. In a study by Scalabrino et al., the values of 121 metrics, both existing and newly defined, are compared to empirical data on the understandability of 50 Java snippets [116]. They found no correlation between the metric values and understandability in almost all cases, and very little in the others (the largest correlation found had a correlation coefficient of -0.18) [116]. A re-analysis by Trockman et al. used PCA and regression analysis to produce a model combining various metrics with limited success, but still concluded that they were "not able to present a model that assessed code understandability with high accuracy" [129]. Scalabrino et al. also concluded that other previous research on measuring code understandability suffer from serious flaws, such as not being empirically evaluated ([78, 79, 122]), considering understandability as a factor in a quality model ([1, 14]), or measuring understandability at the level of a whole system ([26]) [116].

Particularly missing from previous efforts in measuring understandability with metrics is the use of textual measures (e.g., meaningful variable and function names), as opposed using only structural measures. Intuitively, clear and meaningful naming is expected to have a large effect on the understandability of the code. Recent work has used textual features in measuring code quality with success [103, 115, 104, 117], but to our knowledge no such approach for measuring understandability exists. Therefore, we must conclude that measuring understandability using code metrics is of limited use.

2.4 Qualitative assessment

The process of understanding source code is very much a human task [134]. It therefore makes sense to look at qualitative measures to judge code quality.

Developers often have an intuitive sense of code quality ("I know it when I see it" [67, 21]. In a study by Börstler et al., the authors surveyed students, educators and professional developers about their perceptions of code quality. It found that most educators and developers indicates that they can "easily tell good from bad code", although there was weak agreement what properties represented code quality best [21]. Martin describes this intuition as 'code-sense' [82], while Fowler states that "no set of metrics rivals informed human intuition" [47].

2.4.1 Code Smells

This intuitive sense is also supported by the notion of *code smells*, introduced by Fowler [47]. On his website, Fowler defines a code smell as "a surface indication that usually corresponds to a deeper problem in the system" [48]. He further clarifies that the definition contains two important nuances: code smells are by definition quick to spot during inspection and are not always a problem, but are an indication that a problem likely exists [48].

Fowler also introduced a catalog of code smells, providing a reference for identifying smells and providing a vocabulary to ease communication about smells [47, 68]. The code smells often have names that need no further explanation, or provide a metaphor for a phenomenon. Some examples of code smells mentioned are [47]:

- **Duplicated Code**: "Number one in the stink parade": the same code structure in more than one place.
- **Long Method**, **Large Class** and **Long Parameter List**: Self-explanatory. Fowler does not provide specific cut-off points.
- **Shotgun Surgery**: Every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
- **Feature Envy**: A method that seems more interested in a class other than the one it actually is in.

2.4.2 Refactoring

In order to remedy code smells, Fowler suggests *refactoring*. Fowler defines refactoring as follows:

> 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. [47]

Two points in the definition are of special importance. First, the external behavior of the code should not change, or in other words: its functionality should remain exactly the same and when running the software, no difference should be detectable[3]. Secondly, the refactoring should improve the internal structure of the program, ideally making the code more maintainable or understandable. Herein lies the benefit of refactoring.

Fowler describes the process of refactoring in tiny steps. Unit tests are described as indispensable, as they are to be executed after each step. This, together with the small size of the steps, makes sure the functionality remains unchanged during the entire process of refactoring and that potential errors are quickly and easily spotted. Since 1999, tool support for refactoring has improved considerably, such that most modern IDEs provide support for automatically performing some refactoring, easing the process [90, 89, 25].

[3] Arguably, the performance of the system could have been improved as a result. On the other hand, it is debatable whether that still entails a refactoring. That discussion is left out of scope for this thesis.

Additionally, Fowler provides a catalog of refactorings[4], again aiding communication between developers by providing a common vocabulary. Refactorings vary in complexity, from simple refactorings such as *Extract Method* to complex refactorings such as *Substitute Algorithm* and *Tease Apart Inheritance*.

Note that the decision of when to apply what refactoring is left mostly to human judgement. Particularly, some refactorings are opposites of each other, such as *Extract Method* and *Inline Method*. This signifies that refactoring is a trade-off; different situations call for different approaches.

2.4.3 Clean Code

Analogous to the metaphor of 'smelling' code, when this smell is removed, the code can be considered 'clean'. Clean code can therefore be considered code without code smells. The use of the term dates back to at least 2000, where Mena Quintero et al. define it as follows: "Clean code is easy to read; this lets people read it with minimum effort so that they can understand it easily." [88, 39]. This solidifies our reasoning that clean code has a good fit with understandability.

In 2009, Martin released his book on clean code [82]. In it, he does not give a single definition for clean code (stating: "There are probably as many definitions as there are programmers" [82]), instead discussing the definitions of a few other prominent programmers. Recurring themes that can be extracted from these definitions is that clean code contains meaningful names and should be simple, elegant, expressive, and readable to developers other than its original author. Perhaps the most striking definition is given by Cunningham [82]:

> You know you are working on clean code when each routine you read turns out to be pretty much what you expected.

Martin's book on Clean Code contains many principles for detailing what clean code is. Some of them are [82]:

- **Meaningful names**: One of the recurring themes in Clean Code is the important of naming. Some principles mentioned for creating good names are using intention-revealing names (such as `elaspedTimeInDays` as opposed to `d`) and avoiding disinformation (so that the name does not lie or obscure information).
- **Functions should do one thing**: Martin describes the rule as follows: "Functions should do one thing. They should do it well. They should do it only.". This is further detailed by explaining that functions should only perform at one level of abstraction.
- **Small**: While Martin does not give specific cut-off sizes for how small code units should be, the connecting link between all mentions of size is that developers should strive for code units to be as small as possible, extracting out functionality wherever possible.

[4] Refactoring is both used as a verb and as a noun.

Martin also included a catalog of code smells in his book [82]. It builds on Fowler's catalog, expanding it with a large amount of new smells.

2.4.4 SOLID Principles

The SOLID principles are a set of principles of object-oriented design coined by Martin [81, 83], that aims to help developers eliminate smells and build better object-oriented designs [81]. The acronym comes from the first letters of the five principles SOLID consists of. The principles and their meaning are as follows [81]:

- **Single Responsibility Principle (SRP)**: A class should have only one reason to change.
- **Open-Closed Principle (OCP)**: Software entities should be open for extension, but closed for modification.
- **Liskov Substitution Principle (LSP)**: Subtypes must be substitutable for their base types.
- **Interface Segregation Principle (ISP)**: Clients should not be forced to depend on methods that they do not use.
- **Dependency Inversion Principle (DIP)**: Highlevel modules should not depend on lowlevel modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

2.4.5 GRASP Principles

The General Responsibility Assignment Software Patterns (GRASP) are a collection of patterns and principles[5] defined by Larman [72]. In his words, GRASP "describe fundamental principles of object design and responsibility assignment, expressed as patterns" [72]. The nine patterns and principles defined by [72] are [72]:

- **Information expert**: Assigning responsibilities to the information expert: the class that has the information necessary to fulfill the responsibilities.
- **Creator**: Assigning the responsibility of creating class `A` to class `B`, if `B` either aggregates, contains, records or closely uses class `A`, or has the relevant initializing data for construction of class `A`.
- **Low coupling**: Assigning responsibilities such that coupling remains low. Coupling is the measure of how strongly one elements is connected to other elements.

[5] While Larman described GRASP a set of patterns, we use the term 'principles' to describe some of them. This is done to avoid ambiguity due to the term 'design patterns' commonly referring to the GoF Design Patterns by Gamma et al. [51] and because we believe the term principles is more fitting in common usage. This is not without precedent: Larman also regularly uses the term principles in his book to describe GRASP [72], and GRASP is often referred to as the 'GRASP principles' (e.g., [37, 125]).

- **High cohesion**: Assigning responsibilities such that cohesion remains high. Cohesion is the measure of how related the responsibilities within the elements are to each other.
- **Controller**: Assigning the responsibility of handling system events to a controller class.
- **Polymorphism**: Assigning responsibilities of related behaviors to the types for which the behavior varies. Alternatively: Giving the same name to services in different objects when the services are similar or related.
- **Pure fabrication**: Assigning responsibilities to an artificial or convenience class that does not represent a problem domain concept.
- **Indirection**: Assigning responsibilities to an intermediate object to mediate between other components or services so that they are not directly coupled.
- **Protected variations**: Assigning responsibilities to stable interfaces around points of predicted variation or instability.

Chapter 3

Experiment Design

This chapter describes the set-up and design of the conducted experiment. The goal of this experiment is to research the impact on code understandability of refactoring to improve adherence to the Clean Code guidelines, and of supplying an introductory document. In more formal terms, the dependent variable that the experiment wishes to measure is code understandability, with respect to the two independent variables (treatments): adherence to Clean Code and project knowledge.

For this purpose, a code base was selected that did not fully adhere to the principles of Clean Code (Section 3.1). In this code base, refactorings were performed to better adhere to Clean Code principles (Chapter 4), resulting in two different versions of the code: the original version, and the refactored version. Additionally, a document was written that aims to serve as an introduction to the code base (Section 3.2).

In order to measure the impact of both treatments, a group of professional developers (Section 3.3) were given short programming tasks (Section 3.6) to complete. This group was split into four different groups, in order to control for both independent variables (Section 3.4). Data is collected before, during and after the experiment (Section 3.5) in order to allow for broad analysis (Chapter 5). To catch errors and make small optimizations in the experiment, it was tested beforehand with a pilot experiment (Section 3.7).

3.1 Code base

The code base is an important part of the experiment: it is the medium on which the experiment is performed. The code base should adhere to certain criteria:

Representative The code base should be representative for industrial code in order to be able to relate the results to actual practice. As such, it is preferable that the code comes from within the industry (i.e., ALTEN). The more representative the code base is, the more broad the conclusions are that can be drawn from the experiment. Open source code is less representative for industrial code (e.g., [107]).

Size The code base should not be too small, otherwise the tasks may not be realistic. On the other hand, if the code base is very large, care must be taken to keep the tasks

small enough to keep them doable in the short time span of the experiment.

Quality In order to be able to apply sensible refactorings to the code, it must have some parts that have smells or other quality issues that would be improved by refactoring. Code that already (almost) fully adheres to the principles on which the refactoring is based, will not change sufficiently, and will thus likely not produce significant results.

Paradigm The code base should be written according to an Object Oriented paradigm, in order for the studied quality measures (in Chapter 2) to apply.

Language The language the code base is written in should be an Object Oriented language, to support the desired paradigm. Additionally, it preferably is a language a large enough group of ALTEN employees are proficient in, in order to increase the pool of potential participants. As a result, the code language should be either C++, Java or C#, each giving a potential pool of around 50 participants.

Domain The domain of the system should not have too big of an influence on the code. Examples are embedded systems, High Performance Computing and mobile apps, where considerations related to the domain might result in purposely not adhering to code quality guidelines [100, 105, 96].

Ownership Because the code is owned by the customer in many cases of industrial code, it is an important factor to take into account. For educational purposes, we want to disclose as much as we can about the code base. However, the customer might not always agree to disclosure. In such cases, compromises must be made to disclose as much as possible to preserve research integrity while adhering to the customer's wishes.

For the selection of the code base, five projects developed by ALTEN have been studied in terms of the above criteria. From these, a single project was selected and will serve as the code base for the rest of the experiment.

The project is a medium-sized C# project, developed by a single developer over the course of about five months. In general, the code can be considered clean, but still several opportunities for refactoring exist. The project is discussed in more detail in Section 4.1. During the experiment, the participants received the entire code base including unit tests.

3.2 Project knowledge document

In order to introduce project knowledge, a document is drafted. The form of a document was chosen to make sure the introduction was the same for all participants. Other measures that might be more commonplace in practice, such as an verbal introduction or sending the code so it can be explored, are less controllable and more dependent on the amount of time the participant has and is willing to spend. Similarly, the document is also kept short to reduce time strain on the participants: the estimated reading time was 15 minutes.

Available literature does not agree on the preferred contents of such a document should be for an optimal introduction [35, 114, 52, 141]. As a result, the document is written

intuitively, meaning its contents are the author's estimation of what knowledge is most useful as an introduction into the project. The contents are verified by the project developer, verifying that the document would be a decent introduction and that the contents are correct.

The document contains three sections. In the first section, the software and its purpose are introduced shortly, along with an annotated screenshot of the application in action. In addition, the section contains an explanation of the components of the GUI and their purpose.

In the second section, the architecture of the software is explained, detailing the four namespaces of the software, how they relate to one another and what components and functionality they contain. It also contains a dependency diagram, showing the dependencies between the different components in the software.

In the last section, a run-down is given of typical execution flow, giving a high-level overview of the functionality of the software.

A slightly redacted version of the document is shown in Appendix A.

3.3 Participant selection

Participants were recruited by inviting all developers in the regional branch of ALTEN to take part in the experiment. Participation was voluntary, with no reward offered. This means that the participants were likely motivated [118, 71], which is a possible (positive) factor in their performance during the experiment [71]. In order to attract as much participants as possible, participation was open to all developers. This also enables us to get a decent sample of the different types of developers within ALTEN.

The only condition on participants was that they had a (self-assessed) minimal understanding of the C# language, as the code base they had to work on in the experiment was written in that language. While the experiment and its goals are set up to be as generic as possible, it is expected that the experiment is too difficult to solve if the language is completely unknown to the participants. Additionally, it is undesirable if participants spend their time on understanding basic syntax and language features, instead of the actual code. The actual proficiency of the participants is not explicitly validated before the experiment.

Due to limited availability of the participants, the experiment was held in the evening. This means that the participants were likely tired due to already having worked the entire day. This will decrease the performance, but is expected to decrease performance of all participants equally.

The participants that took part in the experiment are discussed more in detail in Section 5.1. The possible results of inexperience are discussed in Section 6.2.3.

3.4 Participant distribution

The participants are distributed in four different groups (a *between subjects design* [119]), in order to control for both the independent variables. The groups are as follows (see also Table 3.1):

Variable	No project knowledge	Project knowledge
Original code	Group 1	Group 3
Refactored code	Group 2	Group 4

Table 3.1: Group distribution for the experiment

Group 1 This group serves as the control group: in the experiment they receive the original code with no additional knowledge about the project.

Group 2 This group serves to assess the impact of refactoring alone: in the experiment they receive the refactored code, without additional knowledge about the project. In particular, its aim is to answer **RQ1**.

Group 3 This group serves to assess the impact of introducing knowledge about the project, without refactorings. In particular, its aim is to answer **RQ2**.

Group 4 This group serves to assess the combined impact of refactored code with a project the subjects have (some) knowledge about. In particular, its aim is to answer **RQ3**.

Participant performance during the experiment does not only depend on the independent variables, but also on several confounding variables, not all of which may be known. For example, it is likely that programming skill is a large influence on performance during the experiment. To reduce the impact of the confounding variables, the groups need to be balanced.

Because groups 3 and 4 need to be informed about the project beforehand, all participants need to be selected well before the start of the experiment, in order to give them time to read the document before the experiment. If only a part of the participants were known at the time of distribution, participants joining after this moment could not be included in the groups receiving the document, which might introduce bias in terms of difference in motivation (participants signing up earlier are likely to be more motivated).

With a small number of participants (the experiment had 16 participants), simply assigning the participants to the four groups at random is likely to produce imbalanced groups. The groups were thus balanced using a rudimentary expertise metric, making the experiment a quasi-randomized experiment. The expertise metric was made by scaling, combining and averaging several answers in the pre-experiment questionnaire (see also Section 3.5.1), such as years of professional experience and experience in the C# language. The participants were then divided such that the difference in average expertise between the different groups was (approximately) as small as possible.

3.5 Resulting data

In the process of the experiment, a lot of data will be collected in order to make quantitative and qualitative analysis possible. The data can be split up into three categories:

- Data collected before the experiment, used mostly to collect data on the background of the participants and to balance the groups.
- Data collected during the experiment, used mostly to directly answer the research questions.
- Data collected after the experiment, used mostly to supplement the data collected during the experiment and check for errors or gaps in the experiment.

3.5.1 Before the experiment

After the participants are selected, data should be collected about them. This data should measure background factors that could influence their performance in the experiment, such as programming experience. Intuitively, participants with more programming knowledge are expected to score better in the experiment. As such, the data is relevant and should be collected.

ALTEN organizes monthly 'Uncle Bob' sessions, where a varying group of about 15 consultants watch one of Robert C. Martin's videos[1] and discuss its contents afterwards. The content of these videos is very close to that of the Clean Code book [82]. As such, consultants who regularly attend these sessions are potentially more used to the coding style of Clean Code and are more likely to apply them in practice themselves. This could improve their performance.

This data was mostly collected by performing a questionnaire beforehand. The digital questionnaire was sent to participants a week before the experiment. It contained questions about:

- Programming experience, both in general and specific for C#.
- Clean Code, asking if they (partially) read the book on Clean Code, or watched any of Martin's videos.
- Knowledge of basic object-oriented design principles, in the form of a small test.
- Habits in method and class length
- Habits in unit testing

The questions about object-oriented knowledge and habits are adapted from similar research done by Ammerlaan [6], in order to enable direct comparison.

Other sources for data about the participants exist within ALTEN. Every consultant is expected to track their skill set, along with a level of expertise, in an internal system. This data is freely available within ALTEN and was used to supplement the data from the questionnaire. In particular, it was used to categorize the type of work the participant generally performs. A caveat is that this data is sometimes incomplete or outdated. In order

[1] `https://cleancoders.com/`

to catch errors and fill in gaps, the data retrieved from this source was verified by a manager familiar with the consultants and their work.

The data collected before the experiment aims to provide insight into the skill and thus expected relative performance of the participants during the experiment. Because the differences in performance may be significant, it is important to balance the groups to have a roughly equal distribution of estimated skill level. This removes the influence of skill level when comparing group performances, strengthening the design that the change in dependent variable (time to completion) is due to the change in dependent variables (refactoring, project knowledge). Thus, the data should steer and verify group assignment.

Additionally, the data might prove useful when analyzing the results, in order to possibly explain some unexpected observations. Furthermore, some conclusion can be made about whether the participants are a representative sample for consultants within ALTEN.

3.5.2 During the experiment

The most important data is collected during the experiment, as this will be the main basis for answering the research questions. This data should measure the extent to which the code is understandable. This will be measured mainly by the performance of the participants in the experiment. In order to make solid conclusions, both quantitative and qualitative data should be collected.

In the experiment, the start and end time for every task (as described in Section 3.6) for every participant is recorded. This is done by having the participants write down the time they finished each task[2]. This timing data provides the core insight in the performance of the participants. In addition, the code changes the participants submit serve as valuable data: from this we can assess the correctness of the solution, but also where and how they made the changes are interesting observations.

Apart from this direct data, it is useful to collect supplementary data during the experiment, which could provide insight in the thought and solving process used by the participants. An example of this is found in a research by Vokáč et al., where intermediate solutions and compilation information was collected [133]. However, tool support for such collection is very limited for Visual Studio, the IDE participants are required to use during the experiment due to the code base. The only adequate tool found was FeedBaG++ [5], but its dependency on the ReSharper plugin made it unfitting for this experiment.

An alternative is to have the participants record their screens during the experiment. In this way, the desired data is still available and even enriched with actions made outside the IDE, for example search queries on the web. A large downside is that the data is unstructured and analysis is manual and time consuming. Because the data gathered by this method is seen as valuable, it was implemented during the experiment. Participants were given the option to abstain from having their screen recorded, for example due to privacy concerns when using their own computer.

[2]Because the participants record the time by looking at the clock, the granularity of the timekeeping is in minutes. In the case when a participant finishes the task in less than a minute, the time difference between the start and the end could be 0 minutes. To correct this, a time difference of 0 minutes is considered to be 0.5 minutes

Additionally, after every set of tasks the participants are asked to answer some questions about the tasks they just completed. In these, the participant is asked how confident they are they answered correctly, and to give their opinion on the quality of the code and the difficulty of the task. The participants who had received a document beforehand were also asked whether they thought the document has helped them in solving the tasks.

The data collected during the experiment primarily aims to provide the appropriate data to answer the research questions. Additionally, the recordings and the questions after the tasks aim to give more insight and strengthen the conclusions drawn from the primary data.

3.5.3 After the experiment

At the end of the experiment, participants are again asked several questions about the code in general and the experiment. The questions can be divided into four groups, all with their own purpose.

Three questions ask the participants about their opinion on the code, and what aspects of the code bothered or delighted them. These questions aims to see what the participants thought about the code in general and allows for possible conclusions to be made about the difference in opinion between the groups.

Two questions ask the participants whether the thought any extra documentation or other prior knowledge would have helped them solve the tasks in the experiment. The aim is to compare the difference between the participants who had received a document beforehand, and those who did not.

Three questions are only given to the participants who had received the document. In these questions, the participant was asked whether they actually read the document prior to the experiment, how much they thought it helped them during the experiment, and whether they used the document during the experiment. The aim of the questions is to provide insight on the effect of the document and how participants used it.

Finally, three questions ask the participant about the experiment itself and their experience participating in the experiment. These questions mostly aim to catch possible errors and other factors that might impact the results significantly.

After the participant is finished, they hand in the papers containing their answers and comments. The code solutions and recordings are collected digitally. After this moment, participants are free to leave, but are invited to have a short discussion with the author of the experiment or other participants about their thoughts and experiences. These unstructured discussions are used to see if there were any other matters that were relevant to the experiment and its results which were not otherwise captured during the experiment.

3.6 Tasks

The experiment consists of multiple exercises. Each exercise is designed to focus around a single piece of functionality that was refactored, usually a class. Every exercise consists of two parts: a small number of assignments (the tasks) and a couple of questions about the tasks the participants just completed, as mentioned in Section 3.5.2. Two types of tasks were drafted: code assignments and multiple choice questions.

3.6.1 Code assignments

In the code assignments, participants are asked to add or change functionality to the code. The tasks were designed such that the actual changes required to solve the task are minimal, usually only a single line of code. This means that the time participants spend in solving the tasks only contains a minimal amount of coding, so that the majority of the time spent is in locating and understanding the code. The participants provide their solutions digitally and these are checked for correctness after the experiment. The participants had rely on their own expertise to judge whether the assignment was finished correctly. While the code did contain an extensive unit test suite, there were no tests testing specifically for the requested functionality (although the participants were free to write their own).

As the provided answer for a code assignment is written code, it is not straightforward to assess its correctness. While there may be a model solution for the correct answer, there are many possible other solutions that have the same functionality, possibly differing in small details such as naming or formatting to large differences such as location in code and approach. Besides functionally equivalent solutions, there is also a myriad of other possible solutions that may perform very similar but slightly different, or may be plain incorrect. In addition, there are several other factors that can be taken into account, such as bugs, code quality and conformity to coding standards.

In this experiment, the purpose of assessing correctness is to assess whether the participant had understood the code well enough to implement the required changes. As a result, only *functional* correctness is considered. This means that a solution is considered correct if the code, when executed, performs the required functionality. Some leniency with regards to small bugs is practiced in assessment: if the solution is mostly correct, but contains a small bug that prevents it from doing exactly as required, it will still be considered correct. This again follows from the goal to assess whether the participant has understood the code, instead of assessing the programming expertise of the participant. Similarly, if the solution is correct for a reasonable alternative interpretation of the assignment text, it will be considered correct as well.

In order to maximize realism, participants are expected and asked explicitly, to code as they normally would. However, when performing a change, the work done by a developer is usually not limited to understanding the code and performing the minimal changes needed. Instead, developers may for example also perform a cleanup or refactoring step, and perform some non-code tasks, such as updating external documentation, pushing changes to version control and submitting the code for review. In order to minimize the extra steps taken by the participants, they were explicitly told before the experiment that they can stop with the code assignment as soon as they are ‘reasonably certain‘ that they have implemented the requested functionality. No specific activities (such as writing unit tests or refactoring) were explicitly mentioned, as not to guide the participants in their solving approach.

3.6.2 Multiple choice questions

The multiple choice questions are a more direct approach at measuring understandability. In the question, the participant is asked a question about the functionality of the code. Exam-

ples of such questions are “What does method X do?” or “What does parameter Y contain?”. The questions are designed such that answering the question requires understanding of the code.

The question task can be seen as a more ‘pure’ measure of understandability, because no coding is needed and as such most, if not all, time is spent understanding the code. On the other hand, the understanding needed to answer a multiple choice question is arguably less than that needed to solve a coding task. Often, some choices make less sense than others and can be eliminated quickly, with only limited understanding of the code. A risk is that participants only answer the question correctly not because they understand the code and select the most relevant option, but understand the code just enough to eliminate the other options. Good question design can partially prevent this.

For multiple choice questions correctness can be defined in a straightforward way: there is usually a single correct option and selecting that answer amounts to answering the question correctly. In the cases where participants need to select multiple options, the correctness is defined as the percentage of correct options selected (e.g., selecting one correct option when three correct options should be selected results in answering the question 33% correctly). For simplicity, no penalty is given to selecting incorrect options in these cases. One could argue that participants would simply select all options resulting in 100% correctness, but in the context of this experiment (where participants are not judged on their performance) this is considered unlikely.

A mix of both code assignments and multiple choice questions was chosen because both methods are seen as valid indications of code understandability. In addition, some parts of the functionality are better suited for certain types of task. For example, a task designed to have users dive into a complex algorithm may better suited to be a multiple choice question, as performing a small change (as required by code assignments) in the middle of an algorithm might not be possible, or be too complex for the limited time frame of the experiment.

For both types of tasks, when analyzing the completion times, only those with correct solutions are considered. From incorrect solutions it cannot be concluded that the participant has understood the code, and the time can therefore not be used as an indication of understandability. In the case of multiple choice questions with multiple correct options, partial correctness (i.e., at least one correct option selected) is considered enough for inclusion, as this means the participant had demonstrated at least partial understanding.

3.7 Pilot experiment

The experiment is tested in two pilot experiments. The goal of these pilots is to catch errors in the design of the experiment, testing the data collection and processing, and to verify the tasks, checking if the participants has enough information and the task is not too difficult, so the task can be completed in time. Both pilots are performed with a single participant, to allow for extensive monitoring.

The first pilot experiment is performed around a week before the experiment. The participant has extensive knowledge of C# and is familiar with the experiment and its setup,

but not familiar with the code base. This pilot gives a first impression about the experiment. After the pilot, improvements are made to the wording of the questions and the final selection of tasks was made that is estimated to fit within the planned time frame. Additionally, a small error in the refactorings was found and fixed.

Shortly before the experiment, the second pilot is performed. This participant has little knowledge of C#, and thus the results are used to determine if the tasks are not too difficult or do not take too long to solve for participants less familiar with C#. The difficulty and length of tasks were deemed fine, and no adjustments to the experiment were made. While the experiment performed was the same as the actual experiment, it was decided not too include the results from this pilot in the analysis, because of differences in the execution.

Chapter 4

Refactoring

This chapter describes the project and the refactorings done, which are used in the experiment. First, the project is described. Afterwards, some changes made to the project are discussed along with the motivation to make these changes. The rest of the chapter details the various refactorings done in the project, along with the thought process behind the refactorings.

4.1 Project description

The software under study is a Windows application written in C#, with a GUI built using the Windows Presentation Framework (WPF). In the application, a barn owner can load an XML file containing the floor plan of their barn, along with the transponders and robots. They can then draw routes for these robots, which can again be exported. The resulting XML file can be loaded by the robot, which will then be able to drive the drawn routes autonomously.

The software is currently in active development and has not been released yet. For the purpose of this thesis, a snapshot of the system was taken at the end of a development sprint. The project is being developed according to Scrum methodology, meaning that there is a working product at the end of every sprint. This is confirmed by inspection of the application at the time of the snapshot: the application is working without any major bugs or gaps in functionality[1].

The software is medium sized, containing about 4000 statements of program code. The project is also well tested: about 8000 statements of test code are in the software, making 2107 tests reaching 90% statement coverage. This means the code can not be regarded as legacy, as it is still under active maintenance and tested. The code also does not have any code quality issues that are apparent through the entire system. However, this does not mean there are no refactoring opportunities, as discussed later in this chapter.

[1] Naturally, the final application will have more functionality, but the functionality as described is all present and working.

4.1.1 Code structure

The software consists of four main namespaces: `Application`, `Core`, `Domain` and `Component`.

The `Application` namespace is the entry point of the application, containing the main view and its model. It is responsible for initializing the application.

The `Core` and `Domain` namespace contain all the application business logic. The `Core` namespace contains functionality that is independent of the project domain, while the `Domain` namespace contains the domain specific functionality. The rationale is that the functionality in the `Core` namespace should be easily re-used in other applications. For example, the `Core.Data` namespace contains classes such as `Point2D` and `Line`, while the `Domain.Data` namespace contains classes such as `Barn` and `Transponder`.

The `Component` namespace contains four projects: `Component.Editor`, `Component.Editor.Wpf`, `Component.Ribbon` and `Component.SpatialCalculus`. The `Ribbon` and `SpatialCalculus` projects are small projects: the `Ribbon` project contains a few extensions for the Ribbon library used in the GUI and the `SpatialCalculus` project contains functionality for calculations on 2D lines and vectors, that originated from another project.

The two `Component.Editor` projects deserve special attention. The contents of these project are a copy of the open source graphing library OxyPlot[2], slightly modified to fit the purposes of the project. Copying the source code of another into your own project as opposed to referencing it as a library is an unusual (and mostly discouraged) choice, but further study of this choice (and potentially undoing it by refactoring) is out of the scope of this thesis. As discussed in Section 4.2.1, these projects were removed and references as external libraries for the purpose of the experiment.

While the refactorings in this thesis are based on the Clean Code guidelines as written by Martin [82], one exception is made due to the project guidelines. These dictate that all public methods should have accompanying XML documentation comments, while Martin would argue that these are “clutter”, “not generally useful” and “amount to little more than cruft and distraction” [82]. However, in this case arguing the project guidelines is out of scope, and must be followed. Therefore, in the cases where relevant, documentation comments were added to (new) public methods and classes.

4.2 Non-refactoring changes

Apart from the refactorings done (to be discussed in the next sections), several non-refactoring changes to the code base were made. These changes were done to improve the experiment and are applied to both the original version of the code and the refactored version. These changes can be categorized in three groups: scoping the experiment, improving portability and fixing small bugs.

[2] `http://www.oxyplot.org/`

4.2.1 Experiment scope

Because the inclusion of the two `Component.Editor` projects is an unusual choice, extra thought should be given on how to handle them. If their presence has a large effect on understandability, the possibility to draw general conclusions from the experiment is reduced, as in that case a specific situation would have too big of an impact on the results. Therefore, it is important to estimate the impact of the `Component.Editor` projects and consider possible solutions.

A few factors are identified that could mean understandability is impacted:

- Developers may treat external code differently than program code while developing. For example, during debugging when they encounter a call to external code, they might skip these as it is usually not expected that a bug lies in third-party code. However, with the inclusion of the code inside the project, the developers can not recognize this boundary, and thus may spend more time understanding this code than they would have otherwise.

- External code is possibly written in a different code style and/or using different standards than the program code, possibly making it harder to understand.

- The contents of the `Component.Editor` projects are geared to graphing and drawing, with a large portion also specific to WPF. Most of the functionality is therefore likely to be too specific to expect participants to understand it in a short period of time during the experiment.

In addition, the `Component.Editor` projects are almost four times as large as the rest of the program code (15000 statements versus 4000 statements). Coverage analysis shows that the majority of the code is not executed during typical program usage. Only about 30% of the almost 15000 statements is executed, compared to almost 60% of the 4000 statements in the `Core` and `Domain` namespaces. Analysis of the history of the projects also show that only about 170 statements (less than 2%) have been changed compared to the original version of the OxyPlot library they originated from.

As there is plausible risk that the included projects affect understandability considerably, and the fact that the changes made in the code are minimal, the decision was made to remove the `Component.Editor` projects from the code base and to reference them as compiled libraries. This should improve the generalizability of the experiment.

4.2.2 Portability

At the time of the snapshot, the project was in development by a single developer and work on setting up a build process was in its early stages. This meant that all developing, building and testing was only done on the developer's machine and that portability of the project was not a priority. This poses a challenge for the experiment, as then the code should run with minimal set-up on the machines of the participants.

To help portability, the software (both before and after refactoring) was tested on multiple systems and several changes have been made to the software:

- The project contained a `Application.Version` project, whose only functionality was to provide automatic version numbering and streamlining assembly version numbers across the software. As it posed challenges in portability (e.g., adding extra build dependencies) and did not contain actual functionality, it was removed.

- Several flaky tests were identified, i.e., they would fail on some systems while passing on others. These tests all related to opening the main GUI window and testing some properties (e.g., title, dimensions, etc.) on that window. As fixing these tests proved to be nontrivial and out of scope for this thesis, and because the tests were deemed to be not relevant for the experiment, they were removed.

- Several small improvements were made to ease first-time setup. These consist of: adding an example import file to the root of the project so participants could easily find it, setting the `Application.Main` as the default startup project so participants did not have to manually select it, and explicitly adding the NUnit test adapters as dependency on the project so participants did not have to install any extra extensions for the Visual Studio to run the unit tests.

To allow the participants to test whether their systems could run the software without sending them the code beforehand, a mock project was created with some dependencies that were identified as potentially challenging. As a result, the only issues with participants not being able to immediately run the code during the experiment, occurred with participants that did not run the mock project on their system beforehand.

4.2.3 Fixed bugs

During refactoring several bugs were encountered. In a normal code review setting, these bugs would be fixed along with the refactorings. However, in the context of this thesis, being a research setting, this would introduce several challenges. The biggest of these is the scope of the research. Refactoring has a clear boundary: whenever external functionality is changed, it is not a refactoring anymore. If changes would be made that would change functionality, the data from the experiment can not be used any more to conclude whether the effects are due to refactoring, because the effects may have been caused by the (subtle) change in functionality.

On the other hand, leaving in the bugs is also far from ideal. Most importantly, the bugs were found during the refactoring process and some even surfaced *because of* the refactoring, when splitting up the methods or renaming code units revealed inconsistencies in the logic. Keeping those bugs in, would negatively impact the quality of the refactorings.

As a compromise, the bugs were fixed in both versions of the code, so both in the original and in the refactored version. As a result, the changes between the original and the refactored code are still all refactoring, preserving the research scope. In order to reduce impact of the fixes on the original code, the changes made aimed to be the minimal changes made to sensibly fix the bugs. All fixes were developed in collaboration with and later reviewed by the developer, to make sure the fixes were correct, minimal (i.e., only fixing the bug) and in style with the rest of the code.

The bugs fixed were as follows:

- In the GetAreas method (discussed in Section 4.3.3), several cases of invalid input (such as overlapping walls) were accepted, resulting in sometimes undefined results. The cause was the method incorrectly continuing calculation when presented with invalid input, especially by continuing forming an Area with walls that were already used to form an area. The fix was to halt calculation and throw an exception whenever such a situation occurred. To be precise, two return statements (on lines 87 and 100 in Listing 4.4) and one throw statement (on line 104 in Listing 4.4) were added. Additionally, the calling method in the BarnLayout class was slightly modified to catch the exception, and test cases were added to document the bug and test the behavior.

- In the Contains method of the Area class (discussed in Section 4.4), the functionality behaved incorrectly whenever the Area was extremely large (i.e., spanning to double.MaxValue. The root cause lay in another method (EuclideanDistance), and a condition was present in the code to handle that. However, this extra condition caused other incorrect behavior. The fix was to remove this condition, and to let the behavior of the Contains method be undefined in the cause of impossibly large areas, because it was not expected that such cases would occur in the software[3]. To be precise, line 45 in Listing 4.7 originally read `if (double.IsInfinity(distance) || distance < 1e-5)`. The IsInfinity check was removed and tests relating to behavior with double.MaxValue were removed.

- In the MainViewModel class (discussed in Section 4.5), whenever the import of a new layout failed, the previous layout was removed anyway. This is not something a user would expect and was confirmed to be a bug by the developer. The fix was to move the statement causing the removal to after the check whether the import was successful.

The rest of the chapter will describe the refactorings made and the motives behind them.

4.3 BarnLayoutHelper class

The BarnLayoutHelper class is a class containing various static helper methods for the BarnLayout class. It boasts over 1400 lines in the editor, which is a first indication that the class is not clean. On topic of class size, Martin states that it is "very desirable" for a class to be "typically 200 lines long, with an upper limit of 500." [82]. In addition, the Helper noun in the class name could be considered a *weasel word*, not providing any information in what the class should (and should not) do [82]. Classes that are simply a collection of helper methods are often incohesive and might have multiple reasons to change.

[3] A 'proper' fix would include significant compensatory code to handle cases around the maximum values that can be represented in a double, which was deemed out of scope for both this thesis and the software

Three distinct groups of functionality can be distinguished in the class. Firstly noticeable is a region block[4] of about 1000 lines long, called `Quadrant30 methods`. Region blocks in C# can be considered a code smell[5], indicating that the block has a different responsibility from the rest of the code in the class, violating the Single Responsibility Principle [84]. The contents of the region block confirm this: the methods inside relate to calculations in the unit circle and have no relation with the rest of the class. They should therefore be moved to their own class. Because the methods do not reason on any domain specific objects, the new class can be moved from the `Domain` namespace to the `Core` namespace.

Further inspection reveals that the methods themselves are not clean as well, containing a large amount of duplication that may be refactored away. However, in the context of this thesis and especially with the experiment in mind, the refactorings might not be useful. Considering the short duration of the experiment, the functionality in this region is expected to be too complicated and specific to expect the participants to understand in a fairly limited time window (i.e., 30 minutes). Therefore, the methods were only be extracted to their own class (*Extract Class* refactoring [47]), leaving further refactorings outside scope. The new class is called `QuadrantMath` and is moved to the `Core` namespace.

With the `Quadrant30` methods extracted away, it becomes clear that the class only has two methods that are called from outside the class: `MarkerLocations` and `GetAreas`. These methods call the rest of the methods in the class, but between each other have no common methods (methods they both call) or properties (being static methods). This is illustrated in Listing 4.1: shaded lines correspond to `MarkerLocations` and the methods it calls, the other lines correspond to `GetAreas` and the methods it calls. This full split makes the class lacking in cohesion, a clear violation of the Single Responsibility Principle [84]. Splitting up the class into two more cohesive classes will make the code a lot cleaner [82]. The `MarkerLocations` method and the methods it calls are extracting to a new `MarkerCalculator` class, `GetAreas` and its helper methods are extracted into `AreaCalculator`. Both new classes can be seen as method objects [47]/command objects [49], meaning they only contain a single public method to be called[6].

4.3.1 `MarkerCalculator` class

The `MarkerLocations` method employs a complex algorithm to determine the locations of waypoints in the layout, which the user can use to draw routes. The method itself is very long (around 40 lines of code) and contains multiple comments explaining the steps of the algorithm, giving a clear signal that it should be split up. Additionally, some of the methods it calls suffer from the same issues. However, the algorithm is estimated to be too

[4] `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives/preprocessor-region`

[5] They can be regarded as an instance of 'Position Markers', which Martin regards as clutter. [82]. Additionally, the popular static analysis tool StyleCop considers region blocks to be a violation [127].

[6] The `MarkerLocations` class has multiple public methods, but these are not called from outside the class other than in test code.

```
1  MarkerLocations(this BarnLayout layout, IRobotScraper
   ↪  scraper):IEnumerable<Point2D>
2  CreateMarkerLocationsOverlay(IReadOnlyCollection<Point2D> locations, double
   ↪  offset):IEnumerable<Point2D>
3  GetAreas(IReadOnlyCollection<WallBase> walls):IEnumerable<Area>
4  GetMostRelevantBoundingLocations(Line firstLine, Line
   ↪  secondLine):IEnumerable<BoundingBoxLocation>
5  CreateValuesInBetweenExtremes(Tuple<double,double> extremes, double
   ↪  offset):IEnumerable<double>
6  IsValidLocation(BarnLayout layout, double offset, Point2D
   ↪  proposedlocation):bool
7  LayoutAreaLinePairs(BarnLayout layout):IEnumerable<Tuple<Line,Line>>
8  WallLocationPairs(Area area):IEnumerable<Tuple<Line,Line>>
9  FlipBoundingBoxLocation(BoundingBoxLocation
   ↪  boundingBoxLocation):BoundingBoxLocation
10 GetArea(IReadOnlyCollection<WallBase> allWalls, IReadOnlyList<Point2D>
   ↪  areaPoints):Area
11 GetConnectedWalls(this IEnumerable<WallBase> walls, Point2D point, WallBase
   ↪  filter):IEnumerable<WallBase>
12 UpdateUnreachableHelper(Point2D currentPosition, HashSet<Point2D> points,
   ↪  List<WallBase> usedWalls, List<WallBase> handledWalls,
   ↪  List<List<Point2D>> areasPoints, IReadOnlyCollection<WallBase>
   ↪  allWalls):void
13 Angle(WallBase a, WallBase b):double
14 Angle(Line a, Line b):double
```

Listing 4.1: `BarnLayoutHelper` methods. Shaded methods are `MarkerLocations` and the methods it calls, the other methods are `GetAreas` and the methods it calls.

complex to fit into an assignment during the experiment. Therefore, further refactoring the `MarkerLocations` class is kept out of scope and mostly kept as-is.

There are two exceptions. Firstly, the `FlipBoundingBoxLocation` method took a `BoundingBoxLocation` as input argument, only reasoned on this input argument and returned another `BoundingBoxLocation`, smelling of feature envy [47]. The method is moved to the `BoundingBoxLocation` enum and renamed to simply `Flip`[7].

The other exception is the `IsValidLocation` method. During refactoring, this method was extracted into its own class.

4.3.2 `LocationValidator` class

The `IsValidLocation` method checks four different conditions in which a location for a marker would not be considered valid, each preceded by a comment. The method is shown in Listing 4.2. These comments are a clear signal that the conditions should actually be extracted into their own methods [82].

[7] As enumerations in C# cannot have methods, the method is defined as an extension method.

```
1  public static bool IsValidLocation([NotNull] BarnLayout layout,
2                                     double offset,
3                                     [NotNull] Point2D proposedlocation)
4  {
5      // Cannot be NaN or infinity
6      if (double.IsNaN(proposedlocation.X)
7          || double.IsInfinity(proposedlocation.X)
8          || double.IsNaN(proposedlocation.Y)
9          || double.IsInfinity(proposedlocation.Y))
10     {
11         return false;
12     }
13
14     // The proposed location should lie inside the outer area.
15     if (!layout.OuterArea?.LiesInsideArea(proposedlocation) ?? true)
16     {
17         return false;
18     }
19
20     // The proposed location may NOT lie inside an inner area.
21     if (layout.InnerAreas.Any(a => a.Contains(proposedlocation)))
22     {
23         return false;
24     }
25
26     // The proposed location must be at least offset away from any line.
27     return !layout.Walls.SelectLocations().Any(wall =>
       ↪  wall.GetEuclideanDistanceToPoint(proposedlocation) < offset - 0.5);
28 }
```

Listing 4.2: `IsValidLocation` method before refactoring. Documentation comments are omitted for brevity.

Doing so, four new methods are created, which are only called from the `IsValidLocation` method. This decreases the cohesion of the `MarkerLocationsCalculator` class. Therefore, they are moved into their own class, called `LocationValidator`, increasing the cohesion of both classes.

This has some extra benefits. Firstly, `layout` and `offset` parameters do not change in between calls to the `IsValidLocation` method, and refactoring these to be fields of the `LocationValidator` class makes sure these only have to be passed once, reducing the amount of parameters and with it, the conceptual power needed to understand the method [82]. Secondly, the new class can now be more easily refactored when the requirements change. For example, if it is expected that more conditions will be added in the future, the functionality could be refactored to use the Strategy pattern [51].

The resulting `IsValidLocation` method is shown in Listing 4.3. The resulting `LocationValidator` class can also be seen as a command object.

```
public bool IsValidLocation([NotNull] Point2D proposedLocation)
{
    return !IsNaNOrInfinity(proposedLocation)
       && LiesInsideOuterArea(proposedLocation)
       && !LiesInsideInnerAreas(proposedLocation)
       && IsAtLeastOffsetAwayFromAllLines(proposedLocation);
}

private bool IsNaNOrInfinity(Point2D proposedLocation)
{
    return double.IsNaN(proposedLocation.X) ||
    ↪  double.IsInfinity(proposedLocation.X)
    || double.IsNaN(proposedLocation.Y) ||
    ↪  double.IsInfinity(proposedLocation.Y);
}

private bool LiesInsideOuterArea(Point2D proposedLocation)
{
    return _barnLayout.OuterArea?.ContainsInside(proposedLocation) ?? false;
}

private bool LiesInsideInnerAreas(Point2D proposedLocation)
{
    return _barnLayout.InnerAreas.Any(a => a.Contains(proposedLocation));
}

private bool IsAtLeastOffsetAwayFromAllLines(Point2D proposedLocation)
{
    return _barnLayout.Walls.SelectLocations().All(wall =>
    ↪  wall.GetEuclideanDistanceToPoint(proposedLocation) > _offset - 0.5);
}
```

Listing 4.3: `IsValidLocation` method after refactoring. Documentation comments are omitted for brevity.

4.3.3 `AreaCalculator` class

The `GetAreas` method and its helper methods are shown in Listing 4.4. The method serves to construct `Area` object from its input. For this, it uses an algorithm mostly implemented in the `UpdateUnreachableHelper` method. This is a recursive private method that violates multiple Clean Code principles.

```
public static IEnumerable<Area> GetAreas(IReadOnlyCollection<WallBase> walls)
{
    var areasPoints = new List<List<Point2D>>();
    var handled = new List<WallBase>();
```

```
5       foreach (WallBase wall in walls)
6       {
7           if (handled.Contains(wall))
8           {
9               continue;
10          }
11
12          handled.Add(wall);
13
14          UpdateUnreachableHelper(wall.Location.Start, new
            ↪  HashSet<Point2D>(new[]
15          {
16              wall.Location.End,
17              wall.Location.Start
18          }), new List<WallBase>
19          {
20              wall
21          }, handled, areasPoints, walls);
22      }
23
24      foreach (List<Point2D> areaPoints in areasPoints)
25      {
26          yield return GetArea(walls, areaPoints);
27      }
28  }
29
30  private static IEnumerable<WallBase> GetConnectedWalls(this
    ↪  IEnumerable<WallBase> walls, Point2D point, WallBase filter)
31  {
32      return walls.Where(w => !Equals(filter, w) &&
        ↪  (point2DEqualityComparer.Equals(w.Location.Start, point) ||
        ↪  point2DEqualityComparer.Equals(w.Location.End, point)));
33  }
34
35  private static Area GetArea(IReadOnlyCollection<WallBase> allWalls,
    ↪  IReadOnlyList<Point2D> areaPoints)
36  {
37      var areaWalls = new List<WallBase>();
38      for (var i = 1; i < areaPoints.Count; i++)
39      {
40          Point2D previousPoint = areaPoints[i - 1];
41          Point2D currentPoint = areaPoints[i];
42
43          areaWalls.Add(allWalls.First(previousPoint, currentPoint));
44      }
45
46      areaWalls.Add(allWalls.First(areaPoints.First(), areaPoints.Last()));
47
48      return new Area(areaWalls);
```

```
49  }
50
51  private static void UpdateUnreachableHelper(Point2D currentPosition,
52                                              HashSet<Point2D> points,
53                                              List<WallBase> usedWalls,
54                                              List<WallBase> handledWalls,
55                                              List<List<Point2D>> areasPoints,
56                                              IReadOnlyCollection<WallBase>
                                                ↪ allWalls)
57  {
58      // only filter the most recent addition so we don't immediately loop back
59      WallBase previousWall = usedWalls.Last();
60      IEnumerable<WallBase> connectedWalls =
        ↪ allWalls.GetConnectedWalls(currentPosition, previousWall);
61
62      // order walls by the smallest increment in angle (looking at it from a
        ↪ CCW perspective)
63      IOrderedEnumerable<WallBase> orderedWalls = connectedWalls.OrderBy(wall =>
        ↪ Angle(previousWall, wall));
64      foreach (WallBase wall in orderedWalls)
65      {
66          handledWalls.Add(wall);
67
68          Point2D next = point2DEqualityComparer.Equals(wall.Location.Start,
            ↪ currentPosition)
69                             ? wall.Location.End
70                             : wall.Location.Start;
71          // found a loop
72          if (points.Contains(next))
73          {
74              // find the intersection in the stack
75              IEnumerable<Point2D> it = points.Reverse().TakeWhile(a =>
                ↪ !Equals(a, next));
76              var area = new List<Point2D>
77              {
78                  next
79              };
80              area.AddRange(it);
81
82              // check to see if this is just a subset of a larger area
83              bool subset = areasPoints.Any(a => area.Intersect(a).Count() ==
                ↪ area.Count);
84              if (!subset)
85              {
86                  areasPoints.Add(area);
87                  return;
88              }
89          }
90          else
```

```
91          {
92              var np = new HashSet<Point2D>(points)
93              {
94                  next
95              };
96              UpdateUnreachableHelper(next, np, new List<WallBase>(usedWalls)
97              {
98                  wall
99              }, handledWalls, areasPoints, allWalls);
100             return;
101         }
102     }
103
104     throw new InvalidOperationException($"The wall at location
        ↪  {usedWalls.Last().Location} is not part of any contour!");
105 }
```

Listing 4.4: `GetAreas` and helper methods before refactoring. Documentation comments are omitted for brevity.

Firstly, the method name does not clearly convey its intent: the word `Update` in the name and the `void` return type imply a change of state. However, the class state cannot be changed, because the method is static. This means that one of the parameters must change[8], but it is not clear which one is changing: none seem to fit the hint `Unreachable`. Closer inspection of the calling `GetAreas` method is needed to discover that the `areasPoints` parameter is used as a result.

Additionally, six parameters are seen in the method signature. This is clearly more than recommended: Martin states that "More than three [parameters] requires very special justification - and then shouldnt be used anyway." [82] and Visser et al. state "Limit the number of parameters per unit to at most 4." [132].

The method also spans 50 lines in the editor and contains comments explaining the flow of the method, suggesting that it is more understandable to split the method up into multiple methods [82]. This is also backed up by Visser et al., who state "Limit the length of code units to 15 lines of code." [132].

For better understanding of the refactorings, some extra context and understanding of the `GetAreas` functionality is useful. In the application, an `Area` is defined by its contour: a collection of `Walls` that form a loop. In Figure 4.1 an example is shown with four areas: the black outer area with four walls, two blue inner areas with four walls each and the yellow inner area with six walls. Note that the two blue areas touch, but are considered two separate areas.

The application receives its input in the form of a list of lines (in any order) that

[8] This is possible because objects are passed by reference in C#, so the caller method can make use of the changed object.

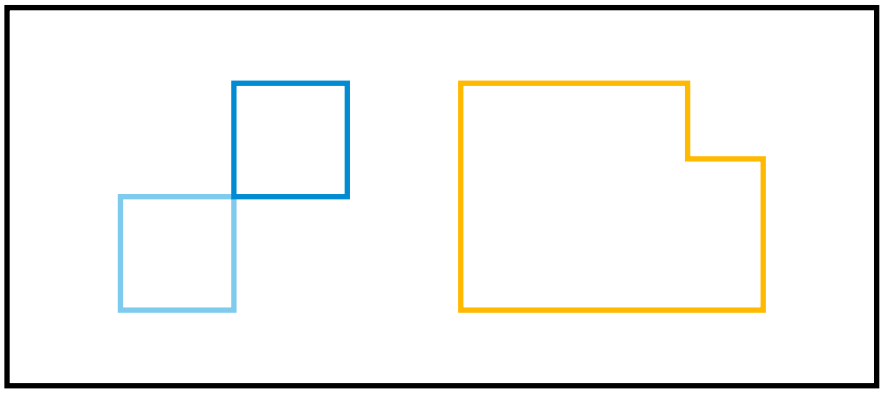


Figure 4.1: Example Barn with four Areas

make up the walls. The GetAreas method serves to find the loops in these walls, grouping and returning them. It does this by picking a point belonging to a Wall (a Wall is defined as a line of two points), and following connected walls until a loop is found in the visited points. This loop might occur at another point than the starting point, so whenever the loop is found the algorithm needs to trace back the found points until the loop point.

A few conditions on the input are noteworthy:

- Each Wall is part of an Area.
- Each Wall can only be part of a single Area.
- An Area is always a loop (and not a line). Therefore, an Area consists of at least three Walls.

Input not adhering to the conditions is considered invalid input.

In the first step of the refactoring, the observation is made that almost all parameters of the UpdateUnreachableHelper are simply passing state in one way or another. Refactoring the method to be non-static allows to change these to class fields, more clearly expressing state and making the code cleaner by reducing the amount of parameters passed around.

- currentPosition holds the location the algorithm currently is at. This is the single parameter the calculations inside the method are based on, and is thus kept. However, instead of accepting a Point2D, it now takes a WallBase. This eliminates the requirement that the calling method should extract the relevant point to start at, and allows the algorithm to keep track of walls, instead of points.
- points holds the found points for the current area being constructed. The same data (in a slightly different form) is available as usedWalls, so the parameter can be removed.

- `usedWalls` holds the found walls for the current area being constructed. When splitting the method up into multiple methods, this parameter is needed often. It is thus extracted to a field, clearing it when appropriate (i.e., when an area has been fully constructed), decreasing the amount of parameters that need to be passed around. The field is renamed to `VisitedWalls`, to make its intent clearer.

- `handledWalls` contains the walls that have been used for any area being constructed, in order to know not to start following them again. This represents state across the entire duration of `GetAreas`, and is thus moved up to a field, so that it does not need to be passed anymore as a parameter. It is renamed to `WallsPartOfAnArea` to better convey its intent. The original implementation is actually slightly different in this regard, this will be discussed further below.

- `areasPoints` can be considered as the 'return value' of this method, seeing as this list is eventually used to construct the `Area` objects in the end. It is a list, where every element is a list containing the points that make up an area. Instead of storing the resulting lists of points in a list and then iterating over the elements again in order to construct the `Area` objects, it is cleaner to simply return the list and have the caller deal with it directly. Additionally, instead of storing the points and having to retrieve the original walls later, it is simpler to store the walls, allowing more direct construction of the final `Area` object. Thus, this parameter is removed and replaced with a return value of `IList<WallBase>`. This also enables the removal of the second `foreach` loop in the `GetAreas` method along with the `GetArea` method, as the `Area` objects can now be constructed directly by calling the constructor with the returned list.

- `allWalls` simply contains all the walls passed to the calculator and is never changed. This is refactored to a field and set once in the constructor of the `AreaCalculator` class.

The fact that a wall is only allowed to be part of a single area, means that walls that are already part of a found area do not need to be considered again. The `handledWalls` list serves this function. However, walls are added to the list inside `UpdateUnreachableHelper`, regardless of whether they are actually used for the construction of an area[9]. This causes walls to potentially be skipped in the main loop in `GetAreas`, while they still need to be included as part of an area. Luckily, this bug does not affect the external behavior of the method, because there will always be a wall that has not been handled, so the missed area can always form[10]. However, it is still incorrect behavior and should be fixed. The addition of the walls to this list is thus moved to right before the found area is 'returned' (line 86 in Listing 4.4), so these walls are sure to belong to an area. As mentioned before, it is renamed to `WallsPartOfAnArea` for clarity.

[9] Passing a `Wall` in the algorithm does not guarantee it being included in an `Area` immediately. Consider starting at the bottom left point of the light blue area in Figure 4.1. Following the walls in the order 'right, up, right, up, left, down' finds a loop the bottom area, which is then constructed. The first two walls visited are therefore visited, but not (yet) part of a formed area.

[10] Formal proof is outside of the scope of this thesis.

On line 64 in Listing 4.4, it can be seen that the UpdateUnreachableHelper method loops over all walls in the orderedWalls variable, trying to find a loop for each one. However, in almost all cases there will be only one Wall connected to the current Wall, and thus the orderedWalls collection will contain only one element. In those cases, the foreach statement is superfluous.

Closely examining the functionality within the loop, it follows that the only situations in which the next iteration of the loop is entered, is when the subset check on line 84 fails. This check fails whenever the found loop was already (part of) an earlier found Area. The bug fix mentioned in Section 4.2.3 already took care in eliminating most of these situations. The only case in which this check could fail is whenever the connectedWalls variable (defined on line 60) contains WallBase objects that were already used to form an Area. As discussed above, a wall can only be part of one area, meaning that already used walls should not be considered when trying to form new areas. Therefore, the initialization of connectedWalls is modified to exclude WallsPartOfAnArea. After this, the entire subset check can be removed. Now the foreach always returns in the first iteration, and thus the foreach (WallBase wall in orderedWalls) line can be replaced with WallBase wall = orderedWalls.FirstOrDefault(). The wall variable should then be explicitly checked for null, throwing the InvalidOperationException if the check fails.

With the previous changes, the recursion of the UpdateUnreachableHelper has become the more specific form of tail recursion. The recursive call can now be refactored away into an equivalent loop using the *Replace Recursion with Iteration* refactoring [47].

With the functionality cleaned up, the UpdateUnreachableHelper is now more easily split up into shorter methods. Firstly, the functionality that is on lines 58 to 63 in Listing 4.4, is recognized as finding the next Wall to follow, and is extracted into a FindNextWall method. The functionality on lines 68 to 70, while short, can be considered difficult to understand. It is tempting to use a comment to explain the functionality, but this information is better kept in a function name [82]. Therefore, it is also extracted into a method, called FindNextPoint. Line 72 already has a comment above it that could also be a method name, therefore it is also extracted to the VisitedTwice method, which also is clearer about how the 'loop' is detected. Finally, the functionality on lines 75-80 concern with finding the points that form the contour loop. This is extracted into the FindContourAtLoopPoint method.

Finally, the names of the two main methods are examined. The unclear UpdateUnreachableHelper is renamed with FindContourStartingAt, better reflecting the functionality of the method. The main method, GetAreas, is renamed to CreateAreas to signify that the method is performing relatively expensive calculations, as opposed to simply retrieving data, which is commonly associated with methods starting with Get. The final, refactored code is found in Listing 4.5.

```
1 public AreaCalculator(IReadOnlyCollection<WallBase> walls)
2 {
3     AllWalls = walls ?? new List<WallBase>();
```

```
4       WallsPartOfAnArea = new List<WallBase>();
5   }
6
7   public IEnumerable<Area> CreateAreas()
8   {
9       foreach (WallBase wall in AllWalls)
10      {
11          if (WallsPartOfAnArea.Contains(wall))
12          {
13              continue;
14          }
15
16          var contour = FindContourStartingAt(wall);
17          yield return new Area(contour);
18      }
19  }
20
21  private IList<WallBase> FindContourStartingAt(WallBase firstWall)
22  {
23      VisitedWalls = new List<WallBase>();
24      var currentPoint = firstWall.Location.Start;
25      var currentWall = firstWall;
26
27      while (currentWall != null)
28      {
29          VisitedWalls.Add(currentWall);
30
31          var nextPoint = FindNextPoint(currentWall, currentPoint);
32
33          // found a loop
34          if (VisitedTwice(nextPoint))
35          {
36              List<WallBase> areaWalls = FindContourAtLoopPoint(nextPoint);
37              WallsPartOfAnArea.AddRange(areaWalls);
38              return areaWalls;
39          }
40
41          currentPoint = nextPoint;
42          currentWall = FindNextWall(currentPoint);
43      }
44
45      throw new InvalidOperationException($"The wall at location
        ↪  {VisitedWalls.Last().Location} is not part of any contour!");
46  }
47
48  private Point2D FindNextPoint(WallBase currentWall, Point2D currentPoint)
49  {
50      return pointComparer.Equals(currentWall.Location.Start, currentPoint)
51          ? currentWall.Location.End
```

```
52          : currentWall.Location.Start;
53  }
54
55  private WallBase FindNextWall(Point2D currentPoint)
56  {
57      // only filter the most recent addition so we don't immediately loop back
58      WallBase previousWall = VisitedWalls.Last();
59      IEnumerable<WallBase> connectedWalls = AllWalls.Except(WallsPartOfAnArea)
60                                              .GetWallsConnectedTo(currentPoint)
61                                              .Except(previousWall)
62                                              .ToList();
63
64      // order walls by the smallest increment in angle (looking at it from a
        ↪  CCW perspective)
65      IOrderedEnumerable<WallBase> orderedWalls = connectedWalls.OrderBy(wall =>
        ↪  Angle(previousWall, wall));
66
67      return orderedWalls.FirstOrDefault();
68  }
69
70  private bool VisitedTwice(Point2D point)
71  {
72      var visitedPoints = VisitedWalls.SelectLocations().SelectPoints();
73      return visitedPoints.Count(visitedPoint => Equals(point, visitedPoint)) >
        ↪  1;
74  }
75
76  private List<WallBase> FindContourAtLoopPoint(Point2D loopPoint)
77  {
78      return VisitedWalls.AsEnumerable()
79                         .Reverse()
80                         .TakeUntilInclusive(wall => Equals(wall.Location.Start,
                           ↪  loopPoint) ||
81                                                     Equals(wall.Location.End,
                                                       ↪  loopPoint),
82                                                 2)
83                         .ToList();
84  }
```

Listing 4.5: `GetAreas` (now named `CreateAreas`) and helper methods after refactoring. Documentation comments are omitted for brevity.

To recap, the `BarnLayoutHelper` was split into four different classes:

- `QuadrantMath`, containing the 'Quadrant30 methods'
- `MarkerLocationCalculator`, containing the `MarkerLocations` method and relevant (private) methods.

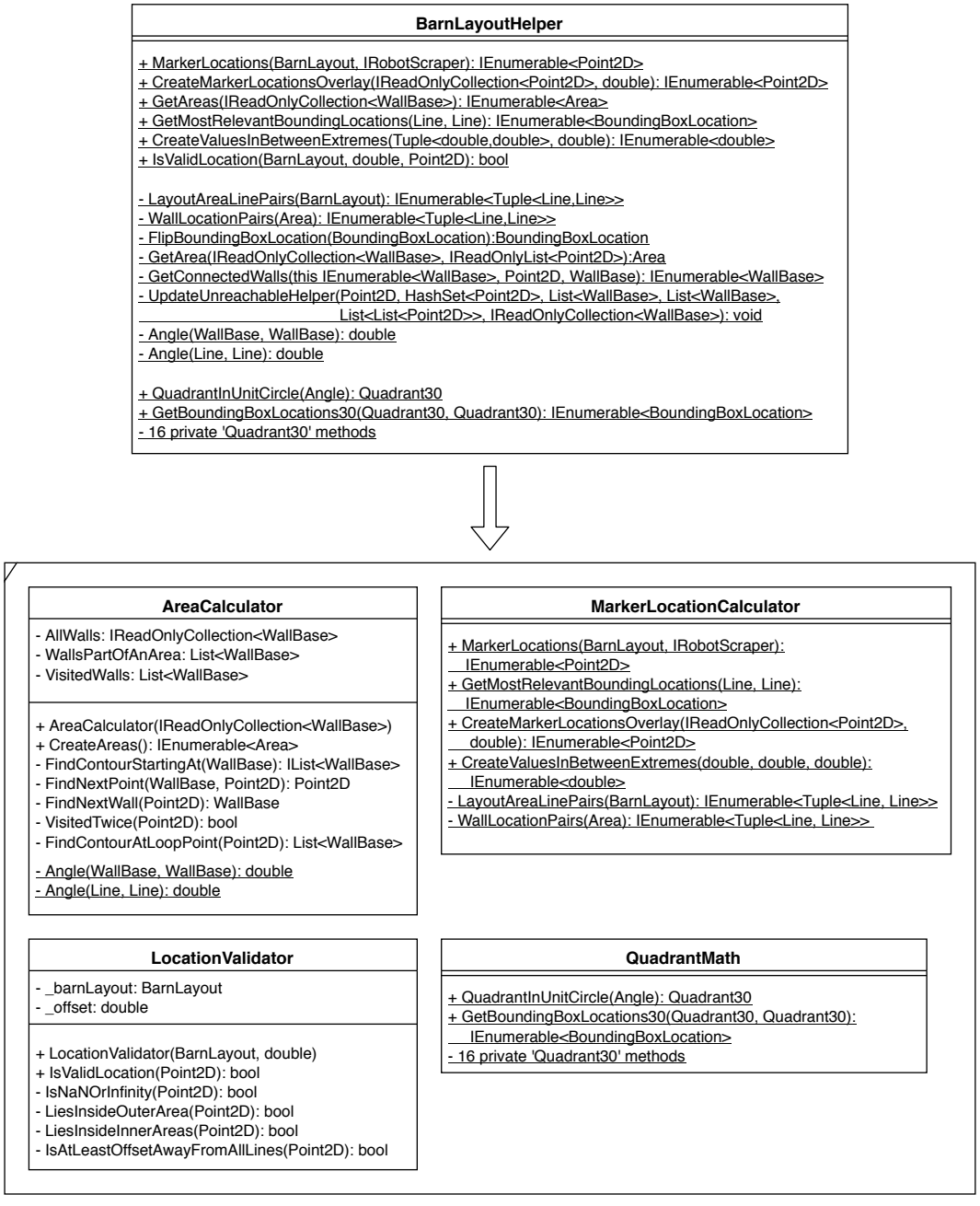


Figure 4.2: `BarnLayoutHelper` class before and the resulting classes after refactoring

- LocationValidator, containing the IsValidLocation method.
- AreaCalculator, containing the CreateAreas method and relevant private methods.

The final result can be seen in Figure 4.2.

4.4 Area class

The Area class represents an area in 2D, initialized with the walls that surround the area. At over 300 lines in the editor, it is relatively long. However, inspecting the class outline in Listing 4.6, we see that most methods and properties are about what we expect from an Area class.

```
1  Area(ICollection<WallBase> contour)
2
3  Walls:ReadOnlyCollection<WallBase>
4  Points:ReadOnlyCollection<Point2D>
5  AbsoluteTop:double
6  AbsoluteBottom:double
7  AbsoluteLeft:double
8  AbsoluteRight:double
9  AbsoluteWidth:double
10 AbsoluteHeight:double
11 Contains(Point2D point):bool
12 LiesInsideArea(Point2D point):bool
13 IntersectsWith(Area otherArea):bool
14 Size():double
15 ==(Area left, Area right):bool
16 !=(Area left, Area right):bool
17 ToString():string
18 Equals(object obj):bool
19 GetHashCode():int
20 Equals(Area other):bool
21 AbsoluteMiddle(Area area):Point2D
22 HasLineInsideArea(Area area, IEnumerable<Line> lines):bool
23 UpdateMinMax():void
24 ContainsWithExtreme(Point2D point, Point2D extremePoint):bool
```

Listing 4.6: Outline of the Area class

4.4.1 InstersectsWith method

One exception is the InstersectsWith(Area otherArea) method. This method tests weather two areas intersect. In the system another class exists, AreaPolygonHelper, containing a single public method: Union(Area area, Area otherArea). As both these

methods are operations on mathematical sets, one would expect both to be in the same place (i.e., class), so this method is a candidate to be moved. However, it turns out that both methods are only called by test code, are therefore dead code, and should be removed [82]. With the removal of `IntersectsWith`, the private `AbsoluteMiddle` and `HasLineInsideArea` methods are also not called by any other method, and are likewise removed.

4.4.2 `LiesInsideArea` method

Some individual methods in the `Area` class can be made cleaner. The method `LiesInsideArea(Point2D point)` is not well-named. This method checks if a certain point lies inside this area. Typical usage will be in the form of `area.LiesInsideArea(point)`. Reading it as a sentence, it is slightly confusing: it reads as if we are checking if the area lies inside some area, but we are giving it a point as a parameter. This is unclean, as "clean code should read like well-written prose." [82].

The method is better named `Contains`, which will make typical usage read a lot cleaner: `area.Contains(point)`[11]. However, the class already contains a `Contains` method. Both methods actually calculate the same, with the only difference between the two being that `LiesInsideArea` considers points that are exactly on the edge of the area to not be 'inside'. This means the methods are actually very much related, which was not obvious from their names. The name `ContainsInside` then makes more sense and the method is thus renamed as such.

The method body is very simple: `!Walls.Select(w => w.Location).Any(line => line.IsOnLine(point)) && Contains(point)`[12]. However, it not clear on first inspection what the first part of the conditional does. Therefore, this is extracted to its own method `PointIsOnEdge`, so the method name serves as an explanation for the purpose of the complex statement. Additionally, the call `line.IsOnLine(point)` is a bit awkward to read, so the `IsOnLine` method in the `Line` class is renamed to `Includes`. The call is now clearer: `line.Includes(point)`.

4.4.3 `Contains` method

The `Contains` method calls the private method `ContainsWithExtreme` four times. This method is very long (50 lines), has an unclear name (what does `WithExtreme` mean?) and an unclear purpose (why is it called four times?). It implements an algorithm where it draws a straight line from a point to the edge of the area (this is the 'extreme'), and counts how often the line crosses an area wall. If it crosses an uneven amount of times, then the point is considered to be inside the area. This is also called the Ray Casting algorithm[13]. The `Contains` and `ContainsWithExtreme` methods are shown in Listing 4.7.

[11] One could consider moving the method to the `Point2D` class, which will make typical usage `point.LiesInsideArea(area)`. However, this introduces a dependency from `Point2D` to `Area`, which is undesirable.

[12] The `null` check also present in the method is omitted for brevity.

[13] `https://en.wikipedia.org/wiki/Point_in_polygon#Ray_casting_algorithm`

```
1  /// <summary>
2  /// Determines if the <paramref name="point" /> lies inside the area, or
   ↪  exactly on the line of the area.
3  /// </summary>
4  /// <param name="point">The point to check.</param>
5  /// <returns><c>True</c> if <paramref name="point" /> lies inside the area,
   ↪  <c>false</c> otherwise.</returns>
6  /// <exception cref="ArgumentNullException">Thrown when <paramref name="point"
   ↪  /> is <c>null</c>.</exception>
7  public bool Contains([NotNull] Point2D point)
8  {
9      if (point == null)
10     {
11         throw new ArgumentNullException(nameof(point));
12     }
13
14     return ContainsWithExtreme(point, new Point2D(Points.Select(p =>
       ↪  p.X).Max() + 1, point.Y))
15            && ContainsWithExtreme(point, new Point2D(Points.Select(p =>
              ↪  p.X).Min() - 1, point.Y))
16            && ContainsWithExtreme(point, new Point2D(point.X, Points.Select(p
              ↪  => p.Y).Max() + 1))
17            && ContainsWithExtreme(point, new Point2D(point.X, Points.Select(p
              ↪  => p.Y).Min() - 1));
18 }
19
20 private bool ContainsWithExtreme(Point2D point, Point2D extremePoint)
21 {
22     var lineToExtreme = new Line(point, extremePoint);
23     var pointsFound = new HashSet<Point2D>();
24
25     bool isHorizontal = lineToExtreme.IsHorizontal();
26
27     var i = 0;
28
29     if (Equals(Points[i], point))
30     {
31         return true;
32     }
33
34     do
35     {
36         int nextIndex = (i + 1) % Points.Count;
37         var currentLine = new Line(Points[i], Points[nextIndex]);
38
39         if (Equals(Points[nextIndex], point))
40         {
41             return true;
42         }
```

```
43
44          double distance =
            ↪  SpatialCalculusSingleton.Instance.EuclideanDistance(currentLine,
            ↪  point);
45          if (distance < 1e-5)
46          {
47              return true;
48          }
49
50          ReadOnlyCollection<Point2D> intersectingPoints =
            ↪  SpatialCalculusSingleton.Instance
51                                      .Intersect(currentLine, lineToExtreme);
52          if (intersectingPoints != null && intersectingPoints.Any())
53          {
54              var intersectingLine = new Line(intersectingPoints.First(),
                ↪  intersectingPoints.Last());
55              if (isHorizontal && intersectingLine.IsHorizontal()
56                  || !isHorizontal && intersectingLine.IsVertical())
57              {
58                  return true;
59              }
60
61              foreach (Point2D point2D in intersectingPoints)
62              {
63                  pointsFound.Add(point2D);
64              }
65          }
66
67          i = nextIndex;
68      } while (i != 0);
69
70      return pointsFound.Count % 2 == 1;
71  }
```

Listing 4.7: `Contains` and `ContainsWithExtreme` before refactoring

The ray casting algorithm is relatively simple in terms of execution: all that is needed is to construct a single line from the point to any direction and counting the number of times this line intersects with the edges of the polygon. However, two issues commonly arise during implementation, which partially explain how the method came to look like this.

Firstly, the algorithm is unpredictable whenever the point to check lies exactly on the border of the polygon. Conceptually this may make sense (it is debatable whether a point that lies on the edge is 'contained' with the polygon), but this implementation requires that these points are considered to be 'contained'. In the original code, this was done by checking if the point is on the same location as one of the corners of the area (lines 29–32 and 39–42), or if it lies exactly on one of the walls (edges) of the area (lines 44–48). However, this check is independent of the ray casting algorithm and is therefore cleaner

outside of this area. The checks are moved up to the Contains method and can actually be replaced with a single call to the newly created PointIsOnEdge method. This means these steps have actually reduced some (obscure) duplication between this method and the ContainsInside method.

The other common issue occurs whenever the constructed ray exactly crosses an intersection between two edges of the polygon. This will commonly be counted as two intersections between the ray and the polygon (after all, it crosses both edges), while it should only be counted once. In the original ContainsWithExtreme method, the fix for this lies with the pointsFound set, to avoid the double counting. A similar issue occurs whenever the ray exactly overlaps with one of the edges with the polygon. The originally attempted fix can be seen in lines 54–59. These lines check whether the ray (lineToExtreme) and the edge (currentLine) overlap, using the fact that the ray is always horizontal or vertical, and returning true whenever there is overlap. These fixes did not catch all edge cases, which prompted to developer to call the ContainsWithExtreme method in four directions, and only returning true whenever they all agree.

Another fix for this is simpler and does correctly catch all edge cases [123]. Both cases occur when one or multiple points of the polygon exactly cross the ray. If a horizontal line is chosen[14] (i.e., a line with constant y coordinate), this means that the y coordinate of the point is exactly the same as the y coordinate of the line. If all such points are moved slightly up (or below), the issue is solved. In the first instance of the issue (exactly on a corner), only one of the two edges will be counted, as the other edge does not intersect anymore due to the displacement. In the other instance (overlapping lines), the lines will not overlap anymore, either counting two intersections when the points are moved up (of the edges that are connected to the previously overlapping edge), or none (when they were moved down). It does not matter whether the points are moved up or down (as long as they are all moved in the same direction), as all that counts is if the number of intersections is even or uneven.

Replacing the old fix with this standard fix (a *Substitute Algorithm* refactoring[47]) allows the removal of all compensatory code relating to the old fix. In particular, lines 54-64 can be removed, simply increasing an intersection counter every time an intersection is found. In fact, it is now more visible that checking whether two lines intersect is rather unwieldy, calling the SpatialCalculusSingleton and checking whether there are any intersecting points. The algorithm does not need to care about those details, it simply wants to ask the Line if it intersects with another Line. Therefore, this functionality is moved to a IntersectsWith extension method for the Line class, so the call can simply be lineToExtreme.IntersectsWith(currentLine)[15]. Additionally, the ContainsWithExtreme method now only needs to be called once, further cleaning up the code.

Now that the method is cleaned up considerably, it becomes clear that entire loop

[14]The ray may be any line that is guaranteed to end outside the polygon, but often a horizontal or vertical line is chosen due to ease of construction and calculation.

[15]The functionality is added as an extension method, because it adding directly to the Line class would add an unwanted dependency from Core.Data to Domain.Data, as the SpatialCalculus implementation lies in the Domain.Data namespace, whereas Line resides in the Core.Data namespace. While it may be desirable to have the method contained within Line, that refactoring is considered out of scope for this thesis.

can be replaced with a simple `Count` LINQ[16] statement, i.e., `areaLines.Count(line => line.IntersectsWith(lineToExtreme))`, where `areaLines` are the edges of the `Area`, corrected to avoid overlap with the ray where applicable.

Finally, now that the method is cleaner and the purpose of the variables are clearer, it becomes visible that the names do not convey their meaning fully. The code keeps talking about `Extreme`, while literature would call the line a `ray`. Therefore, the `lineToExtreme` is renamed simply to `Ray`. The method name should properly reflect what it does and what the return value means. A name such as `ContainsWithRay` does not reflect what the method actually does, only that it uses a ray. Therefore, the name `RayCrossesAreaUnevenAmountOfTimes` is chosen. The context of it being used to check whether the `Area` contains a `Point` is given by the fact that it is called by the `Contains` method.

The end result is shown in Listing 4.8. The functionality is more concise and self-explanatory due to the use of short methods with expressive names. Note that the `AvoidLineOverlapWithRay` had documentation comments added. While not needed due to project guidelines (because it is a private method), it serves as clarifying 'obscure' functionality, one of the rare occurrences of a 'good comment' [82]. The comment inside the `AvoidPointOverlapWithRay` serves as a hint to the ReSharper plugin that the exact comparison between floating points is done on purpose and should not generate a warning, needed to comply with the project guidelines of having no ReSharper warnings.

```
1  /// <summary>
2  /// Determines if the <paramref name="point" /> lies inside the area, or
   ↪  exactly on the line of the area.
3  /// </summary>
4  /// <param name="point">The point to check.</param>
5  /// <returns><c>True</c> if <paramref name="point" /> lies inside the area,
   ↪  <c>false</c> otherwise.</returns>
6  /// <exception cref="ArgumentNullException">Thrown when <paramref name="point"
   ↪  /> is <c>null</c>.</exception>
7  public bool Contains([NotNull] Point2D point)
8  {
9      if (point == null)
10     {
11         throw new ArgumentNullException(nameof(point));
12     }
13
14     return PointIsOnEdge(point) ||
15            RayCrossesAreaUnevenAmountOfTimes(point);
16 }
17
18 private bool PointIsOnEdge(Point2D point)
19 {
```

[16] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

```
20      return Walls.SelectLocations().Any(line => line.IsOnLine(point, tolerance:
        ↪ 1e-5));
21  }
22
23  private bool RayCrossesAreaUnevenAmountOfTimes(Point2D point)
24  {
25      var ray = CreateRayToTheRightOfPoint(point);
26      var areaLines = Walls.SelectLocations().Select(line =>
        ↪ AvoidLineOverlapWithRay(line, point.Y));
27      int intersections = areaLines.Count(line => line.IntersectsWith(ray));
28      return intersections % 2 == 1;
29  }
30
31  private Line CreateRayToTheRightOfPoint(Point2D point)
32  {
33      var maximumX = Points.Max(p => p.X) + 1;
34      return new Line(point, new Point2D(maximumX, point.Y));
35  }
36
37  /// <summary>
38  /// Modifies the points of <paramref name="line"/> to not have exactly the
    ↪ same y coordinate as <paramref name="yCoordinate"/>
39  /// This prevents incorrect results from the Contains algorithm when the ray
    ↪ exactly overlaps a <see cref="Line"/>
40  ///   or is exactly on the crossing between two <see cref="Line"/>s.
41  /// See also http://alienryderflex.com/polygon/.
42  /// </summary>
43  /// <param name="line">The line to modify</param>
44  /// <param name="yCoordinate">The y coordinate to avoid</param>
45  /// <returns>A modified <paramref cref="Line"/> where the start and end point
    ↪ are neither exactly on <paramref name="yCoordinate"/>.</returns>
46  private static Line AvoidLineOverlapWithRay(Line line, double yCoordinate)
47  {
48      return new Line(AvoidPointOverlapWithRay(line.Start, yCoordinate),
49                      AvoidPointOverlapWithRay(line.End, yCoordinate));
50  }
51
52  private static Point2D AvoidPointOverlapWithRay(Point2D point, double
    ↪ yCoordinate)
53  {
54      // ReSharper disable once CompareOfFloatsByEqualityOperator
55      return point.Y == yCoordinate ? new Point2D(point.X, yCoordinate + 1e-7) :
        ↪ point;
56  }
```

Listing 4.8: `Contains` after refactoring (documentation comments not shown)

4.5 `MainViewModel` class

The `MainViewModel` class constructs the application and its various GUI elements, and handles events done by users in the main window of the application, updating the state appropriately. This has however caused it to have multiple responsibilities, a violation of the Single Responsibility Principle. Therefore, functionality must be moved and extracted to other classes.

```
1  barnLayoutFactories:IDrawingXYShape[]
2  barnLayoutUpdateFactories:IDrawingXYShape[]
3  project:Project
4  activeBarn:Barn
5
6  MainViewModel()
7
8  EditorModel:EditorModel
9  RibbonViewModel:RibbonViewModel
10 ImportMapFileCommand:ICommand
11 ExportRobotsRoutesCommand:ICommand
12 SelectRobotCommand:ICommand
13 SelectBarnCommand:ICommand
14
15 RibbonViewModelOnPropertyChanged(object sender, PropertyChangedEventArgs
   ↪  e):void
16 OnDrawGeometry(object obj):void
17 OnSelectRobot(object obj):void
18 OnSelectBarn(object obj):void
19 OnSuccess(Barn barn):void
20 OnImportMapFile(object obj):void
21 OnExportRobotsRoutes(object obj):void
22 OnRoutesUpdated():void
23 UpdateEditorModel():void
24 UpdateRibbonBarnViewModels(Barn newBarn):void
25 CreateRibbonRobotViewModel(Barn barn):RibbonBarnViewModel
26 UpdateRibbonRobotViewModels(IEnumerable<IRobot> newRobots):void
27 CreateRibbonRobotViewModel(IRobot robot):RibbonRobotViewModel
28 RemovePreviousBarnLayout(IEnumerable<Type> drawables):void
```

Listing 4.9: Outline of the `MainViewModel` class

Inspecting the class and its outline (shown in Listing 4.9), a few responsibilities can be distinguished (the other methods are all called by the mentioned methods):

- Initializing and keeping track of the various models and event handlers for the GUI: `EditorModel`, `RibbonViewModel` and the four `ICommand` properties.
- Handling the events: importing a barn file (`OnImportMapFile`), exporting routes (`OnExportRobotsRoutes`), selecting a new robot in the ribbon (`OnSelectRobot`)

and selecting a new barn in the ribbon (OnSelectBarn).

- Keeping track of factories for constructing different elements in the editor (barnLayoutFactories and barnLayoutUpdateFactories), and therefore which elements should be drawn and which should not.

The first two points seem reasonable: the MainViewModel can be seen as central component of the system and the connecting component between the various elements of the GUI (the Ribbon allowing users to switch between barns and robots, and the Editor where users can view the layout and draw routes, specifically). Keeping track of the state and handling various events, to pass them between the GUI components, is logical functionality for such a component.

However, this functionality should be limited to calling methods on the respective classes as much as possible, keeping the logic as close to the actual components as much as possible. This can be greatly improved. For example, when selecting another Barn or Robot in the Ribbon, one of the methods called is the OnDrawGeometry method. This method of 23 lines mostly calls methods on the EditorModel, often calling methods on the properties of the EditorModel. This violation of the Law of Demeter is rampant across the entire class. The OnDrawGeometry functionality should therefore be moved to the BarnLayoutEditorModel class (which is the implementation of EditorModel). Another example is the construction of a Barn when a file is imported, this should not be the responsibility of the MainViewModel.

The third mentioned responsibility has no fit in the class and only related to the Editor. Therefore, it should also be moved to BarnLayoutEditorModel.

Clearly, the MainViewModel class has a lot of functionality that should be moved elsewhere, which was done in the refactorings. This will be discussed from top to bottom.

Beginning at the constructor, 15 lines of code can be seen that directly modify the EditorModel, setting it up. These lines are moved to the constructor of the BarnLayoutEditorModel.

The constructor also constructs a RibbonViewModel with two components: a RibbonBarnViewModel and a RibbonRobotViewModel. These two names are very similar, and can be considered disinformation [82]: the difference between one being for Barn and the other for Robot is 'hidden' in the middle of the name. This distinction should be made more explicit, by moving it to the front of the class name. Additionally, both classes are not actually view models for ribbons, but for gallery items[17]. Better names are therefore BarnGalleryItemViewModel and RobotGalleryItemViewModel.

Renaming the methods in the MainViewModel accordingly, it is immediately visible that the confusing class names did already cause incorrect naming. The class contains two methods called CreateRibbonRobotViewModel, one taking a IRobot parameter, the other taking a Barn. The second one should actually have been called CreateRibbonBarnViewModel!

The RibbonViewModelOnPropertyChanged is bound to the PropertyChanged event on the RibbonViewModel. From the EditorModel, it retrieves the current route

[17] https://docs.microsoft.com/en-us/windows/desktop/uxguide/cmd-ribbons

and sets the `DrivingDirection` for the route to the `DrivingDirection` that was selected in the `RibbonViewModel`. This can be seen as *Inappropriate Intimacy* [47], as the `MainViewModel` should not concern itself with how the `EditorModel` handles the `DrivingDirection`. Therefore, the functionality of retrieving the current route and setting its direction was moved to the `BarnLayoutEditorModel` class in a new method called `SetCurrentDirection`. The `MainViewModel` can then call this method, only acting as a 'message passer' between the `RibbonViewModel` and the `BarnLayoutEditorModel`.

The `OnDrawGeometry` method is called whenever changes have been made in the shapes that are on screen, calling some methods on the `EditorModel` that are needed for it to start drawing. It takes an `object` parameter, which is then converted to a `bool`, so the parameter is actually a flag. Martin is clear about flag arguments: "Passing a boolean into a function is a truly terrible practice". Luckily, in this case, the parameter is only passed as `true`, so it can be removed along with all code that would execute whenever the flag would have been `false`, being dead code.

The rest of the functionality is all calling methods on or editing properties of the `EditorModel`, so it is moved to the `BarnLayoutEditorModel` class. The only exception is the following statement: `drawableRoute.LayerAdding += (sender, args) => OnRoutesUpdated();`. This adds an event to the current route being drawn, calling a method that is within the `MainViewModel`. This method can not be moved to the `BarnLayoutEditorModel`, as it acts on the `RibbonViewModel`, and a dependency between those two is unwanted. However, the statement is unclean: the `MainViewModel` should not have to know about `drawableRoute` and the `LayerAdding` event name is not explanatory enough: What does '`LayerAdding`' mean? What has happened when the event was fired?. The `OnRoutesUpdated` method name provides a hint: the event is fired when a route is added. The `MainViewModel` should not care how that works. Therefore, a `RouteAdded` event is added to the `BarnLayoutEditorModel` class, with the `MainViewModel` binding the `OnRoutesUpdated` to it in its constructor. The `BarnLayoutEditorModel` is then responsible for having the event invoked when relevant, and thus binding the event to the `drawableRoute`.

Finally, the name `OnDrawGeometry` needs improvement. It is named like it is an event method, but it is only directly called by other methods. Additionally, it does not explain what it does, only when it is called. Therefore, it is renamed to `BeginDrawing`, to signify that it initiates drawing.

The `OnSelectRobot` method is called whenever a new `Robot` is selected in the GUI, updating the current state and calling the `UpdateEditorModel` method. This method again mostly operates on the `EditorModel` class, and thus should be moved there. The name is also a very clear sign for this: why should the `MainViewModel` update the `EditorModel`? The method is therefore moved to the `BarnLayoutEditorModel` class and simply renamed `Update`.

The `OnSelectBarn` method is called whenever a new `Barn` is selected in the GUI. It, together with the `OnSuccess` method it calls, updates the current state, constructs new items for the `RibbonViewModel`, and removes and constructs the new layout for the `EditorModel`. Clearly, the last two should be responsibilities of the `RibbonViewModel` and

BarnLayoutEditorModel classes respectively. Therefore, these functionalities are moved there, simplifying the method from about 25 lines in the MainViewModel to the 3 lines as shown in Listing 4.10.

```
private void SelectBarn(Barn barn)
{
    activeBarn = barn;
    EditorModel.SelectBarn(barn);
    RibbonViewModel.SelectBarn(barn);
}
```

Listing 4.10: The refactored version of selecting a new Barn.

The OnImportMapFile method is called whenever the user requests to import a new layout. It handles opening the dialog, calls the BarnImporter to import the data, builds the Barn, adds it to the GUI and selects it. The construction of the Barn should not be the responsibility of the MainViewModel, so this is moved to the Barn class, creating a *Factory Method* [47]. The BarnImporter calls this Factory Method after successful import, so the MainViewModel simply receives the Barn object.

Adding the new Barn is moved to its own AddNewBarn method, which simply updates the state and calls the RibbonViewModel to handle adding the new Barn.

Selecting the Barn has a fair amount of duplication with the OnSelectBarn method. Its functionality is extracted into the SelectBarn method shown in Listing 4.10, which is then directly called from the OnImportMapFile method, reducing duplication.

Finally, the barnLayoutFactories and barnLayoutUpdateFactories fields should be moved to the BarnLayoutEditorModel. The contents of these fields are an array of factory classes that are able to create objects representing the various shapes (e.g., walls, markers, areas, etc.) to be drawn inside the GUI. These fields are used in two scenarios: the first is used whenever a new BarnLayout needs to be drawn, removing all previous shapes defined by barnLayoutFactories, and then redrawing them for the new layout. The second is used whenever a new Robot is selected and only part of the layout needs an update, removing and redrawing the shapes defined by barnLayoutUpdateFactories, which is a subset of barnLayoutFactories. The names of the fields are not very clear (the barnLayoutFactories does not contain different factories that create barnLayouts), most importantly not mentioning that they create shapes, so they are renamed layoutShapeFactories and updateShapeFactories.

For removing previous shapes, the RemovePreviousBarnLayout method is called. This name is not entirely correct, as it simply removes the shapes with the types specified, which is not always an entire BarnLayout. It is therefore renamed to RemoveShapes. Additionally, the functionality for creating and adding shapes is duplicated in two locations, so is extracted to a method CreateAndAddShapes. Both are also moved to the BarnLayoutEditorModel class.

Figure 4.3 shows an overview of the classes before and after refactoring. The MainViewModel class was reduced considerably in size and now has little knowledge of

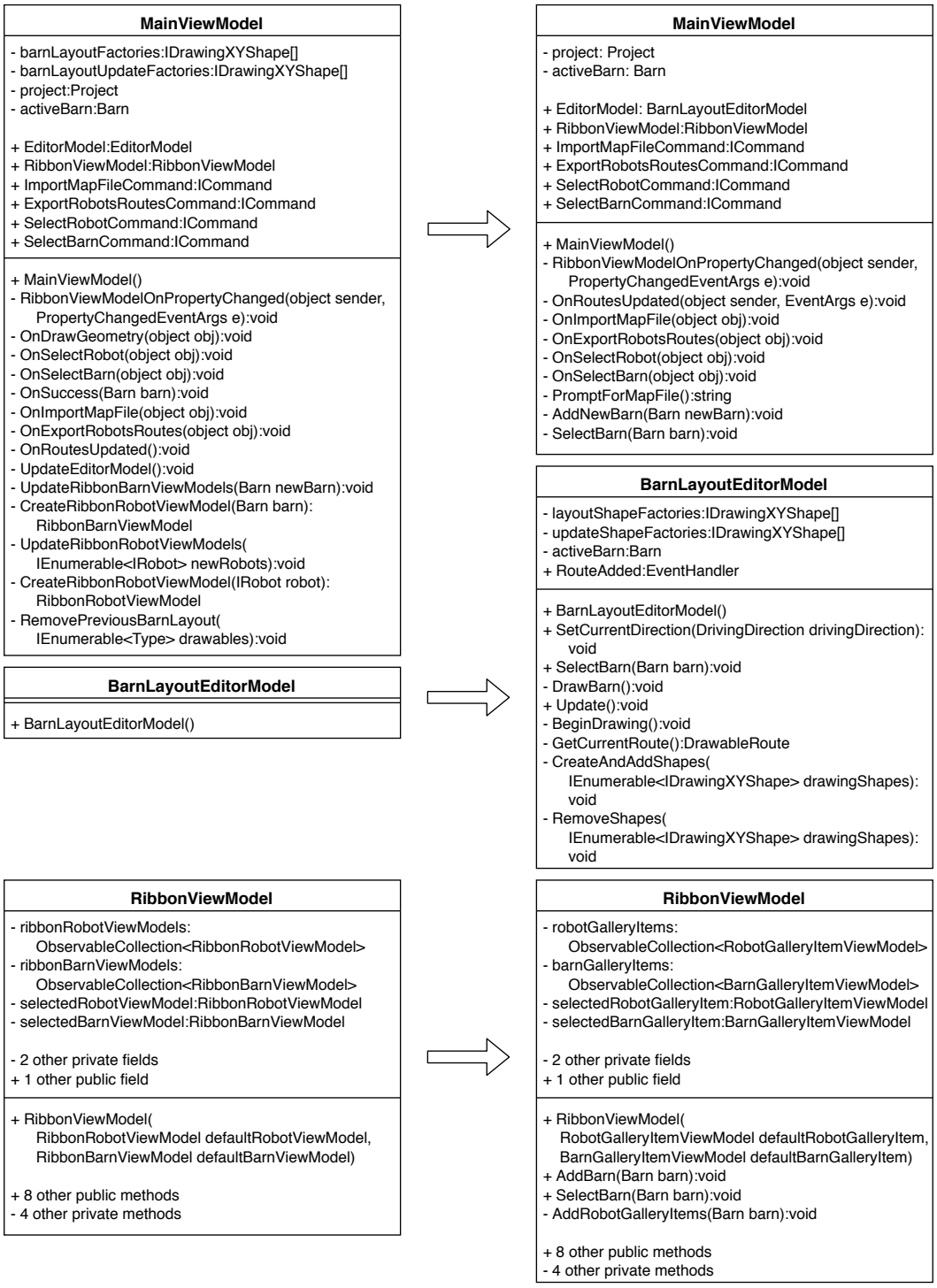


Figure 4.3: Classes `MainViewModel`, `BarnLayoutEditorModel` and `RibbonViewModel` before and after refactoring. Some members of the `RibbonViewModel` class were omitted for brevity.

the internals of the EditorModel and the RibbonViewModel classes.

4.6 ManureRobot class

The IRobot interface represents a robot in the barn. It has a single implementation: ManureRobot. The IRobot has 10 properties describing various attributes of the robot. Listing 4.11 shows the ManureRobot class before refactoring.

```
1  public class ManureRobot : IRobot
2  {
3      private ManureRobot(string name, string model, string machineId, double
       ↪  padding, double scraperWidth, double length, double width,
       ↪  HomeTransponder transponder, PowerDevice powerDevice,
       ↪  IEnumerable<SprayerDevice> sprayers)
4      {
5          Power = powerDevice;
6          Name = name ?? "Unknown";
7          Model = model ?? "Unknown";
8          MachineId = machineId ?? "Unknown";
9          PaddingFromWall = padding;
10         ScraperWidth = scraperWidth;
11         Length = length;
12         Width = width;
13         HomeTransponder = transponder;
14         Sprayers = sprayers ?? Enumerable.Empty<SprayerDevice>();
15     }
16
17     public string Name { get; }
18     public IEnumerable<SprayerDevice> Sprayers { get; }
19     //8 other properties omitted
20
21     public sealed class Builder : IBuilder<IRobot>
22     {
23         private readonly Range<double> oneOrGreaterRange = new
           ↪  Range<double>(1, double.MaxValue);
24
25         private readonly List<SprayerDevice> sprayers = new
           ↪  List<SprayerDevice>();
26         private string robotName;
27         // 8 other properties omitted
28
29         public Builder(PowerDevice powerDevice, HomeTransponder transponder)
30         {
31             robotPowerDevice = powerDevice ?? throw new
               ↪  ArgumentNullException(nameof(powerDevice));
32             robotTransponder = transponder ?? throw new
               ↪  ArgumentNullException(nameof(transponder));
```

```
33          }
34
35          public Builder WithName(double name)
36          {
37              robotName = name;
38              return this;
39          }
40          // 7 more 'With...' methods omitted
41
42          public IRobot Build()
43          {
44              var robot = new ManureRobot(robotName, robotModel, robotMachineId,
                ↪  paddingFromWall, robotScraperWidth, robotLength, robotWidth,
                ↪  robotTransponder, robotPowerDevice, sprayers);
45
46              if (!oneOrGreaterRange.InRange(robot.ScraperWidth))
47              {
48                  throw new InvalidOperationException(string.Format(
                    ↪  Resources.Builder_Build_InvalidScraperWidth,
                    ↪  robot.ScraperWidth, oneOrGreaterRange));
49              }
50              // 2 more similar range checks omitted
51
52              return robot;
53          }
54      }
55  }
```

Listing 4.11: `ManureRobot` class before refactoring. The class is slightly edited for brevity: documentation comments have been removed, some blank lines have been omitted and some functionality that is very similar to other functionality was also removed.

The large amount of parameters has caused the introduction of unclean code in the `ManureRobot` class: a constructor with 10 parameters. This is clearly a violation: as discussed in Section 4.3.3, Martin recommends a maximum of 3 parameters [82]. This may hinder understanding, a large number of parameters takes a lot of conceptual power to understand [82]. Additionally, because there is no natural ordering to these parameters (e.g., is `robotModel` or `robotMachineId` first?), it is easy to mess up their ordering when calling the constructor [82].

This violation is partly mitigated by the constructor being private, with the only caller being the nested `ManureRobot.Builder` class[18] (implementing the Builder design pattern [51]). This means outside code will never have to call the unclean constructor, which lessens the risk: the code needs to be read only once. However, unclean code that is only called once is still unclean code.

[18] `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/nested-types`

To improve this, first private set methods for the properties are introduced. In C#, this can be doing by adding a set accessor to the properties[19] and making it private[20]. Because the ManureRobot.Builder is a nested class inside ManureRobot, it can directly set the private properties. This way the properties are directly referenced and set using their names, improving clarity and ensuring the order does not matter anymore. The long list of setting the properties directly is simplified by using the object initializer notation[21].

The constructor was also responsible for setting default values for the Name, Model and MachineId properties when none was provided. This functionality is moved to the Build method of the Builder. The constructor is now unused, so can be removed. However, when the constructor is omitted, the compiler automatically generates a public empty constructor[22]. This is unwanted, as this would allow direct construction of the ManureRobot class, which was not possible before removing the constructor. Therefore, an empty private constructor should remain.

```
new ManureRobot.Builder(powerDevice, transponder)
    .WithName(name)
    .WithLength(length)
    // etc..
    .Build()
```

Listing 4.12: Typical ManureRobot.Builder usage

Listing 4.12 shows typical usage of the ManureRobot.Builder class. The Builder is first constructed by passing a PowerDevice and HomeTransponder, which are verified to be not null. Then, the properties are set using their WithProperty methods. Finally, the Build method is called, which constructs the actual ManureRobot object and verifying some properties before finally returning the object. The class thus provides a clean interface to construct ManureRobot objects and ensures they are valid.

Because the properties of the ManureRobot class could only be set using the constructor, the Builder class needed to keep track of all properties set using the With methods itself, using its own private fields. This means that all properties of the ManureRobot are essentially duplicated within the Builder class. However, now that the ManureRobot class is refactored to allow setting its properties directly, this duplication can be removed. The constructor of the Builder can construct an empty private ManureRobot object. The With methods can then directly set the properties on this object. As the object is only returned after Build is called (and thus validated), it is still ensured that only valid ManureRobot instances can be used in other code.

[19] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties

[20] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/restricting-accessor-accessibility

[21] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers

[22] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/using-constructors

The only property that cannot be easily replaced this way is the `Sprayers` property. This property is a collection of `SprayerDevice` objects, represented as `IEnumerable<SprayerDevice>`. The `WithSprayerDevice` method does not replace the collection, but adds the specified `SprayerDevice` to it. However, one cannot add elements to an `IEnumerable`. In the original implementation, the `Builder` uses a private field of type `List<SprayerDevice>` to which the `SprayerDevice` can be added, and passing that to the `ManureRobot` (which is valid, as `List` implements `IEnumerable`). For consistency with the other properties, this field is moved to the `ManureRobot` class and the `Sprayers` property is modified so it returns the private field. The `Builder` then simply adds the elements to the field in the `ManureRobot` class.

Two other properties, `HomeTransponder` and `PowerDevice`, are not set using a `With` method, but directly in the constructor of the `Builder`. This is done to signify that they are mandatory, and thus cannot be left `null`. However, this can also be done with a `null` check in the `Build` method, so that the validation of these properties is the same as the other properties that have validation. This improves consistency with the other properties both in terms of setting and in terms of validation. Therefore, the parameters are removed from the constructor of the `Builder`, the methods `WithHomeTransponder` and `WithPowerDevice` are introduced and the `null` check of both properties have been moved to the `Build` method.

The `Build` method now has three distinct stages: First, some properties have default values set if they are `null`. Secondly, the value of some properties is validated such that no `ManureRobot` with invalid values is constructed. Finally, the `ManureRobot` is returned. The first two stages are extracted into their own methods (`SetDefaults` and `Validate` respectively), so their purpose is clearer and the `Build` method reads cleaner.

The final refactored version is shown in Listing 4.13. Note that the use of `I` as a prefix for interfaces is discouraged by Martin [82], the name of the `IRobot` class is left as-is to comply with project guidelines.

```
1  public class ManureRobot : IRobot
2  {
3      private ManureRobot() { }
4
5      private readonly List<SprayerDevice> sprayers = new List<SprayerDevice>();
6
7      public string Name { get; private set; }
8      public IEnumerable<SprayerDevice> Sprayers => sprayers;
9      // 8 other properties omitted
10
11     public sealed class Builder : IBuilder<IRobot>
12     {
13         private readonly Range<double> oneOrGreaterRange = new
           ↪  Range<double>(1, double.MaxValue);
14         private readonly ManureRobot robot;
15
16         public Builder()
17         {
```

```
18              robot = new ManureRobot();
19          }
20
21          public Builder WithName(string name)
22          {
23              robot.Name = name;
24              return this;
25          }
26          // 9 more 'With...' methods omitted
27
28          public IRobot Build()
29          {
30              SetDefaults();
31              Validate();
32
33              return robot;
34          }
35
36          private void SetDefaults()
37          {
38              robot.Name = robot.Name ?? "Unknown";
39              robot.Model = robot.Model ?? "Unknown";
40              robot.MachineId = robot.MachineId ?? "Unknown";
41          }
42
43          private void Validate()
44          {
45              if (robot.Power == null)
46              {
47                  throw new InvalidOperationException(␣
                    ↪ Resources.Builder_Build_InvalidPower);
48              }
49
50              if (robot.HomeTransponder == null)
51              {
52                  throw new InvalidOperationException(␣
                    ↪ Resources.Builder_Build_InvalidTransponder);
53              }
54
55              if (!oneOrGreaterRange.InRange(robot.ScraperWidth))
56              {
57                  throw new InvalidOperationException(string.Format(␣
                    ↪ Resources.Builder_Build_InvalidScraperWidth,
58                      robot.ScraperWidth,
59                      oneOrGreaterRange));
60              }
61              // 2 more similar range checks omitted
62          }
63      }
```

```
64 }
```

Listing 4.13: `ManureRobot` class after refactoring. Similarly to Listing 4.11, the class is lightly edited.

4.7 Singleton classes

The code contains four different classes with the `Singleton` suffix. These classes serve to ensure certain classes do not need to be instantiated more than one time (i.e., the Singleton design pattern [51]) and provide classes with a default implementation for the interfaces, without having to know it. Additionally, the instance can be overwritten, allowing the classes to be faked in testing. One of these, `StreamReaderSingleton`, is shown in Listing 4.14.

Each of the four classes are implemented exactly the same, except for the types they apply to (e.g., in the `SpatialCalculusSingleton`, `IStreamReader` is replaced with `ISpatialCalculus` and `StreamReaderWrapper` with `SpatialCalculus2D`). This suggests that the pattern can be abstracted away using generics, reducing duplication. This is not without downsides. For example, every user of a `Singleton` class then needs a dependency to the generic `Singleton` class. However, the reduction of duplication is worth it: Martin states that "Duplication may be the root of all evil in software." [82]. Visser et al. state "Duplication of source code should be avoided at all times" [132].

The `SetInstance` method of the classes deserves special attention. Usually, the Singleton pattern does not allow its instance to change. In fact, the main intent of a Singleton is to "[e]nsure a class only has one instance" [51]. The reason this code does allow changing the instance is because this method is used in test code to allow for replacing the `instance` with a fake, to reduce dependencies in the tests. The method is not intended for use in normal code, and the `internal` modifier[23] prevents such use in most code[24].

The `SetInstance` method poses two challenges. Firstly, methods that are only supposed to be called in test code, should not be in the code. Secondly, if the functionality of the `Singleton` classes were to be moved to a generic superclass, the access modifier of the `SetInstance` method needs to be made less restrictive to allow it to be derived, increasing the exposure of the method and therefore the risk it will be called. Using reflection[25], access modifiers can be bypassed, and thus the `instance` can be set directly without the need of a method for it within the `Singleton` classes. An added extra is that the code can then be moved to the test code, so the functionality is not included in the program itself.

[23] `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/internal`

[24] The `internal` modifier normally also prevents the use of the method in test code, as such code is located in another assembly. However, the `InternalsVisibleToAttribute` is used to circumvent this: `https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.internalsvisibletoattribute`

[25] `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection`

```
1  public sealed class StreamReaderSingleton
2  {
3      private static volatile IStreamReader instance = new
       ↪  StreamReaderWrapper();
4
5      /// <summary>
6      ///     Creates a new instance of <see cref="StreamReaderSingleton" />.
7      /// </summary>
8      private StreamReaderSingleton() {}
9
10     /// <summary>
11     ///     Gets the singleton instance.
12     /// </summary>
13     public static IStreamReader Instance
14     {
15         get
16         {
17             return instance;
18         }
19     }
20
21     /// <summary>
22     ///     Sets the singleton instance to <paramref name="reader" />.
23     /// </summary>
24     /// <param name="reader">The new instance to use.</param>
25     internal static void SetInstance(IStreamReader reader)
26     {
27         instance = reader;
28     }
29 }
```

Listing 4.14: `StreamReaderSingleton` class before refactoring.

With the `SetInstance` removed, the functionality can relatively easy be extracted into a generic `Singleton` superclass, using the *Extract Superclass* refactoring [47]. The `Singleton` classes are now completely empty, with the exception of the empty private constructor, disallowing explicit construction of the class. The final result is shown in Listing 4.15.

4.8 Other refactorings done

The previously mentioned refactorings were used explicitly as part of the experiment. Other refactorings have been done, but these were not the subject of a task during the experiment, either because tasks about that code were scrapped or because the refactorings were not consequential enough to warrant their own task (e.g., renaming a single variable). However,

```
1  public abstract class Singleton<TInterface, TImplementation>
2          where TInterface : class
3          where TImplementation : TInterface, new()
4  {
5      private static volatile TInterface _instance;
6
7      /// <summary>
8      ///     Creates a new instance of <see cref="TImplementation" />.
9      /// </summary>
10     static Singleton()
11     {
12         _instance = new TImplementation();
13     }
14
15     /// <summary>
16     ///     Gets the singleton instance.
17     /// </summary>
18     public static TInterface Instance => _instance;
19 }
20
21 public sealed class StreamReaderSingleton : Singleton<IStreamReader,
   ↪  StreamReaderWrapper>
22 {
23     private StreamReaderSingleton() { }
24 }
```

Listing 4.15: `Singleton` and `StreamReaderSingleton` classes after refactoring.

the participants working in the refactored code received the full code base, including these other refactorings. While the chance is low (and during analysis of the recordings no occurrence was seen), the participants might have encountered these refactorings and thus might have been influenced by them. Therefore, these refactorings are described here shortly, for completeness.

4.8.1 Removal of dead code

As the developer of the software used the ReSharper plugin heavily during development, code quality issues that are easily checked by tools such as ReSharper were rare in the code. One of these is the presence of dead code[26]. Martin is clear about such dead code: "When you find dead code, do the right thing. (...) Delete it from the system." [82].

While most forms of dead code were already removed during development, some still remained. Most of the remaining dead code were methods and classes that were only used by test code. As they were unused by the rest of the system, they are dead code and should be removed. However, ReSharper did not detect the code as dead code, as it was still

[26] https://www.jetbrains.com/help/resharper/Find_dead_code.html

called from somewhere, not making the distinction between program code and test code. Detection of such dead code was done by unloading all test projects in Visual Studio, and letting ReSharper rescan for dead code, removing all found instances. Afterwards, the test code was reloaded and the now failing tests were removed. About 500 lines of program code was removed this way.

4.8.2 Extension method classes

The code contains many classes containing extension methods[27]. These are not only implemented on classes not 'owned' by this code, but also on own code, which is explicitly discouraged by the Microsoft guidelines on extension methods [92]. The rationale is to avoid adding additional dependencies on the class across the different projects in the code. For example, extension methods on the `Line` class (which is in the `Core` namespace) that use classes from the `Domain` namespace are not desirable to be refactored to instance methods on the `Line` class, as that would introduce a dependency from the `Core` to the `Domain` namespace.

However, the code contains multiple cases where this rationale does not hold, and thus the methods should be refactored to instance methods on the classes. For example, the code contains two classes containing extension methods for the `Line` class, one in the `Core.Data` project and the other in `Domain.Data`. The first is in the same assembly as the `Line` class itself and thus no extra dependencies can be introduced by moving the methods[28]. The second also contains an extension method that would not introduce new dependencies, so is also moved.

In total, 9 extension methods have been moved.

4.8.3 `Domain.IO` project

The `Domain.IO` project concerns itself with importing barn layout data and exporting the same data, together with robot route data, from and to XML. Its organization seems messy at first sight. The namespaces are shown in Listing 4.16.

The `Surrogate` namespaces contain classes that serve as surrogate objects for the actual domain classes (e.g., the `LocationXmlSurrogate` class represents a `Location`). These only contain the properties that need to be serialized from/to the XML. The `Extensions` namespaces contain classes with extension methods, one for each of the surrogate classes. These methods deal with constructing surrogate objects from domain objects and vice versa.

It seems unnecessary to place this functionality in extension methods as opposed to within the surrogate classes: the surrogate classes are freely modifiable, it does not pollute the XML output (methods are not serialized) and it makes sense for the surrogate classes to depend on the domain classes, so extra dependencies are also not a reason to use extension methods. Implementing the functionality as extension methods may hurt the understandability of the system, and it is against Microsoft's guidelines for implementing extension

[27] `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods`

[28] One method remains, as it is actually an extension on `IEnumerable<Line>`

```
Domain.IO.Extensions
Domain.IO.Surrogates
Domain.IO.Export
    Domain.IO.Export.Extensions
    Domain.IO.Export.Surrogates
        Domain.IO.Export.Surrogates.Types
    Domain.IO.Export.TypeConverters
Domain.IO.Import
    Domain.IO.Import.Extensions
    Domain.IO.Import.Surrogates
        Domain.IO.Import.Surrogates.Types
```

Listing 4.16: Namespaces in the `Domain.IO` project before refactoring.

methods [92]. Therefore, the functionality with the `Extensions` classes is moved to the surrogate objects, removing all `Extensions` namespaces in the process.

One exception is the `BarnXmlSurrogateExtension.Transform` method, which is not used to converting from and to surrogate objects, but rather transforming locations to conform to the desired output format. This is refactored into its own `BarnXmlSurrogateTransformer` class so the method can more easily be split up without having to pass many parameters (using a *Replace Method with Method Object* refactoring [47][29]).

The existence of separate namespaces for importing and exporting is confusing at first, as one would expect them to be the same operation (but in reverse). However, the `Import` namespaces concern with the importing of the map layout, while the `Export` namespace concern with the exporting of the route data to be uploaded to the robots (along with the layout). This means the format of the XML is not exactly the same. Overlapping objects between the two are in the `Domain.IO.Surrogates` namespace.

Merging the two would not be considered clean, as the classes then have multiple reasons for change: when the import functionality needs to change and when the export functionality needs to change. This breaks the Single Responsibility Principle [84]. However, the names of the namespaces can be changed to more clearly show the difference between the two. `Import` is renamed to `LayoutImport` and `Export` is renamed to `RoutesExport`. The `Domain.IO.Surrogates` is renamed to `Domain.IO.Shared.Surrogates` to make it more clear that those objects are shared between the two functionalities.

[29] Also called *Replace Method with Command Object* in the newer editions of Fowler's book on refactoring [49].

Chapter 5

Results

This chapter will present the results of the performed experiment as described in Chapter 3. First the participants will be discussed, after which the resulting data for each assignment will be discussed individually.

5.1 Participants

The experiment started with 16 participants, 15 male and 1 female[1]. All participants filled in the pre-questionnaire as described in Section 3.5.1. This data was used to form balanced groups based on expected performance. Additionally, it allows for insight on the participants and their background going into the experiment.

Four participants filled in the questionnaire after the groups were formed (but before the experiment). In order to be able to form and balance the groups, the missing responses relevant for the estimated expertise were estimated based on available employee data. These were later compared to the actual responses from the participants, to validate group balance.

5.1.1 Programming experience

The participants varied widely in programming experience, ranging from recent starters with less than a year of professional development experience, to 24 years of professional experience. This is shown in Figure 5.1. Of the 16 participants, 5 participants reported less than a year of professional development experiment, with a further 5 participants having less than 5 years of experience.

Because of limited participant availability, participants were asked to only have at least a minimal (self-assessed) understanding of C#. It was expected that because the assignments contained only a minimal amount of programming, this would have been sufficient for them to successfully participate. Of the 16 participants, 5 participants reported less than a year of experience in C#, 5 participants had a year of experience and the other 6 participants had more than a year of experience. This means the participants were relatively inexperienced in programming in C#.

[1] For anonymity and simplicity, participants will always be referred to as he/him.

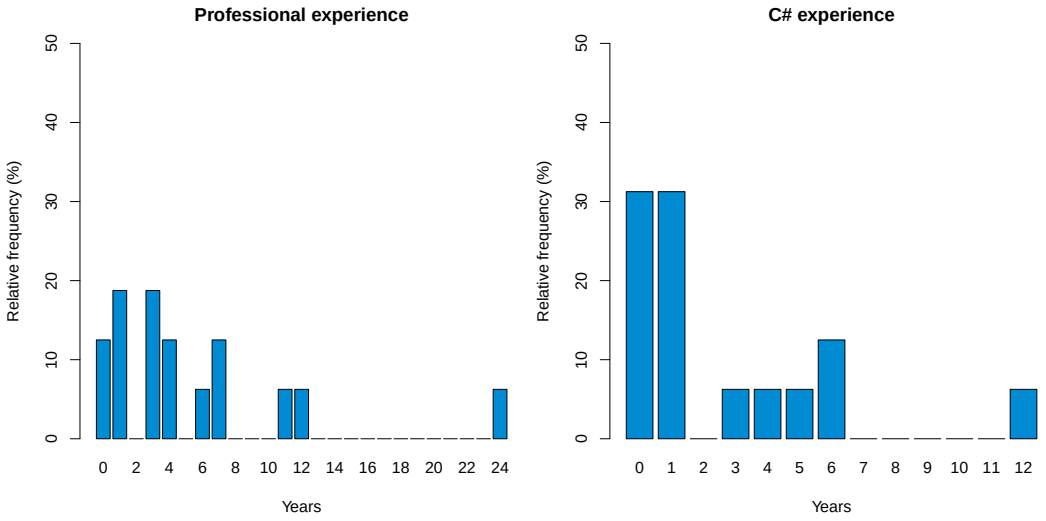


Figure 5.1: Participants experience in professional and C# development

Consultants at ALTEN can work in a variety of functions related to software engineering. This means not all participants develop object oriented systems in their daily work. Of the 16 participants, 5 carry the job title of 'Scientific Software Engineer'. Their work is in designing and implementing algorithms and simulations, in which object oriented design is usually not practiced. One participant is a 'Test Engineer', meaning his work is mostly in designing, implementing and executing software tests[2]. This is also not done in an object-oriented fashion. This does not mean these participants have no experience in object-oriented design: all participants have received schooling and/or training in object-oriented programming. The other 10 participants are 'Software Engineers' and are regarded to work in object-oriented systems daily.

Table 5.1 shows an overview of the participants that took part in the experiment.

5.1.2 Object-oriented knowledge

In the pre-questionnaire, the participants were asked a few questions about basic principles of object-oriented design: inheritance and overloading. These questions are adapted from the contents of a first year course on object-oriented programming and are similar to questions ALTEN asks in the application process for new consultants, so the participants are expected to perform well. The participants were given the code as shown in Listing 5.1[3], and were asked to indicate for each block if the code would give an error (either at compile time or runtime) and what the value for `c` would be after execution (if there was no error). In total, 8 mistakes could be made. Figure 5.2 shows the results. Half of the participants

[2]Note that these are not unit tests testing code, but software tests testing the application as a whole, possibly written in a different language than the application.

[3]Along with the clarification that `'A' + 1` would result in `'B'`

```
class SimpleCode
{
    public virtual char Encode(char character)
    {
        return (char) (character + 1);
    }
}

class SmartCode : SimpleCode
{
    public override char Encode(char character)
    {
        return Encode(character, 3);
    }

    public char Encode(char character, int key)
    {
        return (char) (character + key);
    }
}

// Block A
SimpleCode code = new SimpleCode();
char c = code.Encode('B');

// Block B
SimpleCode code = new SmartCode();
char c = code.Encode('C');

// Block C
SimpleCode code = new SmartCode();
char c = code.Encode('D', 4);

// Block D
SmartCode code = new SmartCode();
char c = code.Encode('D', 4);
```

Listing 5.1: Code given for the small quiz on object-oriented knowledge.

Participant	Function title	Most commonly used language	Professional development experience (years)	C# experience (years)	Refactoring	Document
P1	SE	C#	12	4	No	No
P2	SE	Python	1	0	Yes	No
P3	SSE	C++	1	1	No	Yes
P4	SSE	C#	7	6	Yes	No
P5	SE	C#	3	3	No	No
P6	SE	C#	3	5	Yes	No
P7	SE	C++	6	0	No	No
P8	TE	Python	0	0	No	No
P9	SE	C++	1	0	Yes	Yes
P10	SE	C#	24	12	Yes	Yes
P11	SSE	C#	11	6	No	Yes
P12	SE	C++	4	1	Yes	No
P13	SE	C++	0	1	Yes	Yes
P14	SSE	C++	3	1	No	Yes
P15	SSE	Python	7	0	Yes	Yes
P16	SE	C++	4	1	No	Yes

SE: Software Engineer
SSE: Scientific Software Engineer
TE: Test Engineer

Table 5.1: Participants data at the start of the experiment. Note that the 'C# experience' column might have higher values than in the 'Professional development experience' column, because it can include non-professional experience, such as experience gained during school or in free time.

answered all questions correctly. Conversely, half of the participants made some errors in what is regarded as basic knowledge of object-oriented programming [28].

It was expected that the participants would have performed better than the participants in a similar experiment performed by Ammerlaan [6]. Consultants at ALTEN receive explicit training in object-oriented programming, while Ammerlaan describes their participants as working and thinking in a procedural way [6]. While a higher percentage of participants in this experiment answered all questions correctly (50% versus 33%), the average number of correct answers is very similar (around 6.5). This means that the data does not support the claim that the participants in this experiment are more knowledgeable about basic object-oriented concepts than the participants in the experiment by Ammerlaan.

Because the participants of this experiment do not have experience with C# per se, the cause for errors might be found in language specific differences. Short conversations with a few of the participants that made errors in this quiz indicate that this is not the case.

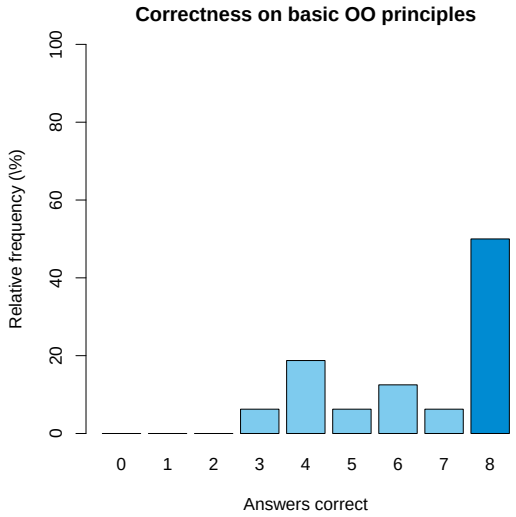


Figure 5.2: Correctness on questions on object-oriented knowledge.

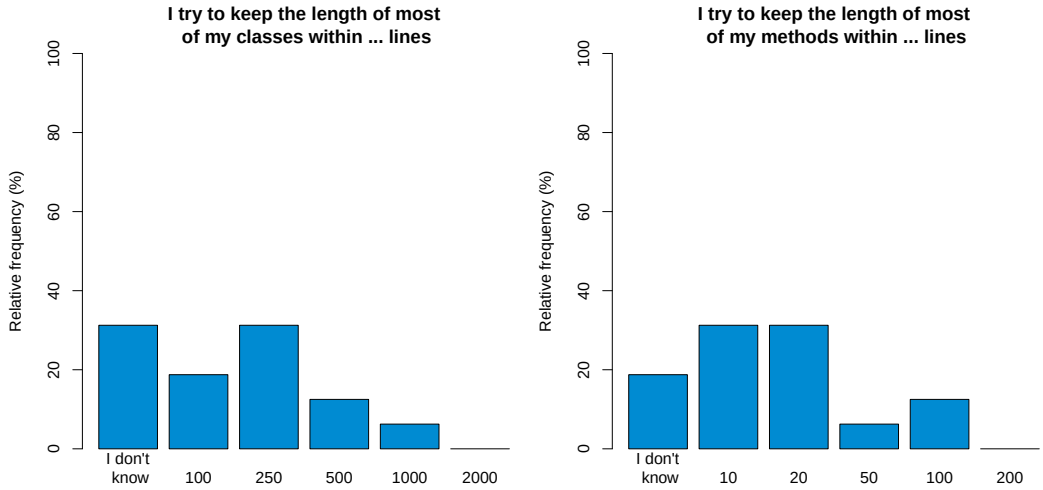


Figure 5.3: Size preferences for class and method sizes

Instead, the participants state that they believe this knowledge has faded because they do not regularly use the concepts and rely on their compiler for situations like those questioned[4].

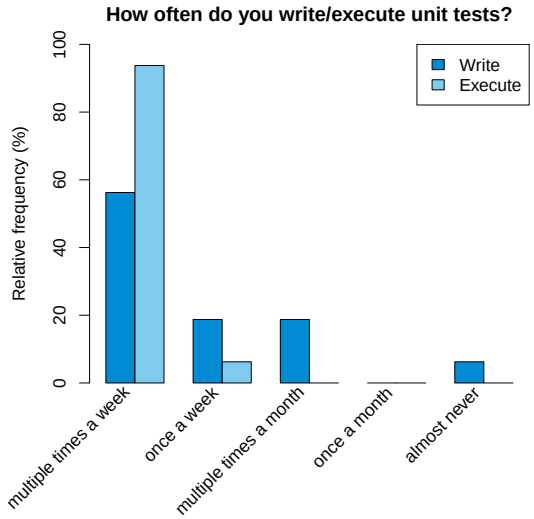


Figure 5.4: Testing frequency

5.1.3 Programming habits

Figure 5.3 shows the participant preferences for the size of classes and methods. Again, the questions are the same as those asked by Ammerlaan [6] to allow comparison. Results for class size are very similar, with a similar distribution between the answers. The participants in this experiment are more divided over the optimal method size, and on average even preferred longer methods.

It was expected that the participants who are familiar with Martin's work (indicated that they watched more than five of his videos or have at least partially read his book [82]) would answer with the smallest possible answers, seeing as that is a central thesis in his work, e.g., "But as with functions, smaller is the primary rule when it comes to designing classes." [82]. Instead, those most experienced with Clean Code most often chose a class size of 250 lines over a class size of 100 lines, and method sizes of 20 lines over method sizes of 10 lines. When one of the participants was asked about this, he explained that this was mostly for pragmatic reasons. Thus, a method of 10 lines is considered cleaner than a method of 20 lines, but it deemed not always worth the effort to get to that point.

Figure 5.4 shows how often the participants write and execute unit tests. From this, we see that almost all participants execute unit tests very frequently, and that the majority also write them frequently. Thus, we can expect them to be used to the unit tests and use them. In comparison to the study by Ammerlaan, the majority of their participants "wrote or executed unit tests only once a month or less frequent".

[4]Interestingly, if they would have encountered the situation from Block B in Listing 5.1, the compiler would not have been of much help in the question of whether the `Encode` method from `SimpleCode` or `SmartCode` is executed.

5.2 Experiment results

The experiment consisted of 6 exercises with a total of 13 assignments. The setup and purpose of these tasks is described in Section 3.6, the specific assignments will be described individually in the sections below.

The submitted answers and code were all manually examined in order to guarantee data validity and to check if there are any unexpected observations to be made. The screen recordings were only examined when extra insight was warranted (e.g., understanding surprising results).

For every assignment, the completion times and correctness percentages are reported, comparing the performance between the different groups. As discussed in Section 3.6, the timing results only show the time taken for correct solutions. During the experiment, some participants did not attempt certain assignment for various reasons, such as having to leave early. If a participant did not attempt a certain assignment, their data will be disregarded for that assignment, meaning they are also not counted as answering incorrectly. Because the timing results only show correct solutions and not every participant attempted every assignment, the number of observations (reported as n) may differ from assignment to assignment.

To test for statistical significance between the different groups in the timing results, the Mann-Whitney-Wilcoxon (MWW) test [59] is used. The choice for a non-parametric test is made because the sample sizes are too small to assume normality in the data. For calculation the `wilcox.test` function of the R `stats` base package is used, with the default settings [110]. For every test, the values of W, n_1, n_2 and p are reported as they are returned by R. The first group is always the data from the participants without the treatment, and the second group the data with treatment. This means that when $W > 0.5 * n_1 * n_2$ the completion times for the group without treatment are more often higher than lower when compared to the group with treatment, indicating improved performance for the group with treatment (as lower completion times are better), indicating a positive effect for the treatment (which may or may not be statistically significant). Because the data often contains ties, the p values are not exact, but are a normal approximation.

5.2.1 Deviations

During the experiment there were a few (unplanned) deviations from the experiment design and other oddities in the data. In this section, these will be discussed along with their impact on the results.

- One participant (P14) gave up after being unable to complete the first assignment and as a result did not complete any assignment in the experiment. As there is no resulting data from this participant, the rest of the analysis will exclude him, leaving 15 participants.

- One participant (P16) was placed in a group that was asked to read a document before the experiment. Despite explicit instructions to do so, the participant did not read the document before the experiment. Because he also did not use the document during

the experiment, the participant is, for the purpose of analysis, considered to be part of the group that did not receive the document.

- One participant (P5) was unable to get the recording working during the experiment. As the data from the recording is secondary, the participant completed the experiment and his data is included as normal.

- Due to an unfortunate error in the distribution of the code, all participants initially received the original version of the code, including those that should have received the refactored version. Luckily, the first assignment text had the participants look for a class that did not exist in the original version, so the error was quickly spotted and could be fixed immediately. The affected participants were able to start the experiment about 15 minutes later than initially planned.

- In the cases were there was no natural order, the order of the answer options in the multiple choice assignment questions was randomized. However, due to an oversight in the randomization process, the resulting permutations were always the same. This means that all questions with the same amount of options had the same answer. 4 of the 8 multiple choice questions had four options in 'randomized' order and for all the questions the correct answer was the second option. Luckily this was likely not noticed by the participants: after the experiment, several participants were asked if they had noticed this, and none of them indicated that they did.

- The timing data from one participant (P6) was found to be inconsistent with the times shown on the recording. When asked about this, the participant indicated that he was inconsistent with timekeeping during the experiment and some times were estimated as a result. The times were manually corrected by the author using the recording.

- One participant (P16) did not write down an end time for assignment 1 of exercise 2, as he did not finish it. However, he did solve the next two assignments. In order to use his solution time for assignment 2, the time the participant ended with assignment 1 and continued with the next assignment was estimated from the recording.

- One code file received from one participant (P2) contained some bogus text that caused the code to be unable to compile. The recording shows that the text was pasted in the code by accident and the participant either did not notice or was in a hurry, and quickly closed (and saved) the file without checking. As the text was not deliberately included, it was discarded in analysis, leaving only the deliberate changes made by the participant.

- One participant (P9) had some unexpected answers to the questions relating the document he had read. When asked about it, it became clear that the participant misunderstood the questions. Therefore, his answers for those questions were discarded. While this means some extra insight is lost, the rest of his data is still valid. The participant did read the document before the experiment, so the effects of the document on his performance (if any) are still present.

	Document	No Document
Refactored	P9, P10, P13, P15	P2, P4, P6, P12
Original	P3, P11	P1, P5, P7, P8, P16

Table 5.2: Final group membership

One of the effects from the above items is that the groups became unbalanced. Table 5.2 shows the final group assignment with P14 removed and P16 moved. Unfortunately, both participants were in the same group, leaving the group that have read the document beforehand and worked in the original code with only two participants.

5.2.2 Exercise 1

The first exercise focused on the `GetAreas` method, which was renamed `CreateAreas` and moved to the newly created `AreaCalculator` class in the refactoring (see also Section 4.3.3). The introductory text shortly explained what Areas are and what they represent. It explicitly mentioned the `GetAreas`/`CreateAreas`[5] method as the place where Areas are created. As such, almost all participants started their search there and most implemented their changes there. The introduction text read:

> To correctly identify the locations the `Robot`s can and cannot move to, the software splits up the layout of the `Barn` into `Area`s: One outer area representing the outer walls of the layout (which you cannot be outside of), and zero or more inner areas representing parts of the barn inside the outer walls that are also unreachable for the robots (i.e., you cannot be inside of them). The `BarnLayout` class gets its `Area`s from the `BarnLayoutHelper.GetAreas`/`AreaCalculator.CreateAreas` method.

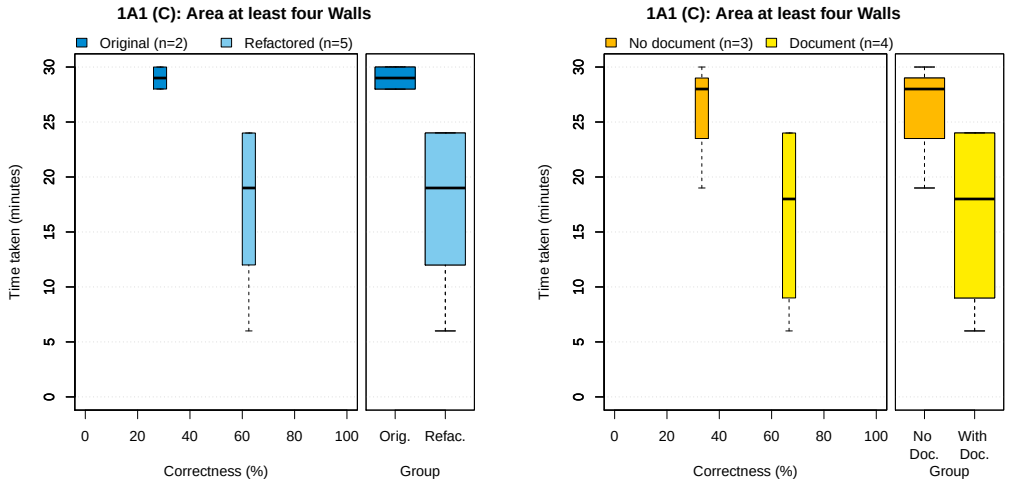
Assignment 1

The assignment text read:

> The customer has decided that an `Area` must contain at least four `Wall`s, so input that would lead to the construction of an `Area` of three or less `Wall`s should throw an `InvalidOperationException`, as with other invalid input. Implement this.

The assignment was intended to test whether the functionality of constructing `Area` objects was made more understandable by the refactoring. The expectation was that the participants in the refactored code would perform better, as both the `GetAreas`/`CreateAreas` and the

[5]The assignment texts for the participants working in the refactored code was slightly different when it referred to units of code of which the name has changed in refactoring. In this thesis, the texts are combined, with the differences shown as Original name/Refactored name



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.5: Correctness and completion times for correct solutions for assignment 1A1.

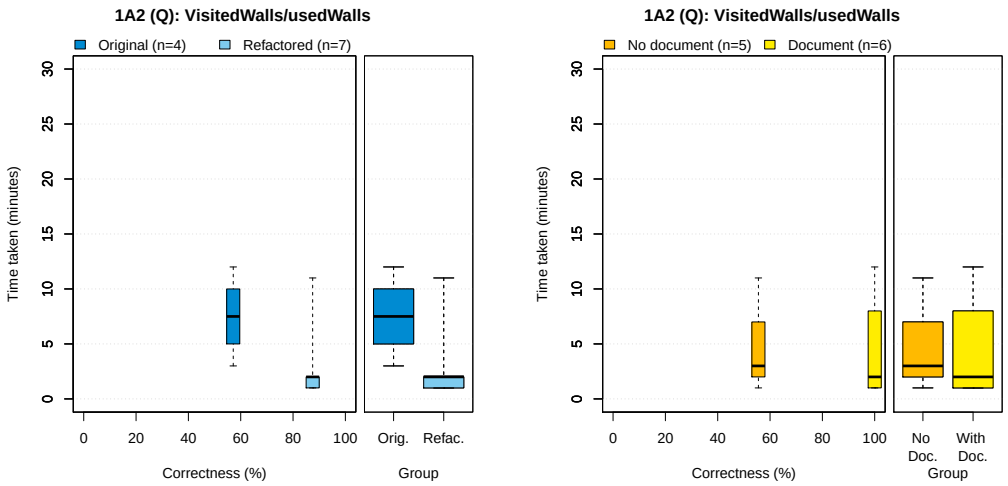
`UpdateUnreachableHelper`/`FindContourStartingAt` methods have been made shorter and the naming was improved to be more clear about what is happening.

Figure 5.5 shows the timing and correctness data for this assignment[6]. In Figure 5.5a the data for the participants working in the original code is compared to the participants working in the refactored code[7]. It shows a large difference in both correctness and time taken. 63% of the participants working in the refactored code answered correctly, as opposed to 29% of the participants working in the original code. Those working in the refactoring version also finished quicker, with a median time of 19 minutes compared to 29 minutes. However, because of the very small sample sizes one cannot make strong conclusions from the data alone. In particular, the difference in distribution in completion time between the two groups can not be considered statistically significant (MWW test, $W = 10, n_1 = 2, n_2 = 5, p = 0.0786 > 0.05$). However, despite the lack of statistical significance of the completion times, the data is a strong indication that the refactoring improved the understandability.

In Figure 5.5b the data for the participants who have read the document is compared to the participants who have not. It shows similar results as Figure 5.5a, with the group with document performing better than the group without document: 67% of the participants who have read the document beforehand answered correctly in a median time of 18 minutes, compared to 33% in a median time of 28 minutes in the other group. However, the effect may be caused by the refactoring, rather than the document. This is due to the unbalance between the groups: the two participants with document who answered incorrectly both worked in the original code, while the other four participants with document all worked in the refactored code, meaning the completion times regarded for the group with document are

[6] Because of the small amount of data, the whiskers of the boxplots shown in this thesis always extend to the minimum and maximum values.

[7] In order to always be able to compare the completion time boxplots, even when the correctness is far apart or very close together (causing overlap), the boxplots are duplicated on the right of the graphic.



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.6: Correctness and completion times for correct solutions for assignment 1A2.

all from participants that worked in the refactored code. The MWW test indicates that the difference in distribution in completion times is not statistically significant ($W = 10, n_1 = 3, n_2 = 4, p = 0.2118 > 0.05$).

With a correctness percentage of about 47%, this assignment had the lowest correctness of all assignments. The intended solution was to place a check in the complicated (and heavily refactored) `UpdateUnreachableHelper`/`FindContourStartingAt` method. However, only 3 out of 15 participants did so. 4 participants placed the check in another location (namely the constructor of the `Area` class), which was also considered correct.

Most of the incorrect solutions (6 out of 8) had implemented the check in the wrong place in the `GetAreas`/`CreateAreas` method. This indicates that the code might be not entirely clear, but may also indicate that the assignment text was unclear or perhaps ambiguous, which might invalidate the results. However, conversations after the experiment with some participants who had answered incorrectly reveal that they made incorrect assumptions about the functionality of the method. Additionally, the large difference in correctness between the participants working in the original code and those working in the refactored code indicate that at least part of the difficulty lied in the code and could be 'solved' by refactoring. Therefore, the results are deemed valid and are thus included in this analysis.

Assignment 2

The assignment text read:

> What does the parameter `usedWalls` in the `UpdateUnreachableHelper` method/field `VisitedWalls` in the `AreaCalculator` class contain?

This multiple-choice question intended to have the participants dive into the complex UpdateUnreachableHelper/FindContourStartingAt method in order to understand it. The ex-

pectation was that the participants working in the original code would have a harder time in understanding the code. The original method had a large number of parameters (6), so tracking one specific parameter is likely to be more difficult. In addition, the functionality is grouped in its own class in the refactored version and is generally closer together, likely making the 'following' of the execution flow easier. A factor that might make the refactored version harder to understand, is the fact that the method parameter has been refactored to a class field, increasing its scope and with it, the amount of locations it can be used and modified.

Figure 5.6a compares the data of the groups working in the original code and those working with the refactored code. Again, the participants with the refactored code answered correctly more often: 88%, versus 57%. Additionally, the participants were faster: A median time of 2 minutes, versus 7 minutes, a statistically significant difference (MWW test, $W = 25, n_1 = 4, n_2 = 7, p = 0.0432 < 0.05$).

Figure 5.6b compares the data for the groups working with and without document. A big difference in correctness can be seen: all participants who have read the document answered correctly, while 44% of participants without document answered incorrectly. The timing data is very similar with a median of 2 minutes for the group with document versus a median of 3 minutes for the other group, and the difference between the two can not be considered statistically significant (MWW test, $W = 17, n_1 = 5, n_2 = 6, p = 0.7803 > 0.05$).

Overall

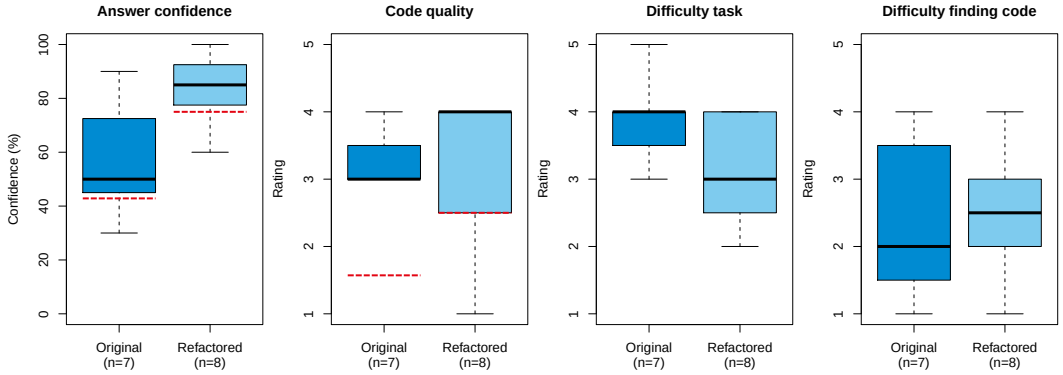
At the end of each exercise, the participants answer a set of questions about the exercise and the code they just worked in (see also Section 3.5.2). The results are shown in Figure 5.7.

Figure 5.7a compares the answers of the groups working in the original and refactored code. The participants working in the refactored code reported a higher answer confidence (median 85%) than the participants working in the original code (median 50%). This is also reflected in the actual correctness, shown as a red line in the first plot, with an average correctness percentage of 75% for the participants working in the refactored version, versus 43% for the participants working in the original version. The code quality was rated similarly, but the participants working in the original code more often thought that the code should be refactored, shown as a red line in the second plot[8], with 86% of the participants in the original code stating that the code needs refactoring, opposed to 63% of the participants in the refactored code. The task difficulty was also rated slightly lower by the participants working in the refactored code.

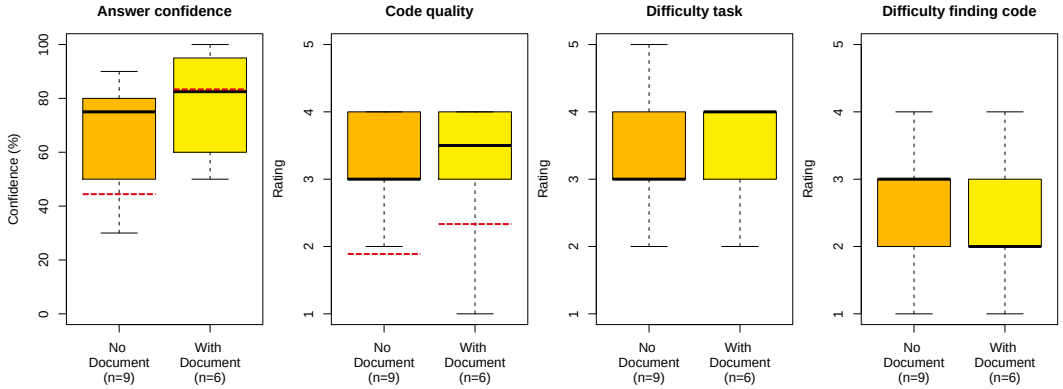
Figure 5.7b shows the data for the groups with and without document. In these, little difference can be seen between the two groups, with the exception of the correctness percentage. The correctness percentage is 44% for the group working without document, versus 83% for the group working with document[9]. However, as a response to the question "How much do you think the document you received before the experiment has helped you

[8]The percentage of participants stating that the code should be refactored is scaled to a rating by setting "Needs refactoring" equal to a rating of 1 and "Does not need refactoring" equal to a rating of 5. The red line is the mean of all participants in the group.

[9]Interestingly, this difference is not reflected in the confidence.



(a) Original versus Refactored code



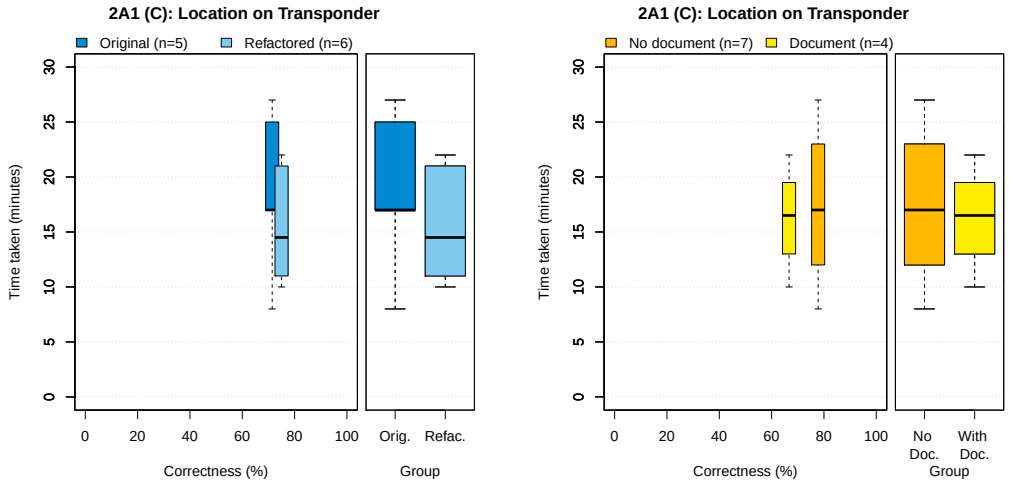
(b) No document versus Document

Figure 5.7: General data for exercise 1

in solving the above task(s)?", all participants with document[10] answered with either a 1 or 2 on a scale from 1 to 5, with a 1 indicating "Not at all". There is also no meaningful difference between the groups in terms of difficulty finding code, while the document is assumed to help participants find the relevant parts of code faster.

In general, the data supports the hypothesis that the refactoring has improved performance for this exercise. The assignment data also gives an indication that the document has improved performance for the participants, but this may also be explained by the fact that the unbalance in the groups, in particular the fact that the participants in the group with document are a lot more likely to also work in the refactored code. Unfortunately, there is not enough data to correct for this unbalance in a meaningful manner (see also Section 6.1.2). Therefore, we should conclude that the data also supports the hypothesis that the document has improved performance for this exercise.

[10]The question was only asked to participants who had received the document.



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.8: Correctness and completion times for correct solutions for assignment 2A1.

5.2.3 Exercise 2

The second exercise focuses around the `IsValidLocation` method, which was moved to its own `LocationValidator` class in the refactored version (see also Section 4.3.2). The introductory text explicitly mentions the context of the `MarkerLocations` method as using the `IsValidLocation` method to check which locations are valid. While for all the assignments, only the `IsValidLocation` method was needed to inspect, most participants started in the `MarkerLocations` method as it was mentioned in the text. The introduction text read:

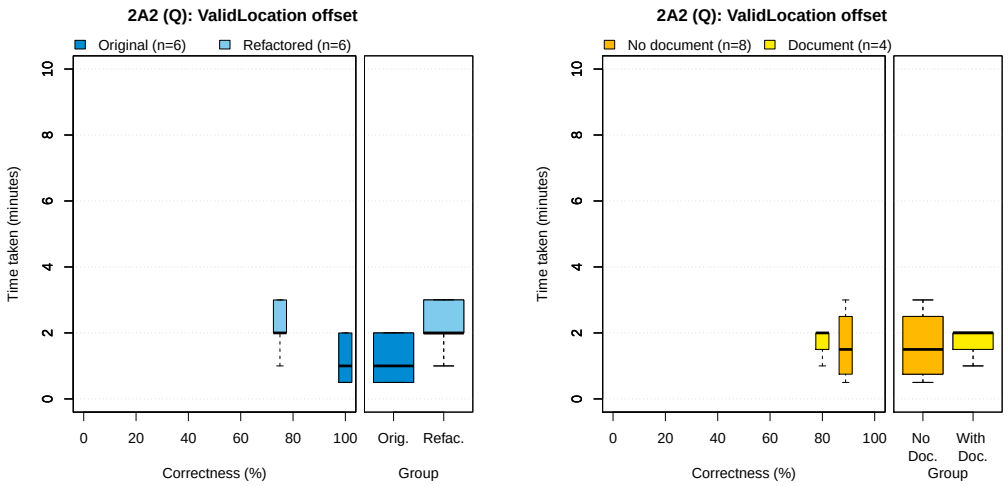
> The `MarkerLocations` method in the `BarnLayoutHelper`/ `MarkerLocationCalculator` generates the waypoints shown in the layout. The user can use these waypoints to draw the route for the Robots. The algorithm uses the `IsValidLocation` method in the `BarnLayoutHelper`/ `LocationValidator` class to make sure the waypoints it generates are valid, given the layout and the width of the scraper of the robot.

Assignment 1

The assignment text read:

> The customer has decided that locations lying on the same location as a `Transponder` cause confusion. Ensure locations lying on the same location as one of the `Transponders` in the `BarnLayout` are not considered valid.

This assignment had the participants add a new condition for checking valid locations. Its purpose was to see if adding a new condition would be easier in the refactored version,



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.9: Correctness and completion times for correct solutions for assignment 2A2.

where the conditions are split into separate methods. The expectation was that there would be little difference between the original and refactored code, as in both cases the purpose of the method should be clear relatively quickly, and the individual conditions do not need to be understood in order to add the condition.

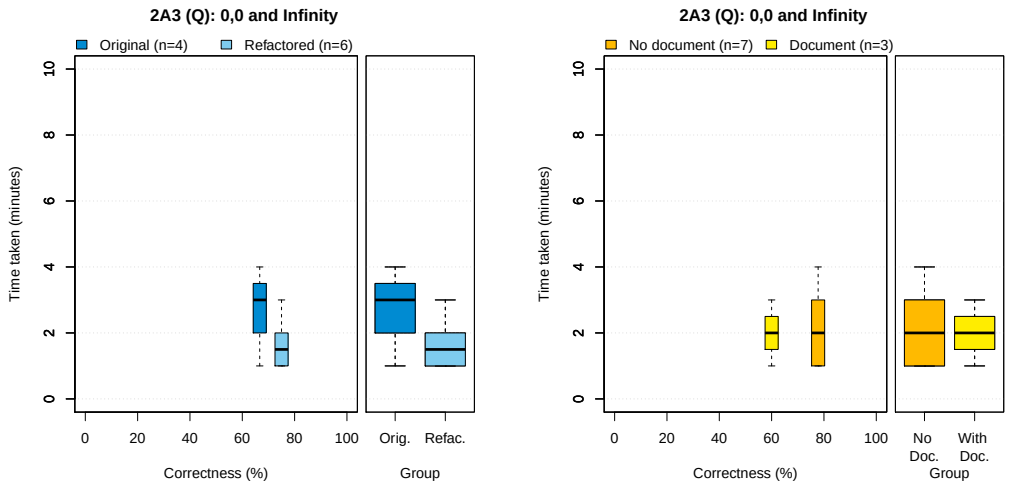
Figure 5.8 shows the timing and correctness data for this assignment. As expected, there is little difference in completion times and correctness when comparing both the groups working in original or refactored code (shown in Figure 5.8a) and the groups with and without provided document (shown in Figure 5.8b). The differences in completion times can not be considered statistically significant for both refactoring (MWW test, $W = 20, n_1 = 5, n_2 = 6, p = 0.4102 > 0.05$) and document (MWW test, $W = 15.5, n_1 = 7, n_2 = 4, p = 0.8498 > 0.05$) treatments.

An interesting observation that arises from the submitted solutions is that most participants (8 out of 11 correct solutions) added the new condition in the same manner as the rest of the method, meaning that participants in the original code mostly added an extra statement (mostly with comment), while the other half mostly made an extra method. This suggests that people are flexible about code style and will generally try to copy the existing code style. This is supported by the 'Broken window theory'[68], meaning that people working in unclean code generally will produce less clean solutions.

Assignment 2

The assignment text read:

> What does the `offset` parameter in the `IsValidLocation` method/`_offset` field in the `LocationValidator` class contain?



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.10: Correctness and completion times for correct solutions for assignment 2A3.

This multiple choice question asks about the function of a single variable in the `IsValidLocation` functionality. Its goal is to have the users understand the usage of the parameter/field, which might be easier in the refactored version due to the use of descriptive method names.

The timing and completion data for the groups working in the original and refactored code, shown in Figure 5.9a, shows that the participants working in the refactored code were generally slower (a median time of 2 minutes versus 1 minute) and answered correctly less often (75% versus 100%). The difference in time is not statistically significant, but close (MWW test, $W = 6, n_1 = 6, n_2 = 6, p = 0.0534 > 0.05$). However, because the assignment was very short (the longest time a participant spend on this assignment was 3 minutes), the times are likely to be affected by the low granularity of the timing data, and the effect size is very low.

Due to an error made during refactoring, the refactored version did not have any XML documentation comment for the `layout` and `offset` parameters of the constructor, which were present in the XML documentation comment in the unrefactored version (as parameters of `IsValidLocation` method). This means the comments of the refactored version contained less information. The documentation for the `offset` parameter was "The minimum distance from any wall", which could alone be used to answer the question with reasonable confidence. This possibly contributed to the decreased performance of the participants working in the refactored code.

When comparing the data for the groups with and without document, shown in Figure 5.9b, little difference exists in both correctness and timing data. The differences in times is not statistically significant (MWW test, $W = 14, n_1 = 8, n_2 = 4, p = 0.7893 > 0.05$).

Assignment 3

The assignment text read:

> Can a point at 0,0 be considered valid in any layout? How about a point at `Infinity`,`Infinity`?

The question was written so participants had to go through the flow of the method, examining the cases. The expectation was that the participants would be slightly faster in the refactored version, due to the method names making the code easier to follow. However, the code is also more spread out in the refactored version, which might cause the participants to perform slower.

Figure 5.10a shows the timing and correctness data comparing the groups working in the original and those working in the refactored code. The difference is again small, with the participants working in the refactored code finishing slightly faster, with a median time of 1.5 minutes, compared to 3 minutes for those working in the original code. This difference is not statistically significant (MWW test, $W = 18.5, n_1 = 4, n_2 = 6, p = 0.1797 > 0.05$).

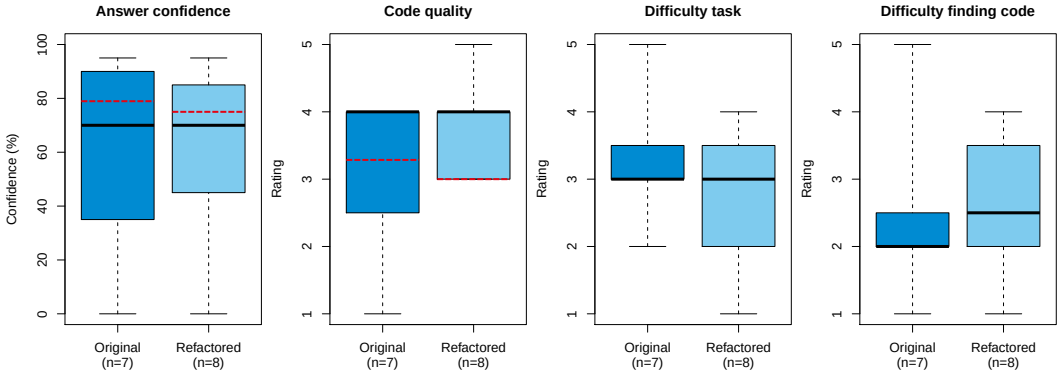
In Figure 5.10b the data for the groups with and without document is compared. The participants working without document were slightly more likely to answer correctly (78% versus 60%), but this difference is small. Both groups also had equal median completion time of 2 minutes, and the difference in distributions can not be considered statistically significant (MWW test, $W = 11, n_1 = 7, n_2 = 3, p = 1 > 0.05$).

Overall

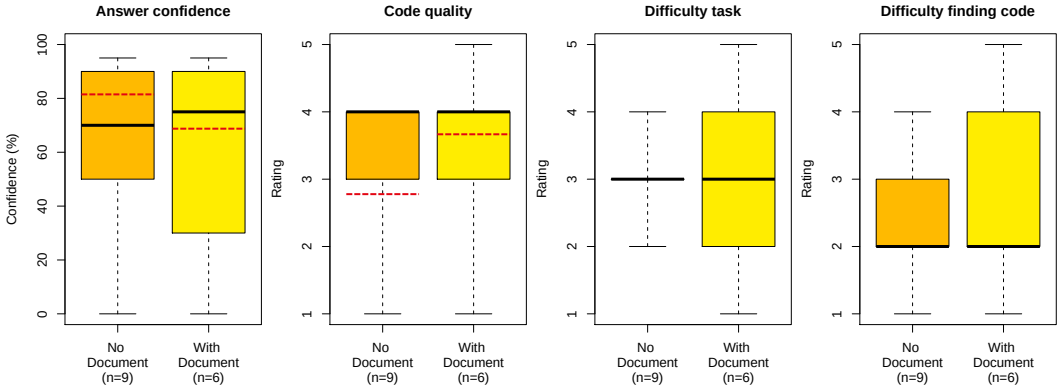
Figure 5.11a shows the answers to the questions after the exercises, comparing the groups working in original and refactored code. It shows little to no difference between the two. The participants working in refactored code rated the task as slightly less difficult (although both with a median of 3 out of 5). Additionally, they indicated they had slightly more difficulty finding the code than those working in the original code. This is interesting, as the expectation was that the separation of the functionality into multiple classes would make relevant code easier to find. No clear reason for this can be found from the data. One participant stated that he found the assignments more difficult because "the logic is divided over 2 classes", indicating that the participants indeed spent quite some time in the `MarkerLocations` method, which was left unrefactored (see also Section 4.3.1).

Figure 5.11b shows the data for the groups working with and without document. It shows little difference between the groups on all questions. The participants who had received the document also did not think it helped them during the experiment, with all but one participant stating that the document "not at all" helped them in solving the tasks. The one participant that answered with a 4 on a scale of 1 to 5 (indicating that the document helped him) indicated that he used the document during the experiment, to review the functionality of placing the markers. However, he was not able to solve the first two assignments correctly.

In general, the data for this exercise does not support the hypothesis that refactoring helped the performance for these assignments. Likewise, there is no support for the hypothesis that the document helped the performance.



(a) Original versus Refactored code



(b) No document versus Document

Figure 5.11: General data for exercise 2

5.2.4 Exercise 3

Exercise 3 focused around the `Contains` functionality in the `Area` class (see also Section 4.4). The introductory text was very concise, only mentioning the `Area` class, as the assignment texts provide enough contextual information for the participants. The introduction text read:

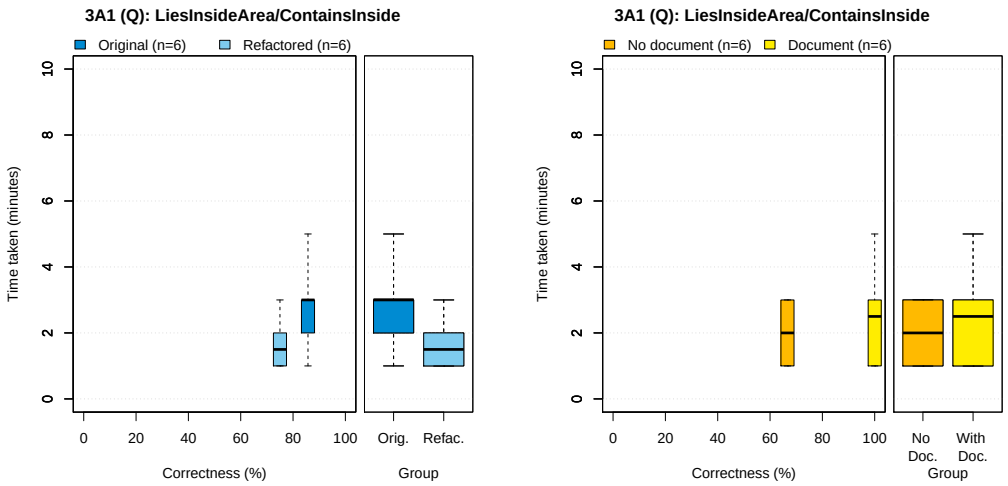
> Enclosed areas are represented by the `Area` class.

Assignment 1

The assignment text read:

> What does the method `LiesInsideArea`/`ContainsInside` in the `Area` class do?

This question about the LiesInsideArea/ContainsInside method was included to see whether the method was made more understandable by the refactoring. In particular, the refactoring



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.12: Correctness and completion times for correct solutions for assignment 3A1.

aimed to make the method name clearer and replaced a complex conditional with a method call describing the conditional. It was expected that the participants would perform faster in the refactored version.

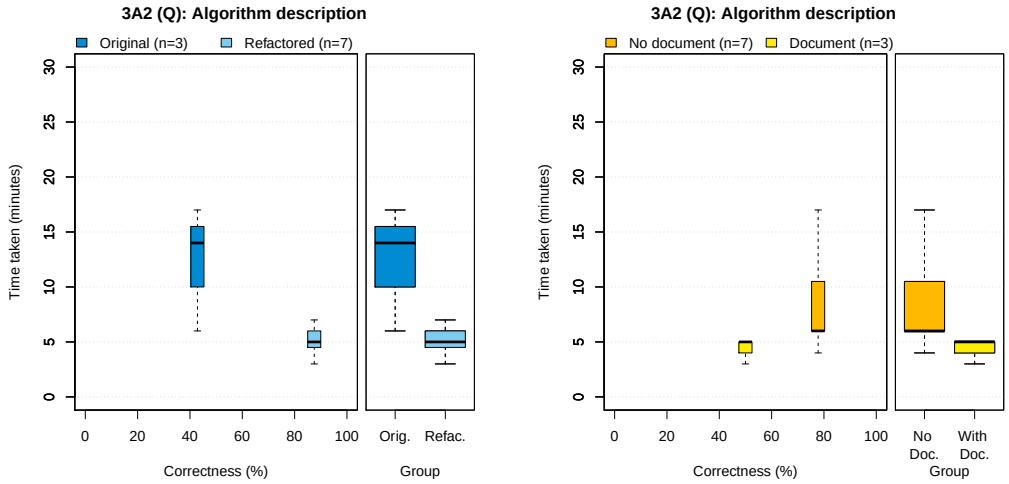
Figure 5.12a shows the completion times and correctness for the participants working in the original code and those working in the refactored version. It shows that the participants were slightly faster in the refactored version, finishing in a median time of 1.5 minutes, compared to 3 minutes for those working in the refactored version. The participants working with the original code were slightly more often correct, with 86% answering correctly, as opposed to 75% of the participants in the refactored code. The difference in timing data cannot be considered statistically significant (MWW test, $W = 28, n_1 = 6, n_2 = 6, p = 0.112 > 0.05$).

Figure 5.12b shows the data for the groups working with and without document. The timing data is similar, with the participants without document solving the assignment in a median time of 2 minutes, compared to 2.5 minutes for the participants who had received the document before the experiment. This difference can also not be considered statistically significant (MWW test, $W = 15, n_1 = 6, n_2 = 6, p = 0.6758 > 0.05$). The participants with the document were more likely to answer correctly, with 100% of them answering correctly, opposed to 67% of the participants working without document.

Assignment 2

The assignment text read:

> What is the most accurate description of the algorithm the `ContainsWithExtreme`/`Contains` method uses to determine whether the `Area` contains a point?



(a) Original versus Refactored code

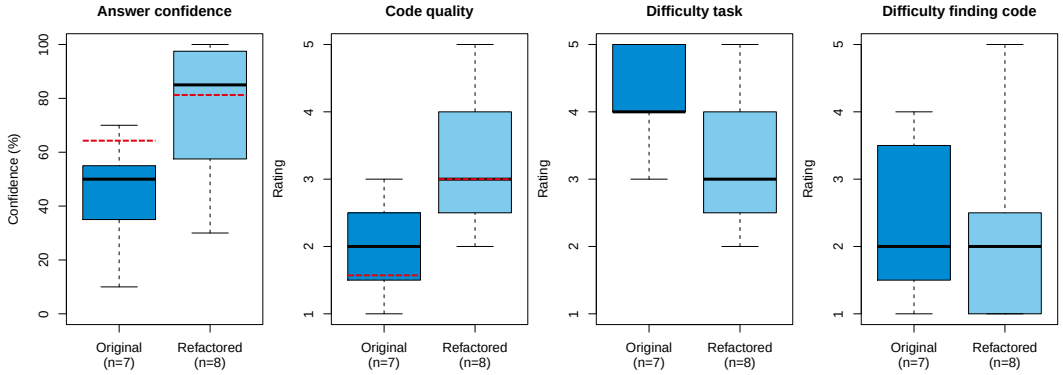
(b) No document versus Document

Figure 5.13: Correctness and completion times for correct solutions for assignment 3A2.

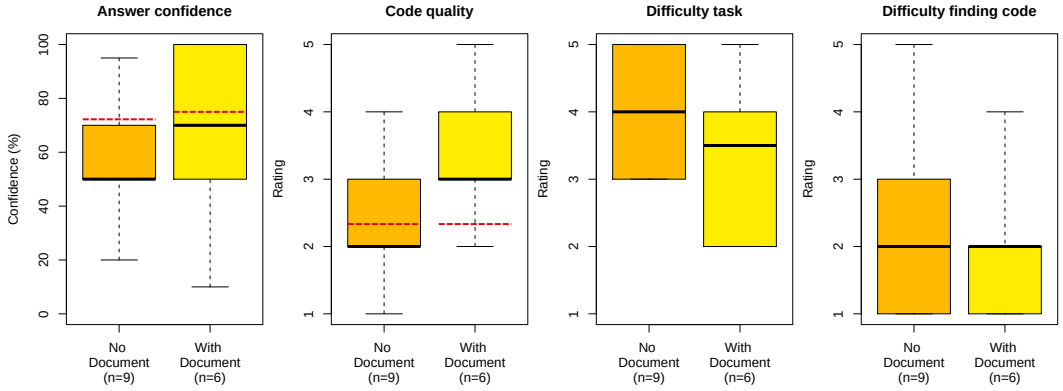
This question is about core algorithmic functionality of the `Contains` method, and requires the participants to dive into the relatively complex method. The refactored version is expected to be more understandable, due to the rigorous refactoring splitting up the large method containing the algorithm and using clearer method names. In the multiple choice options, care was taken not to use terms specific to the algorithm, or terms that were introduced in the refactored version (such as 'ray').

Figure 5.13a shows the correctness and timing data for the groups working in the original and refactored code. A large difference can be seen in both completion times and correctness. The participants working in the refactored code finished in a median time of 4.5 minutes, much faster than the 14 minutes median of the participants working in the original code. Additionally, 88% of the participants with refactored code answered correctly, versus 43% of the participants with the original code. However, due to the small number of correct solutions in the original version, the difference in completion times can not be considered statistically significant (MWW test, $W = 19, n_1 = 3, n_2 = 7, p = 0.0641 > 0.05$).

When comparing the groups working with and without document, as shown in Figure 5.13b, the group with document mostly finished faster, although the median times are close together (5 minutes versus 6 minutes). On the other hand, the group without document answered correctly more often, with 78% versus 50%. The difference in completion times is also not statistically significant (MWW test, $W = 19, n_1 = 7, n_2 = 3, p = 0.0641$), again likely due to the small number of correct responses in the group with document. It is also unclear why the participants with document finished faster, and this effect may again be caused by the refactoring, as all participants who have read the document and answered correctly working in the refactored version of the code.



(a) Original versus Refactored code



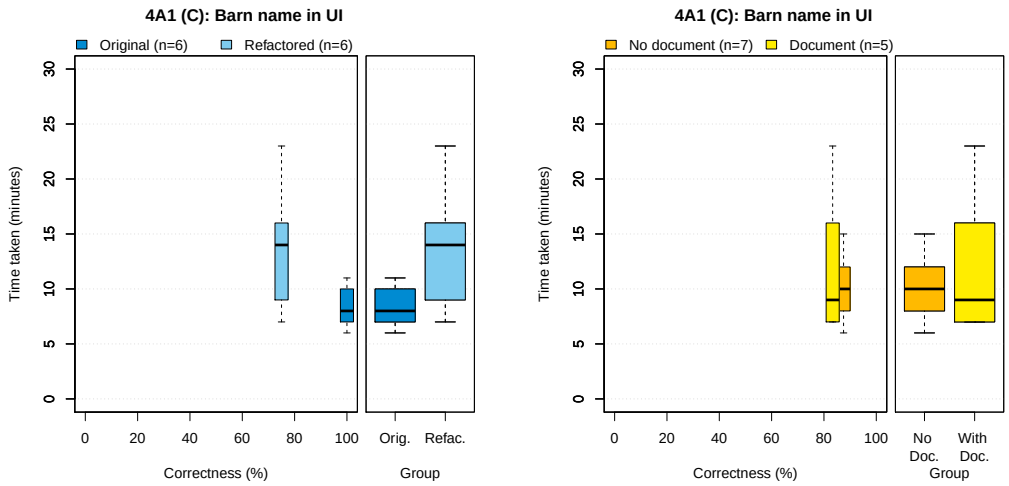
(b) No document versus Document

Figure 5.14: General data for exercise 3

Overall

In Figure 5.14a the responses to the questions after the assignments are shown, comparing the groups working in the original and the refactored code. It heavily indicates increased understandability in the group working in the refactored code, with higher answer confidence (median 85% versus 50%) and correctness (64% versus 81%), higher rating for code quality (median 3 out of 5 versus 2) and lower perceived task difficulty (median 3 out of 5 versus 4).

This is also reflected in the comments made by the participants, with the participants in the original code finding the `ContainsWithExtreme` method hard to understand and unreadable, while the participants in the refactored code were positive about the naming and conciseness of the `RayCrossesAreaUnevenAmountOfTimes` method and the methods it calls. This does not mean they were universally positive about the functionality, citing the wish for more documentation or comments and general unclarity of the functionality. One participant explicitly disliked the functionality being split up into multiple, smaller methods, stating it was “hard to keep track of things” due to the “large call stack”.



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.15: Correctness and completion times for correct solutions for assignment 4A1.

Figure 5.14b shows the responses for the groups with and without document. It shows the participants working with the document were slightly more positive about the code quality and the task difficulty, but these differences are small. Additionally, the participants who had received the document did not believe it helped them during this exercise, all rating the extent to which it helped a 1 or 2 out of 5.

In general, the data supports the hypothesis that the refactoring has improved performance, especially the data from assignment 2 and the responses from the participants after the exercise. The hypothesis that the document improved performance is not supported, with small or no effects observed in both assignment data and the responses.

5.2.5 Exercise 4

Exercise 4 focuses on functionality present in the UI, meaning things the participant can actually see when running the program. In the unrefactored version, all relevant functionality is in the `MainViewModel` class, while in the refactored version some relevant functionality is found in another class (see also Section 4.5). This means that in the unrefactored version the functionality is more easily found during exploration with scrolling.

There was no introductory text. This means the `MainViewModel` was not explicitly mentioned, and for the first time in the experiment the participants have to do their exploration relatively unguided.

Assignment 1

The assignment text read:

> In the UI, `Barns` are described using the amount of areas it contains. Add to this so it also shows the number of robots that are associated with the `Barn`.

The assignment hints at a description in the UI, describing a `Barn`. Participants were expected to either look into the UI and use free-text search to locate the text, or perhaps use the `Barn` as a starting point. Knowledge of WPF, the GUI framework used in the project, was explicitly not required to solve the assignment. The purpose of the assignment was to see if the refactored `MainViewModel`, with its responsibilities more divided over several classes, made it easier to look for specific functionality.

In Figure 5.15a the timing and correctness data is shown, comparing the group working in the original and those working in the refactored code. The participants in the original code finished faster, completing the assignment in a median time of 8 minutes versus the median of 14 minutes for the group working in refactored code. Additionally, the correctness was higher, with 100% of the participants working in the original code giving a correct answer, opposed to 75% of the participants with the refactored code. The difference in time can not be considered statistically significant (MWW test, $W = 6.5, n_1 = 6, n_2 = 6, p = 0.0756 > 0.05$).

Figure 5.15b shows the data for the groups with and without the document. Here, little difference exists between the two. The median completion time for the group without document is slightly higher, with 10 minutes versus 9. Also the correctness is slightly higher, with 88% versus 83%. Both effects are very small however, and thus no clear difference between the two groups can be concluded. The differences in timing data is also not statistically significant (MWW test, $W = 15.5, n_1 = 7, n_2 = 5, p = 0.8059 > 0.05$).

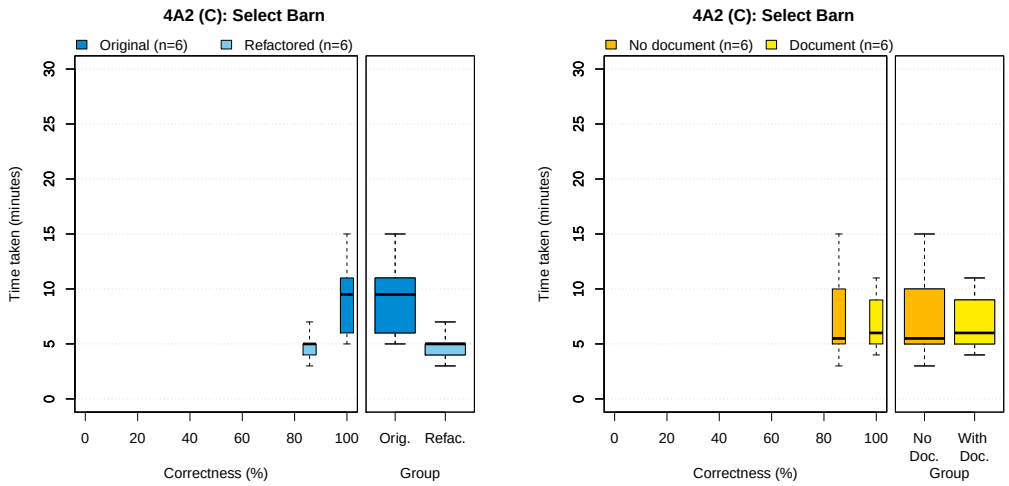
The recordings do not show a clear reason for the difference in performance found in the groups with and without refactoring. Most of the time spent by the participants was to find the relevant code that needed to be changed, because the assignment did not explicitly mention a starting point. When the participants found the code, they mostly needed only 3 minutes for the implementation and verification. Difference in the performance would have been explained if the participants more often looked for the functionality in the `MainViewModel` class than in the `Barn` class. However, almost all participants found the relevant code by doing a free text search over the entire project using part of the text they could find in the UI. As such, solution time was likely mostly affected by how quickly the participant thought of using that search strategy, instead of whether the code was refactored or not.

Assignment 2

The assignment text read:

> After successfully importing a file, the newly imported barn is not shown automatically. Fix this.

This assignment contains the only deliberate code change made for the purpose of the experiment, as the old code already had this functionality. The change in the refactored code amounted to removing a single method call. In the unrefactored code, multiple lines were deleted. This difference is due to the reduction in duplication in the refactored version. It is therefore expected that the participants working in the refactored version will be faster.



(a) Original versus Refactored code

(b) No document versus Document

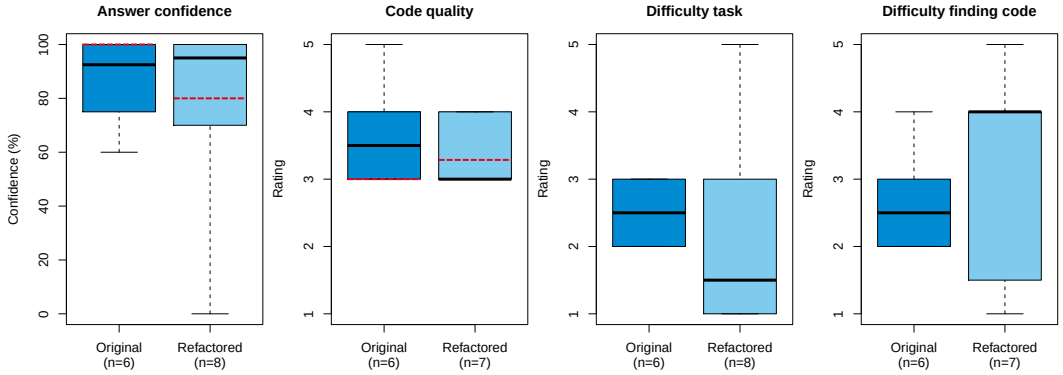
Figure 5.16: Correctness and completion times for correct solutions for assignment 4A2.

It could be said that, while the assignment was not designed that way, the assignment favors the refactored version substantially, due to the changes needed in the refactored version being significantly less than in the original version. However it can also be considered fair, as refactoring generally aims to improve the maintainability of the code and therefore increase the amount of changes that can be made easier. It is therefore likely that in regular maintenance one will encounter change tasks that will be easier in refactored code.

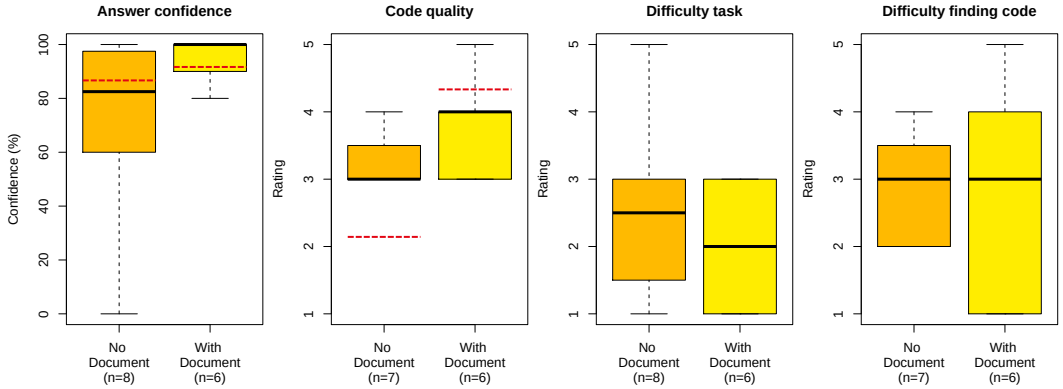
Figure 5.16a shows the completion time and correctness data for the group working in the original code, compared to the group working in the refactored code. It shows that the participants in the refactored code finished faster, with a median time of 5 minutes, compared to the median time of 9.5 minutes for the participants in the original code. They were slightly less often correct, with 85% of the participants in the refactored code answering correctly, compared to 100% of the participants in the original code. The difference in timing is statistically significant (MWW test, $W = 32.5, n_1 = 6, n_2 = 6, p = 0.0225 < 0.05$).

Another interesting observation lies in the submitted solutions. While all participants working in the refactored version simply used the existing method as expected, the solving methods of the participants in the original code differed greatly. The solution of only one participant came close to the code before the change made for the experiment, having copied the relevant lines from another method. All other solutions re-used the existing methods, or refactored the existing functionality to make re-use easier. This indicates that developers are generally wary to introduce duplication, and rather spend extra time to avoid it.

In Figure 5.16b the data for the participants who had received the document is compared to those who have not. Little difference can be seen, with the participants without document finishing in a median time of 5.5 minutes, compared to a median of 6 minutes for those with document. The correctness data is also close, with 85% compared to 100%. The difference in completion times is not statistically significant (MWW test, $W = 18, n_1 = 6, n_2 = 6, p = 1 > 0.05$).



(a) Original versus Refactored code



(b) No document versus Document

Figure 5.17: General data for exercise 4

Overall

In Figure 5.17a the data from the responses to the questions at the end of the exercise is shown, comparing the participants working with the original code to those working in the refactored code. It shows little difference in answer confidence and perceived code quality by the participants. The participants with the refactored code found the task to be a little less difficult. They also reported higher difficulty finding the code they needed (although the answers in both cases varied wildly from 1 out of 5 to 5 out of 5). This is an indication that the splitting of the functionality might have made the relevant functionality harder to find. Comments made by the participants reveal that while some participants in the original code felt that the `MainViewModel` class was too big and should be split up, some participants in the refactored code disliked the placement of some functionality in the `Barn` class (which was moved there by the refactoring), indicating the refactoring could have been better.

The data from the participants with and without document is shown in Figure 5.17b. The participants who had received the document rated the code quality slightly higher and were a lot more likely to believe that the code should be refactored, regardless of whether

they were working in the original or the refactored code. This difference is quite large, with only 17% of the participants working with the document believing the code needs to be refactored, opposed to 71% of the participants without document. It is not clear why this is the case. A possible explanation is that the code was as they expected it to be from the information in the document, and thus were less likely to believe it needed changing. The participants with the document were also relatively positive about the extent to which the document helped during this exercise, with two participants answering with 4 out of 5, indicating that the document helped them solving the tasks. On the other hand, the other participants responded with a rating of 1 out of 5, indicating that they did not believe the document helped them.

Across all participants, answer confidence is in general higher, compared to other assignments. This is likely due to the changes being easily verifiable by starting the application and checking if the functionality works as expected in the interface, making the participants more confident that their changes are actually functionally correct.

In general, the data does not support the hypothesis that the refactored has improved participant performance for this exercise, with mixed results from both assignments. Likewise, the data also does not support the hypothesis that the document has improved participants performance, with very little differences between the groups with and without document.

5.2.6 Exercise 5

This exercise focuses on the `ManureRobot` class (see also Section 4.6). The introductory text is again very short, pointing to the `ManureRobot` class and its interface, along with indicating that there are no other implementations the participants need to take into account.

> Robots in the application are represented by the `IRobot` interface. Currently, the only implementation is `ManureRobot`.

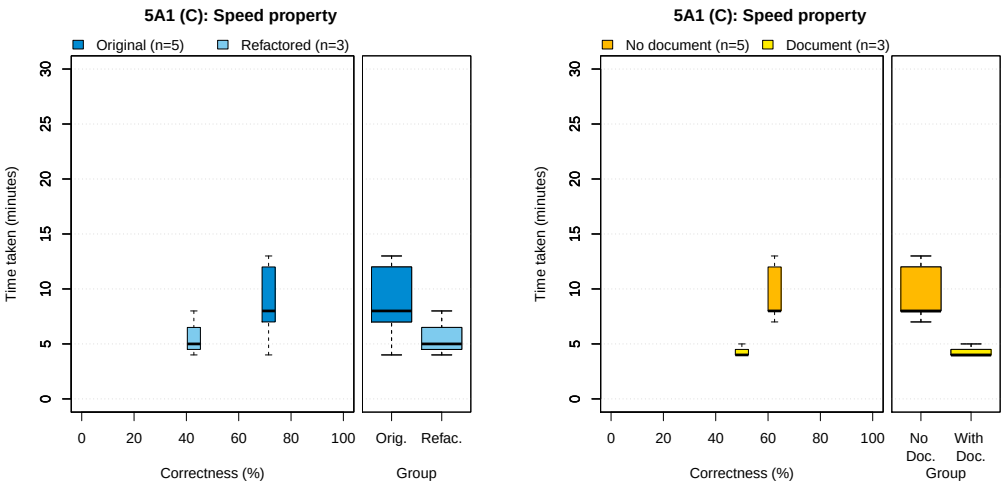
Assignment 1

The assignment text read:

> Add to the `IRobot` interface and `ManureRobot` class so they also have a speed property. This can be represented by a double. The property should be able to be set in the same manner as the other properties.

The goal of this assignment is to measure if the large amount of parameters in the constructor in the original version will impact the understandability, as this change requires a parameter to be added to the constructor in the original version. However, the effect might not be large, as adding a parameter is likely a simple task.

Figure 5.18a shows the completion times and correctness for the groups working in the original and the refactored code. The participants working in the refactored version finished faster, with a median time of 5 minutes compared to 8 minutes for the participants with the original code. However, they were also correct less often, with only 43% giving a correct



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.18: Correctness and completion times for correct solutions for assignment 5A1.

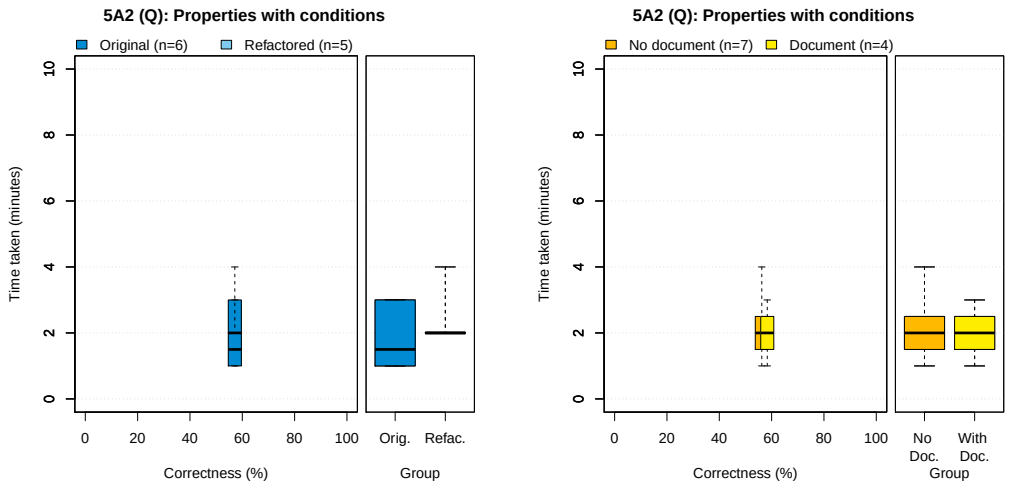
answer, compared to 71% of the other participants. The difference in completion time is not statistically significant (MWW test, $W = 11, n_1 = 5, n_2 = 3, p = 0.3653 > 0.05$).

Figure 5.18b shows the same data for the groups with and without document. The participants who had received the document beforehand were a lot faster than those who have not, finishing in a median time of 4 minutes, compared to 8 minutes for the other participants. However, they were less often correct in their submissions, with a correctness percentage of 50%, compared to 62.5% for the participants who did not read the document beforehand. The difference in completion times is statistically significant (MWW test, $W = 15, n_1 = 5, n_2 = 3, p = 0.0347 < 0.05$). It is unclear why the participants working with the document were faster: the document did not mention this functionality and the recordings do not reveal anything that could have improved performance.

As discussed in Section 4.6, the `ManureRobot` class uses the Builder pattern to construct instances. This means that every new parameter also needs to be added to the `Builder` class, otherwise it will never be set. In most of the incorrect solutions, the participants added the property to the interface and the class, but forgot the relevant method in the Builder. The recordings reveal that participants did not see the Builder class being present and also missed other clues that the properties were set in the Builder, such as the private constructor/ private setters and object construction in the tests. This is also evidenced by the completion times of the incorrect solutions: the participants with a incorrect solution took a maximum of 6 minutes (median 3), while participants who submitted a correct solution mostly took more time (median 7.5 minutes), indicating that the participants stopped too soon.

Assignment 2

The assignment text read:



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.19: Correctness and completion times for correct solutions for assignment 5A2.

> What properties of `ManureRobot` have extra conditions enforced on their values by the `Builder`? You may select more than one answer.

This assignment attempts to check whether the explicit `Validate` method in the refactored version was more understandable in comparison to the conditions spread out over the class as in the original version. It also explicitly mentions the `Builder` class, meaning the participants could not miss it.

The answer options were all properties of the `ManureRobot` class, and the participants could select multiple options. The properties, and as a result the answers from the participants, can be grouped into several groups:

- `Name`, `Model` and `MachineId` have a default set to ”Unknown” if the string was null at the time of building. This is not a condition.
- `PaddingFromWall`, `ScraperWidth` and `Length` are checked if they had a minimum value of 0 or 1, throwing an exception if they did not. This is a condition.
- `ScraperWidth`, `Length` and `Width` have a default value of `NaN`. This is not a condition.
- `HomeTransponder` and `Power` can not be null. In the unrefactored version, this was done in the constructor of the Builder. In the refactored version, this functionality was moved to the `Validate` method. It is expected that it would be clearer in the refactored version that these conditions existed. This is a condition.
- `Sprayers` does not have a condition enforced on it, but the `WithSprayer` method in the `Builder` did have a `ContractAnnotation`[11] with attribute ‘`null => halt`’.

[11] `https://www.jetbrains.com/help/resharper/Contract_Annotations.html`

> This is a non-functional annotation that the ReSharper plugin uses as a hint for the code highlighter that it can expect the method to fail whenever null is passed into it[12]. However, the participants might be confused about its presence and may guess that it does have a functional effect of being a condition. This is not a condition.

All answers from participants selected one or more of these groups, and only these groups (e.g., no participant selected `HomeTransponder` without also selecting `Power`). The question can thus be regarded as having 5 options, with 2 correct options and 3 incorrect options.

Figure 5.19 shows the completion time and correctness data for the participants. Because multiple correct answers were possible, some participants only got one of the two correct answers. These answers are regarded as 'half correct', so in those cases the participant is considered to have a correctness of 50%. Their times are included in the completion time analysis, as they have demonstrated at least a partial understanding.

In Figure 5.19a compares the completion times and correctness for the participants working in the original code with those working in the refactored code. Both groups at the same correctness percentage at 57%. The participants in the original code finished slightly faster with a median completion time of 1.5 minutes, compared to 2 minutes for those working in the refactored code. The difference in completion times is not statistically significant (MWW test, $W = 10, n_1 = 6, n_2 = 5, p = 0.3828 > 0.05$).

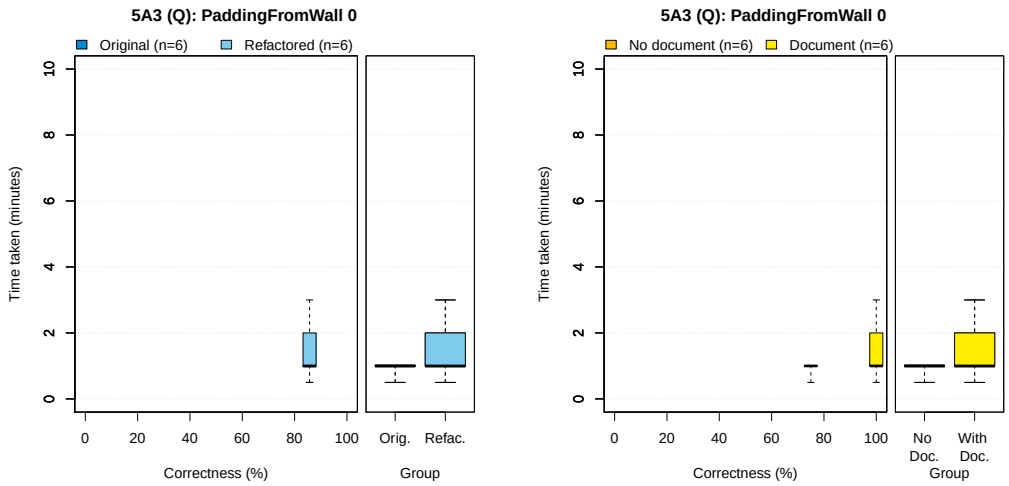
Some effect of the refactoring can be seen when the answers are studied more closely. The participants working in the original code who answered only half correct, all missed the `HomeTransponder` and `Power` group, while the participants working in the refactored code with only half correctness all missed the `PaddingFromWall`, `ScraperWidth` and `Length` group. The recordings reveal why this is the case: the participants working in the original code saw the ranges being checked in the `Build` method, but overlooked the conditions in the constructor of the `Builder`. This means they would likely have benefited from the `Validate` method in which all conditions would be in one place. The participants in the refactored code with half-correct answers however, missed the `Validate` method entirely and answered half correctly by luck, likely believing that a ReSharper annotation amounted to a condition. It is not clear why the participants in the refactored code missed the `Validate` method more often than participants in the original code missed the `Build` method, as they are in the same location, namely the end of the class.

The correctness and timing data for the participants working with and without document is compared in Figure 5.19b. The completion times are very similar with both groups having a median completion time of 2 minutes. The correctness percentage is also very similar: the participants working with document had a correctness of 58%, compared to 56% for the participants who had not received a document before the experiment. The difference in completion times is not statistically significant (MWW test, $W = 14.5, n_1 = 7, n_2 = 4, p = 1 > 0.05$).

Assignment 3

The assignment text read:

[12] Interestingly, the code does not actually do that, making the annotation actually incorrect as well.



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.20: Correctness and completion times for correct solutions for assignment 5A3.

Is it allowed to build a `ManureRobot` with a `PaddingFromWall` of 0?

This question was included to let the participants check the `Build`/`Validate` method and interpret the range checks. However, people who missed the method would have answered 'Yes' (the correct answer) as well, because they found no reason to believe that it would not be possible. This is an oversight not caught during the pilot experiments.

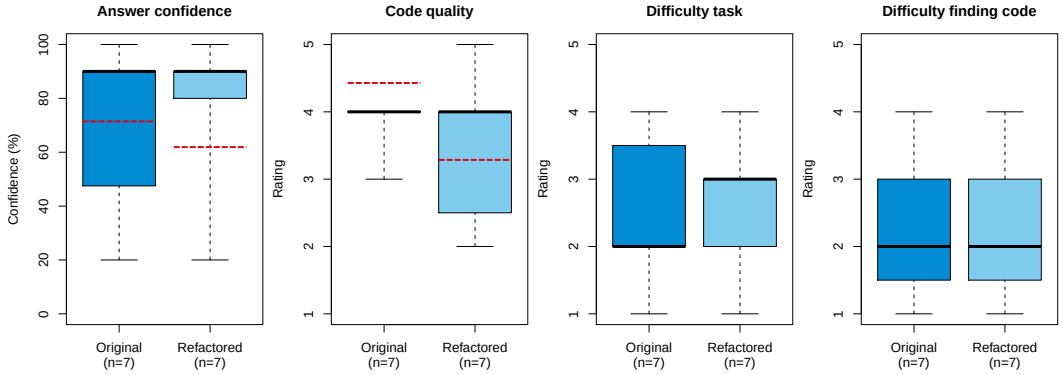
As a result, the results are unlikely to be meaningful. For completeness, they are shown in Figure 5.20. The participants working in the refactored version and those working in the original code had the same correctness percentage of 86% and the same median completion time of 1 minute, causing overlapping bars in Figure 5.20a. The difference in time between the two groups is not statistically significant (MWW test, $W = 13, n_1 = 6, n_2 = 6, p = 0.39 > 0.05$).

The participants working with the document had a correctness percentage of 100%, compared to 75% of the participants who had not received the document beforehand, as shown in Figure 5.20b. The median completion times were the same at 1 minute. The difference in completion times between the participants who had not received the document beforehand and those who did is not statistically significant (MWW test, $W = 13, n_1 = 6, n_2 = 6, p = 0.39 > 0.05$).

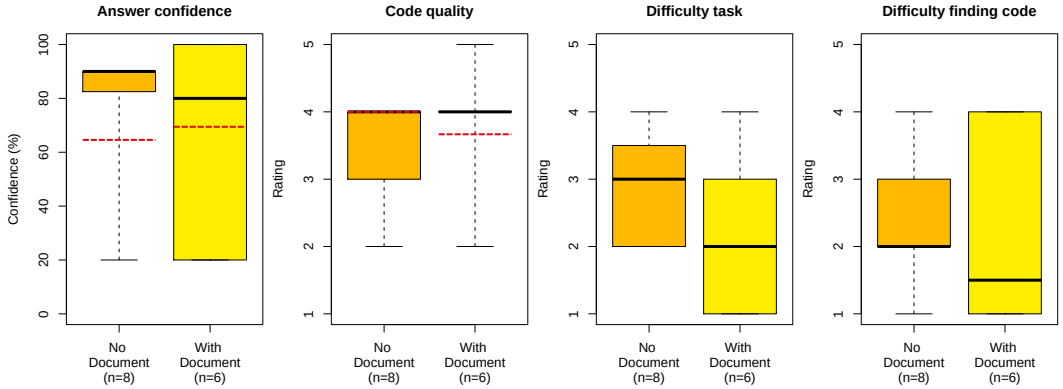
Despite the expectation that all participants would answer correct due to the oversight, two participants answered incorrectly. From their recordings it can be inferred that they did find the method with the range checks, but that they answered incorrectly for other reasons, likely misunderstanding the range checks as not inclusive.

Overall

Figure 5.21 shows the participant responses to the questions after the experiment. Comparing the groups working in the original and in the refactored code, as shown in Figure 5.21a,



(a) Original versus Refactored code

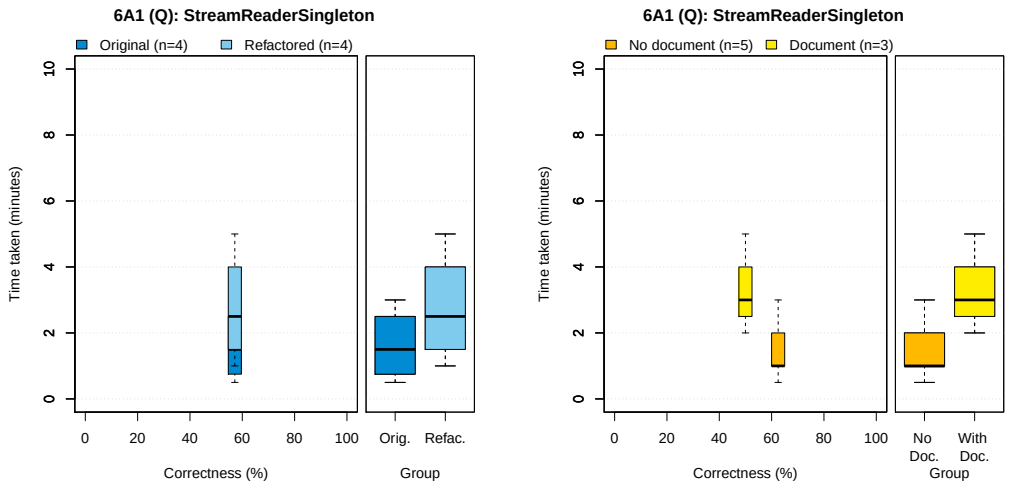


(b) No document versus Document

Figure 5.21: General data for exercise 5

we see little difference between the groups. The participants in the refactored code were a little less positive about the code quality and more often thought the code needed refactoring (43% versus 14%). However, the comments from the participants who rated the quality lower do not point to factors specific to the changes made in the refactoring. Instead, they disliked the use of the Builder pattern in general. In Figure 5.21b the groups with and without document are compared. It shows little difference between the groups. The participants also did not think the document helped them in solving the assignments, all rating the extent to which the document helped as a 1 or 2 on a scale of 1 to 5.

In general, the data does not support the hypothesis that the refactoring has improved performance, as the participants in the refactored code performed very similar to those working in the original code. Likewise, the data also does not support the hypothesis that the document improved performance.



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.22: Correctness and completion times for correct solutions for assignment 6A1.

5.2.7 Exercise 6

The last exercise contains a single assignment about the `StreamReaderSingleton` class, aiming to test the refactoring done to all Singleton classes (see also Section 4.7). The introductory text hinted that there are multiple Singleton classes, and their purpose.

> The code uses various Singleton classes that allow for easier replacement in unit testing.

Assignment 1

The assignment text read:

> What type is the actual class returned by `StreamReaderSingleton.Instance`?

The assignment was included to test if the refactorings made to the Singleton classes made the code more understandable. The expectation was that the refactored code would be *less* understandable than the original code. While the refactoring was justified by reducing duplication, the code can be regarded as being more complex after refactoring due to the introduction of generics.

The assignment can be considered confusing for the participants working in the original version. The original version contains a method that allows changing the instance in the Singleton from the concrete class (which is the expected answer) to another class that implements the interface defined as return type by the Singleton. While this method is not public and is only used in the test framework, the participants could still interpret the possibility as making the concrete class an incorrect answer. The refactored code does not have

this confusing, as the functionality of replacing the instance (required in the tests) is moved to the test framework and was not seen by the participants. The answers show that this possible confusion has likely not influenced the results: of the participants that indicated the interface as the returned class, half worked in the refactored code and half in the original code.

Figure 5.22a shows the completion times and correctness for the participants working in the original code and those working in the refactored code. It shows that the participants in the original code were slightly faster, with a median completion time of 1.5 minutes, compared to 2.5 minutes for the participants working in the refactored code. Both groups had the same correctness percentage of 57%. The difference in completion times is not statistically significant (MWW test, $W = 4.5, n_1 = 4, n_2 = 4, p = 0.3778 > 0.05$).

Figure 5.22b shows the same data for the groups working with and without the document. The group that received the document before the experiment finished slower, with a median completion time of 3 minutes compared to 1 minute for the participants working without document. They were also correct less often, with a correctness percentage of 50%, compared to 62% for the other group. The difference in timing is not statistically significant (MWW test,$W = 2, n_1 = 5, n_2 = 3, p = 0.129 > 0.05$).

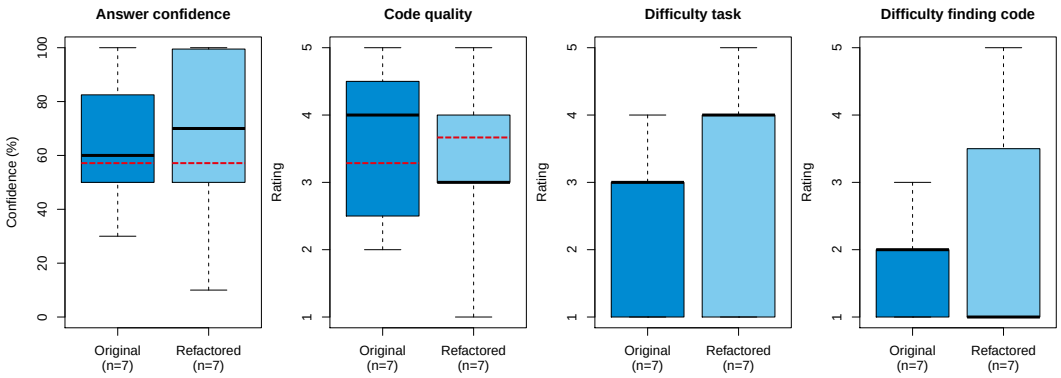
Overall

Figure 5.23 shows the data from the participants' responses to the questions after the experiment. In Figure 5.23a, the responses between the participants working in the original code and those working in the refactored code are compared. Not many differences between the two groups can be seen. The participants in the refactored code are divided about this assignment: for the questions relating to code quality, task difficulty and difficulty finding code the answers take the full range of 1 to 5.

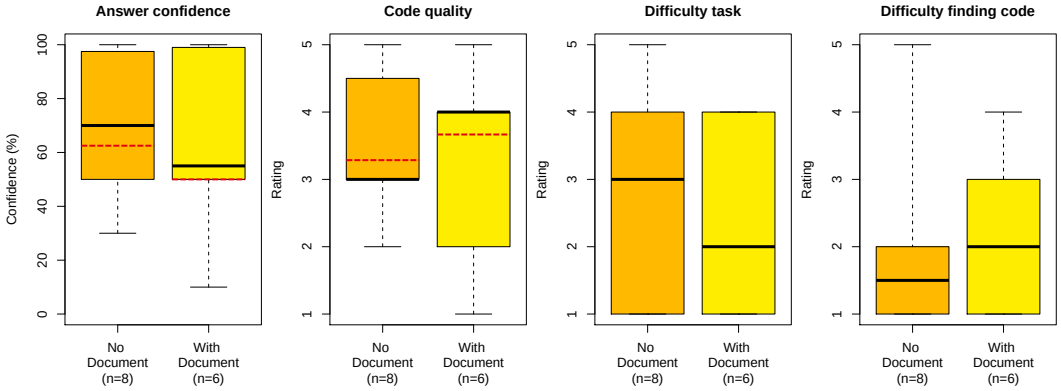
Some participants in the refactored code disliked the use of generics, stating that they found it confusing, did not understand what was happening or did not see the added value. Other participants liked the approach, stating that they liked the use of inheritance and that the code was short and clear. Of the 7 participants working in the refactored code giving comments about this exercise, 5 gave negative comments about the code quality and 2 positive. This indicates that the participants might be more negative about the code than is reflected in their ratings. For the 7 participants working in the original code, only 2 gave negative comments about the code.

In Figure 5.23b, the same data for the participants working with and without document is shown. There is little difference between the two groups. This is supported by all participants who had received the document stating that believed the document "not at all" helped them in solving this exercise.

In general, the data does not support the hypothesis that refactoring has improved performance for this exercise. While a slight decrease in performance was found, it cannot be concluded if this effect is small because of randomness, or because of the small size of the assignment. Similarly, the data also does not support the hypothesis that the document has improved participant performance.

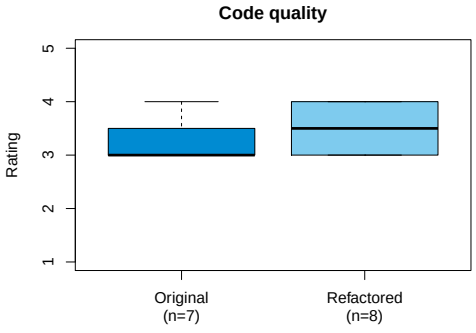


(a) Original versus Refactored code

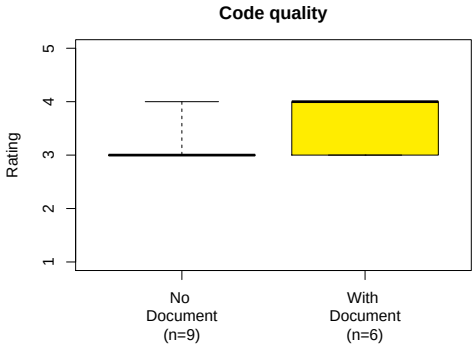


(b) No document versus Document

Figure 5.23: General data for exercise 6



(a) Original versus Refactored code



(b) No document versus Document

Figure 5.24: Participants' opinion on code quality

5.2.8 After the experiment

At the end of the experiment, participants were asked to give feedback about the code and the experiment in general. Figure 5.24 shows how the participants rated the code quality of all the code they worked in during the experiment. Figure 5.24a compares the scores given by the participants working in the original code with those given by the participants working in the refactored code, Figure 5.24b compares the scores for those working with and without the document. All participants gave a score of 3 or 4 out of 5 and because of this, little difference is seen between the participants in the different groups. The comments given by the participants after the experiment also do not show any clear difference in opinion between the groups.

Participants were not convinced the document helped them during the experiment. The 5 participants who rated the usefulness of the document rated it with a 2 out of 5 on average. Interestingly, the 2 participants with the highest score (3 out of 5), indicated that they used the document during the experiment. This indicates that while there is no evidence the document has helped the participants in understanding overall, it might serve a purpose as a reference document. Further research is needed to establish this.

The participants were also asked to indicate if any extra documentation or prior knowledge would have helped them during the experiment. Most participants indicated that extra documentation would have helped them, ranging from extra comments in the code to detailed class diagrams to simply get a general overview of the code. Interestingly, the participants who did not receive the document before the experiment often indicated a general overview of the code and its purpose would have been useful, all of which are mentioned in the document, while the participants with the document in general did not believe the document helped them. Those participants more often indicated more detailed documentation, such as class and sequence diagrams, would have helped them in the experiment.

5.3 Overall

The results per assignment are summarized in Table 5.3. A + in a cell signifies that a statistically significant improvement was found in that combination of assignment, treatment (i.e., refactoring or document) and measured data (i.e., completion time or correctness percentage), with a = signifying no significant difference[13]. For the time columns, this is measured by the result of the MWW test, with the direction of the effect provided by the W-statistic (i.e., $W > 0.5n_1n_2$ means a positive effect) and the statistical significance provided by the p-value.

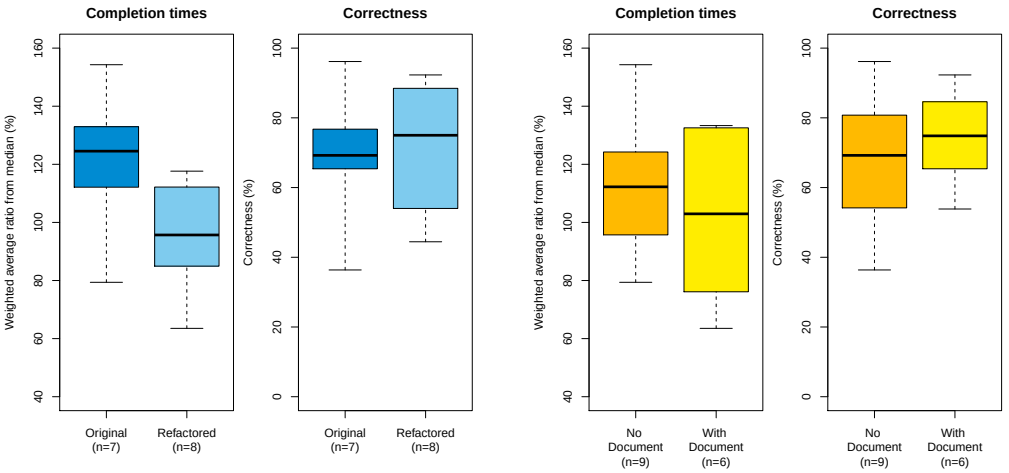
In order to compare the correctness percentages, 'correctness ranges' are constructed. These range from what the percentage would have been if one correct answer in that group would have been incorrect, to what the percentage would have been if one incorrect answer in that group would have been correct, and can be regarded as a confidence interval of sorts. For example: if the correctness percentage was 50% because of 5 correct answers out of 10, the range would be from 40% to 60%, from 4 correct out of 10 to 6 correct

[13] A − would signify a negative statistically significant difference, but none were found

Assignment	Refactoring		Document	
	Time	Correctness	Time	Correctness
1A1	=	+	=	+
1A2	+	+	=	+
1	+		+	
2A1	=	=	=	=
2A2	=	=	=	=
2A3	=	=	=	=
2	=		=	
3A1	=	=	=	+
3A2	=	+	=	=
3	+		=	
4A1	=	=	=	=
4A2	+	=	=	=
4	=		=	
5A1	=	=	+	=
5A2	=	=	=	=
5A3	=	=	=	=
5	=		=	
6A1	=	=	=	=
6	=		=	

Time: + when completion times improved significantly at $p < 0.05$.
Correctness: + when 'correctness ranges' do not overlap.
Exercise: + when hypothesis was concluded to be supported.

Table 5.3: Summary of results. A + denotes a positive effect observed in the group with treatment compared to the group without treatment.



(a) Original versus Refactored code

(b) No document versus Document

Figure 5.25: Combined correctness and weighted completion times for the whole experiment.

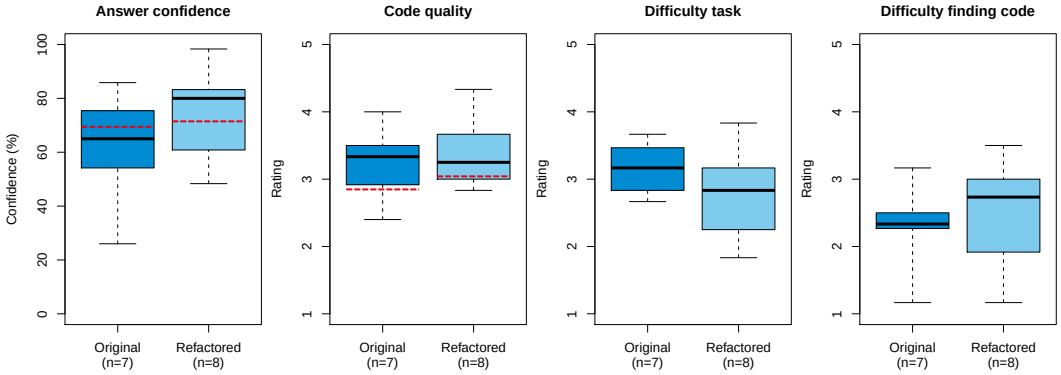
of 10, respectively. If the correctness ranges do not overlap, the difference is considered significant.

In most observations, little to no difference is found between the groups without treatment and those with treatment, or the difference can not be regarded as statistically significant: only 9 out of 52 combinations can be seen as statistically significant, albeit all with a positive direction.

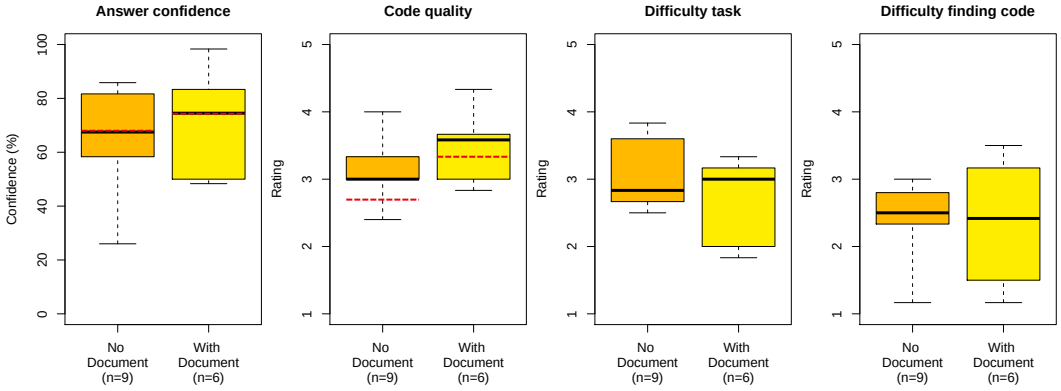
Additionally, the conclusions per exercise are also shown in Table 5.3. A + signifies that for that combination of exercise and treatment, it was concluded from the results that the hypothesis of improved performance was supported by the data. This was in the case of exercise 1 for both the refactoring and document treatments, and with exercise 3 for the refactoring.

Insight in the performance over the entire experiment is possible by combining the results from all assignments. To combine the completion times per participant, a weighted average of the correct solution times divided by the median correct solution time for that assignment is taken, weighed by the median solutions (meaning longer assignment have heavier weight). The completion times are divided by the medians to allow for fair comparison between participants, so the normalized time does not depend on the distribution of assignments answered correctly, but rather their performance in the assignments they answered correctly. The average is weighed by median solution times to give more importance to the longer solutions, where bigger improvements compared to the median are more impactful and more likely to be caused by the treatment, instead of noise (i.e., the difference between 1 and 2 minutes is less important than the difference between 15 and 30 minutes). The correctness percentages per participant are combined by simply averaging the correctness for all assignments the participants attempted.

The combined results are shown in Figure 5.25. In Figure 5.25a the results for the group



(a) Original versus Refactored code



(b) No document versus Document

Figure 5.26: Combined participant responses to questions after each exercise

working in the original code is compared to the group working in the refactored code. It shows comparable average correctness, with a median of 69% correctness in the group working in the original code, compared to 75% for the participants in the refactored code. It also shows a large difference in the completion times, with the median participant in the original code being on average about 25% slower than the median, compared to the median participant with the refactored code being on average 4% faster. This difference is also statistically significant (MWW test, $W = 47, n_1 = 7, n_2 = 8, p = 0.0323 < 0.05$), indicating that, overall, the refactoring did improve participant performance during the experiment.

Figure 5.25b shows the same data, comparing the groups with and without document. Here, the differences are very small. The median correctness percentages are the same as in the other comparison, with 69% correctness for the participants who did not receive a document, compared to 75% for the participants with document. The completion times are also close together, with the median participant without document finishing about 12% slower than the median time, compared to the median participants with document finishing about 3% slower. This difference can not be considered statistically significant (MWW test, $W = 33, n_1 = 9, n_2 = 6, p = 0.5169 > 0.05$).

Similarly, the responses to the questions after the exercises can be combined, by simply averaging the responses. This is shown in Figure 5.26. In Figure 5.26a, comparing the responses between the groups with and without refactoring, it can be seen that the participants working in the refactored code had slightly higher answer confidence and rated the task difficulties slightly less. In the comparison between the groups with and without document, in Figure 5.26b, no real differences between the groups are found.

5.4 Summary

This chapter discussed the background of the participants that participated in the experiment, and presented the results for each assignment individually. At the end of the chapter, the results are summarized and an analysis was made on combined data from all assignments. The results are mixed, with some assignments pointing to an increase in performance for the treatment(s), but other assignments pointing to a decrease. Overall, when inspecting Table 5.3 and Figure 5.25, an indication for improved performance in refactored code is found.

Chapter 6

Discussion

This chapter will first discuss some observations that may have influenced the results, after which the results described in the previous chapter will be related to the research questions. Finally, potential threats to validity are discussed.

6.1 Data limitations

The experiment consisted of 13 assignments grouped in 6 exercises, each of which are attempted by at most 15 participants divided over 4 groups by 2 treatments, and answered correctly by at most 12 participants, resulting in 186 timing and correctness observations. In addition, participants answered several questions after each exercise and after the experiment, resulting in 754 extra observations. However, there are several limitations in terms of data availability.

6.1.1 Small number of participants

A large limitation in this experiment is the small number of participants. With only 15 observations per assignment, effects are more likely to be obscured by chance and conversely, visible effects are more likely to be caused by chance. In addition, the small number of participants leaves little room for error in the experimental design, as there is generally not enough data to accurately correct these errors in a meaningful manner. For this reason, the experiment data was thoroughly inspected for potential shortcomings and these are discussed extensively in this chapter.

Additionally, the small number of observations make meaningful analysis of the interaction between the refactoring and the document treatment impossible, with a maximum of 5 observations per specific group (and a minimum of 2 in the case of the group with original code and document).

However, the small number of observations or the lack of statistical power does not invalidate this experiment. The resulting data is able to give strong indications of possible effects and provides insight into opportunities for further research. In addition, the quantitative data is supplemented with qualitative data, such as the participants' feedback about the

code and the recordings, to compensate for the lack of statistical power and provide extra insight into the possible effects.

6.1.2 Unbalanced data

Due to the unforeseen changes in groups they were left unbalanced, as discussed in Section 5.2.1. When comparing the groups working in refactored code with the groups working in the original code, the groups with refactored code had twice as many participants who have read the document (50% versus 29%). Similarly, when comparing the groups with and without document, the groups who have read the document were more likely to work in refactored code (67% versus 44%). As a result, the measured effects are correlated: observing an effect for one of the independent variables, may (partially) be caused by the other independent variable.

Statistical models, such as ANOVA, are able to deal with these unbalances by correcting for them [139]. Unfortunately, these models cannot be applied to this experiment, due to the small number of observations per group and the lack of normality in the timing data [139].

An alternative is to weigh the observations. In this method, observations in underrepresented groups will get heavier weight to restore balance. For example, when there are three observations for the refactored group where one is without document and two are with document, the observation without document would be counted twice. While this may improve the balance between the groups, this heavily increases the reliance on a small set of observations, which already is an issue with the overall small data set.

Because attempting to balance the data is either unsuccessful or introduces new issues, the analysis was done on the unbalanced data. The rationale is that the data as-is is the 'best guess' available for the underlying effects. The potential inaccuracies are accounted for as much as possible by validating the results with available qualitative data and with theory.

6.1.3 Missing data

For various reasons and at multiple points in the experiment, data is missing or excluded from the analysis, either on accidentally or on purpose. Some of these cases are already mentioned and explained in Section 5.2.1. Here an overview of cases of missing data is given, along with their possible effects on the results.

In some cases, some participants did not attempt an assignment. This had a variety of reasons. The most impactful was the early departure of P14 (discussed in Section 5.2.1), who is regarded as not having attempted any of the assignments due to not having submitted any data. In two cases, a participant did not attempt subsequent assignments in the exercise after being unable to finish the first assignment, because of time constraints. In three other cases, a participant did not attempt an entire exercise due to having to leave early. As a result, out of a possible 208 observations (16 participants times 13 assignments), 22 (11%) were marked as 'not attempted', with 13 of these stemming from the departure of P14.

Removal of these may slightly introduce bias in the correctness percentages, as it is reasonable to assume the assignment would have been more likely to be answered incorrectly, if the participant did submit an attempt, compared to the participants that did attempt. How-

ever, the impact is likely to be small (if any), due to the even spread of the disregarded observations over the assignments and treatment groups.

When analyzing the completion times, the times for the incorrect solutions are not taken into account. The rationale is that these solutions can not be considered a measure for understandability, as no understanding was demonstrated. An example of this is seen in the analysis of assignment 5A1 (see Section 5.2.6), where the completion times for the incorrect solutions were definitively lower. Of the 186 total attempts, 50 (27%) were considered incorrect.

The screen recordings made by the participants were not always available. In addition to the lack of recording from P5 (discussed in Section 5.2.1), recordings from one other participant (P4) were found to be incomplete, only containing data from exercises 1 and 2. This may pose a challenge when analysis on the recordings are performed, being unable to analyze the solving method of these participants. However, the impact of this is likely to be small, as for all cases where the recording was analyzed, there were sufficient other recordings available to give insight.

In one case (P9), all answers to questions after exercises relating to the document were discarded, because it was evident (and confirmed) he had misunderstood them (as discussed in Section 5.2.1). This decreases the (already low) amount of data on the effects of the document. However, the analysis is likely to not be impacted, because, in general, there was little variance in the answers from other participants, making it likely this participant would also have agreed.

6.1.4 Data format

Some factors in the format of the various data collected also posed a challenge in the analysis.

During the experiment, participants recorded their screen (as discussed in Section 3.5.2). This data is useful in giving insight on the steps the participants took when solving the assignments. However, the recordings are a large and unstructured form of data. This makes analysis unpractical and time consuming, as it has to be done manually. As a result, the recordings were only analyzed when there was a specific reason to do so, and thus were only partially examined. As a result, some extra insight may have been missed.

Additionally, participants were asked several open-ended questions after every exercise and after the experiment. These mostly regarded their opinion on the code and difficulties they encountered. Because of the free-form input, the topics the participants mentioned varied wildly. This means it is often not possible to gauge the opinion all participants had on a specific topic (e.g., naming, method size), because only a small subset of the participants gave their opinion on that topic. Drawing conclusions on those subsets is also challenging, as it cannot be assumed that the participants who did not comment on that topic have the same opinion on average that those that did.

6.2 Analysis of participant activities

The analysis of the results mainly depends on comparing the completion times between the different groups of participants. However, the time spent in solving a task may be used for a variety of activities. Particularly in the coding tasks participants may be vastly different in how they spent their solving time. This poses a risk to the validity of the results: if factors that were not accounted for in the balancing of the groups influence the completion times, comparison between the different groups is affected by confounding factors. In particular, the following questions arise:

- Are code assignments solving times a valid indication of understandability?
- Do the different solving strategies employed by the participants impact their completion times?
- To what extent is the experience of the participants a factor in their completion times?

To get an indication of how the participants spend their time while solving a coding assignment, the participants' recordings for assignment 2A1 (described in Section 5.2.3) are examined[1], categorizing their activities. These are split into four rough categories that could reasonably be extracted from the recordings:

- **Read** refers to reading and exploring the code before starting any implementation.
- **Test** refers to the implementation and running of tests.
- **Implementation** refers to the implementation of the functionality.
- **Cleanup** refers to any (non-testing) activities done by the participants after implementing the minimum functionality required for a correct answer, such as extra verification and refactoring.

Figure 6.1 gives a graphical representation of the participant activities. Note that the activity groups have the same order for every participant and do not necessarily reflect the actual order in which the participants performed the activities. In particular, some participants did their **Test** activities before **Implementation** and others performed them afterwards.

6.2.1 Using code assignments for understandability

> Are code assignments solving times a valid indication of understandability?

The coding assignments were designed so that little actual coding was needed, i.e., the reference implementation was never more than a handful lines of code. The rationale is that by minimizing the coding that needs to be done, the majority of the time spent by the participants is understanding the code, thus making the completion times a valid measure

[1] Assignment 2A1 is chosen because it was near the start of the experiment, had relatively high correctness and had little differences in the results between the different groups, allowing for easier analysis.

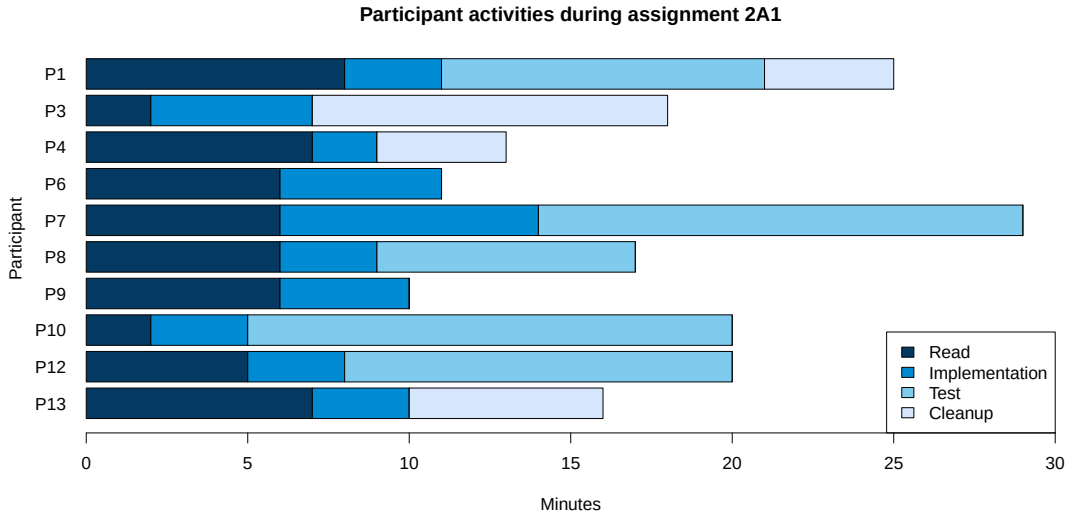


Figure 6.1: The activities the participants performed while answering assignment 2A1. Only participants who answered correctly and had a recording available are shown.

for the understandability of the code. If the participants spent a majority of their time on the implementation instead, this brings into question the validity of using coding assignments as a measure for understandability.

In Figure 6.1 it is noticeable that the participants spent only roughly a third of their time exploring and reading the code, spending the rest on testing, implementation and cleanup activities. By contrast, other research shows that developers spend at least half of their time understanding code [46, 33, 94]. While some difference is expected due to difference in measuring and environment, but the large difference is striking. From this, it may seem that participants do not spend the majority of their time understanding the code.

However, time spent while implementing tests and functionality also improves one's understanding of the code [130, 53]. For example, the participants often used the code completion aids in Visual Studio to explore the available properties and methods of classes. This is clearly done during implementation, and clearly serves to improve the understanding of the code. As a result, it is not justified to characterize any of the activities as not contributing to understandability. Therefore, these results do not lead to a conclusion about the validity of using coding assignments as a measure for understandability, but do indicate that further research might be needed the establish this validity. In this thesis, this further research is out of scope, so the use of coding assignments is rationalized by its use in earlier research, in particular in the research done by Ammerlaan [6], to allow for better comparison to their results (see Section 7.3).

6.2.2 Solving strategies

> Do the different solving strategies employed by the participants impact their completion times?

While solving coding assignments, participants employed different strategies. This was an intended side-effect of trying to maximize the realism of the experiment, as developers simply have different approaches in developing. These differences may have impact on the results, as different strategies may lead to higher completion times than others (but may have other advantages, such as increased code quality, maintainability or answer confidence). In other words, in different strategies, understanding the code might be a different percentage of what was done.

Various different approaches to solving the assignment are seen from the recordings. 5 of the 10 examined participants wrote new test code to solve the task. Of those 5, 3 wrote the test before starting the implementation (test-driven), while the other 2 wrote their tests after implementation. While writing tests is generally seen as improving productivity [50], this analysis indicates that this may not be the case in this experiment: with the exception of P3, all participants that did not spend time writing tests finished earlier than all participants that did.

An explanation for this may be found in the specifics of the software that was used during the experiment. The tests for the code used in the assignments were also relatively complex, with various domain objects needing to be constructed to allow testing the functionality. As this is done in several places in the test code, the test code also contains extensive code that helps in setting up the tests. In general, the unit tests in this software are very extensive, with the total test code having twice as many lines of code than the program code (whereas a ratio of one to one is commonly advised [132]). As a result, writing a test in this software can be seen as having to understand twice as much code, and can therefore lead to higher completion times. Arguably, the participants that wrote unit tests may have a better understanding of the code, but this is not measured in this experiment.

Analyzing other assignments may give other distributions of participant activities and consequently, different distributions for solving strategies. For example, while solving the assignments in exercise 4, no participant wrote extra tests. This is likely due to the fact the assignments in exercise 4 were in the context of the GUI, which is harder to test[2]. Additionally, participants are able to validate their solutions manually by running the code and observing the behavior directly in the GUI, so writing a test is not needed to validate the work. Another impacting factor is that the amount of guidance in finding the right starting point varied between assignments, so in some assignments, the participants likely needed more time to explore the code to find the relevant functionality.

The data gives a strong indication that the solving approach the participants employed indeed affected their completion times. On the other hand, it does not seem to have impact on the comparison of the times between the groups, as the different strategies are spread fairly evenly between the groups. Additionally, the small amount of participants makes it impossible to appropriately correct the times for such variations (see also Section 6.1.2). Therefore, the analysis of the results is not impacted.

[2]One participant mentioned trying to find a relevant unit test, but was not able to find one and was also not able to write one.

6.2.3 Participant experience

> To what extent is the experience of the participants a factor in their completion times?

The experiment contained a relatively large amount of participants that had little experience with C#: 10 out of the 16 participants had a year or less experience in the programming language. The use of inexperienced participants in this experiment is rationalized by the limited availability of participants and all groups being affected equally by possible effects of inexperience, due to the balancing of the groups.

The participants with little experience in C# are expected to perform worse than those with more experience for two main reasons. Firstly, they have less knowledge of the programming environment (such as the syntax, standard libraries and IDE), meaning they are expected to spend more time researching and understanding these basics. Secondly, as seen in Section 5.1, the participants with less experience in C# are also on average less experienced in developing in general, which intuitively results in lesser performance.

In addition, earlier research (see Chapter 7) indicates that refactoring might have different effects on inexperienced developers than on experienced developers. In particular, refactoring to a 'cleaner', but more complex code might be beneficial to experienced developers, while inexperienced developers may perform worse in the refactored code. The rationale is that the experienced developers are better equipped to deal with the added complexity due to extra experience and familiarity, while the inexperienced developers struggle to understand the more complex code.

During the experiment, all 10 participants that had a year or less experience in C# reported at some point that a lack of experience and/or knowledge in the C# language or the Visual Studio IDE made the tasks more difficult for them. Some of these were specific to C#, such as unfamiliarity with LINQ[3], while some other comments can be regarded as pointing to general programming inexperience.

It is difficult to determine the actual effects of the inexperience on the developers in this experiment. Simply comparing completion times between experienced and inexperienced developers does not suffice, as experienced participants may employ different solving strategies, such as writing tests or aiming for solutions with higher quality. The time analysis in Figure 6.1 shows that participants inexperienced in C# spent slightly more time on implementation on average than the experienced participants (4.3 versus 3.3 minutes), but similarly to what is discussed in Section 6.2.2, this does not tell the whole story. For example, spending time on writing unit tests may result in less time needed for implementation, but increases the total time spent. Comparing the completion times shows a small difference (average of 18.3 versus 17.3 minutes), but this is not enough to determine the effect of inexperience on performance, if any.

While there is not enough data to determine what effect inexperience had, it does not impact the analysis of the results, as the groups were balanced in respect to experience.

[3] `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/`

6.3 Differences between tasks

The results, as summarized in Table 5.3, show large differences in performance in the various tasks. This makes it difficult to draw general conclusions on the effects of the treatments on the participant performance. Several factors can be identified that may explain this difference.

6.3.1 Task type

The experiment contained two different types of tasks, coding tasks and multiple choice questions (see also Section 3.6). Both tasks aim to measure understandability, but both do so in slightly different ways. The validity of using coding tasks to measure understandability is previously discussed in Section 6.2.1.

Multiple choice questions can be seen as a more straightforward way of measuring understandability: understanding the code is the only activity that is needed to answer correctly, so there are little other factors that influence time spent. However, they also have their downsides.

Firstly, answering a multiple choice question takes significantly less time than a coding assignment. Of the 8 multiple choice questions in this experiment, in only two of them participants took longer than 5 minutes to answer correctly: 1A2 and 3A2. In such short times, possible effects are less visible and more confounded by other factors, such as small distractions and the granularity of the timekeeping. A difference in completion times between 1 and 2 minutes may be due to a great number of things, while a difference between 15 and 30 minutes is more telling. Interestingly, the two multiple choice assignments where the participants did spend considerable time answering, are among the assignments showing the biggest (positive) effects.

Another downside is the possibility of guessing the answer. This inflates the correctness percentage, as more participants may answer correctly than actually understood the code. Additionally, the possibility of guessing the answer may also lead to participants giving up earlier, especially when they find the code hard to understand, making the time spent on the task less than one would spend to understand the code.

Furthermore, the presence of the options may also guide the participants in their solving process, giving them hints at what the answer may be and perhaps concrete starting points in the understanding process. Answering the question may also have been the result of eliminating other options, instead of understanding the code enough to be to have answered without having options to choose from.

A possible solution for the caveats of multiple-choice questions may be found in asking open-ended questions instead. However, these are also not without downsides. For example, when asked about the functionality of code, participants may simply ‘translate’ the code into English, without actual understanding of its peculiarities. Additionally, participants may have different preferences in the extent of understanding they have before answering, or may answer in different levels of detail, both influencing time spent without a difference in understandability. These reasons, along with the lack of precedent found in earlier research, is why multiple choice questions were used in this research. Nonetheless,

it is interesting for further research to examine to different types of tasks and their use in measuring understandability.

All of these caveats are likely to decrease the effect size, as solving the task can be seen as easier than understanding the code. However, analysis is not impacted, as comparison between the groups is still valid, due to the groups being impacted equally. In addition, the different downsides of both task approaches (coding assignments and multiple choice) is one of the reasons a mix of both types was employed in this experiment.

6.3.2 Task difficulty

Between tasks of the same type, there also is variation in the difficulty of those tasks. Differences are expected due to varying complexity of the code, but on the same code the task difficulty can also vary. As a result, completion times and correctness can be different, while the underlying understandability remains the same. This does not impact analysis for a single task, as long as the tasks are the same inherent difficulty for all groups. It does impact potential comparison between the different tasks, in the sense that one cannot say that one part of the code is more understandable than the other because the completion times and correctness of the tasks on that part of the code are better.

In general, finding the balance in the difficulty of the task is challenging [97, 113]. The tasks need to be difficult enough so that solving them is not trivial to allow for variability in the results, but not too difficult to make sure most of the observed effect comes from understandability (as opposed to other factors, such as maintainability when the changes made are extensive) and to prevent the experiment from taking too long. The effect of tasks being too easy is seen in this experiment in most of the multiple choice questions: only in 2 of the 8 multiple choice questions (1A2 and 3A2) participants took longer than 5 minutes to finish. As a result, the timing data is not able to show any effect in those 6 tasks.

In a single case it could be argued that the task difficulty was considerably less for the participants working in the refactored code than those in the original code. In assignment 4A2 (described in Section 5.2.5) the changes needed in the refactored version amounted to simply calling a method with a clear name, while in the original version the participants had to either duplicate code, call an existing method in a unintuitive way or refactor the existing functionality to allow for better re-use. Arguably, it is natural that refactoring by splitting up methods increases the chance of possible re-use [47]. As the assignment was not designed to explicitly favor the participants working in the refactored code, it was decided to include it in the analysis.

6.3.3 Refactoring quality

Another factor that can explain differences between tasks is the quality of the refactorings made in the code. While refactoring aims to keep the functionality the same while improving the structure of the code [47], the act of refactoring is still performed by humans, so mistakes can be made. Apart from the possible effects of a ‘perfect’ refactoring (which is under study here), mistakes made in refactoring may introduce bugs or inadvertently re-

move information. In addition, the refactoring may have been of low quality, for example splitting up functionality that is better suited together or introducing more confusing names.

An example of an error made can be seen in assignment 2A2 in this experiment (see also Section 5.2.3). During refactoring, the `offset` parameter was moved from the method to the class constructor. Unfortunately, the existing XML documentation describing this parameter was not copied over, resulting in the participants working in the refactored version having less information than those working in the original version, which was likely a factor in the reduced performance by the participants working in the refactored code. The assignment was kept in the analysis, as the error made during refactoring was non-functional, and similarly to the potential positive effects of increased re-use as seen in assignment 4A2, the potential negative effects of errors made during refactoring are a realistic byproduct of refactoring.

This shows that, while great care was taken in performing the refactorings and their theoretical basis, mistakes can happen. In this thesis, all refactorings are described extensively in Chapter 4. This allows the reader to judge the quality of the refactoring for themselves.

Additionally, all refactoring was done using the Clean Code book [82] as basis for the refactoring, instead of the author's personal preferences. However, the Clean Code book only contains guidelines, instead of hard rules ("Clean code is not written by following a set of rules." [82]). This makes their application open to interpretation and following the guidelines a trade-off. Too little results in unclean code, while too much may lead to over-engineering. These interpretations are discussed in Chapter 4 to allow the readers to judge their applicability.

However, because the application of the guidelines of Clean Code is open to interpretation, this means that the process of refactoring to apply to the guidelines of Clean Code may result in a different outcome every time. As a result, it is not possible to draw definitive conclusions on the effects of Clean Code based on one set of refactorings. However, the results from this experiment can give an indication of a typical application in a typical setting.

Finally, the size of the refactoring is likely to play a role in the extent of the effect. Intuitively, when there are only small changes, the effect will be small as well. In this experiment, the code used in exercises 1 and 3 was arguably modified the most by refactoring. These two exercises are precisely the two that have shown an increase in performance for the participants working in the refactored code.

6.3.4 Learning effects

Although the code under study was different between the exercises, the experiment takes place in a single code base. Therefore, the participants are expected to gradually learn more about the code over the course of the experiment, which is expected to improve their performance. To keep this from influencing the results, the order of the assignments was the same for every participant, so any learning effects were equal for all participants.

Another possibility is that answering an assignment incorrectly has negative impact on the performance of the subsequent assignments, compared to answering correctly. The rationale for this is that answering incorrectly means the previous code was not (well) un-

derstood, so if the same or related code would return in a subsequent assignment, extra time is needed for understanding. Between the different exercises, the risk for this is deemed low, as there is little overlap in code by design. However, there is overlap in the domain concepts between exercises. For example, a better understanding of the code in Exercise 1 (the construction of `Area` objects), may help in the understanding of the `Area.Contains` algorithm from Exercise 3. Additionally, within exercises the tasks are generally about the same or similar code, where performance in an earlier task may realistically impact performance in a subsequent task. In particular, time spent during the first task may have partially been used for understanding code that is (also) used in the second task.

The size of this effect is unknown, as the small amount of observations prevents uncovering this using statistical analysis. However, as seen in Figure 5.25, there seems to be no substantial difference in the correctness percentage between the different groups, making it likely that this possible effect did not occur.

The treatments may still cause other learning effects. It is not expected that refactoring had impact on the learning effects. For example, similar research by Krein et al., geared specifically to design patterns, found no evidence that learning effects were influenced by the presence of design patterns [71], which can be seen as similar to the presence of refactoring in this study.

For the document, learning effects were part of the hypothesized motivation for possible positive effect on performance. It was expected that the participants would 'learn' about the code through the knowledge in the document, and thus profit from the learning effect. However, the general lack of effect for the participants who had received the document compared to those who have not, make it likely that this effect never occurred.

If the results are impacted due to this effect, it can be seen as a realistic side-effect of more understandable code. If one would start working in code where the basis of the functionality is easily understood, this better foundation of understanding may help in further understanding of the system. This may also be an interesting direction for further research.

6.4 Experimental setting

As detailed in Chapter 3, the experiment was designed to be as realistic as possible. For participants, the experiment should have been as close to their normal work environment as possible, taking into account inherent limitations caused by the experimental setting.

Before the experiment, three differences were identified and communicated to the participants:

- The participants did not have access to version control history, because that would have revealed to the participants whether they were working with the refactored or the original code. While developers do sometimes use version control history when developing [126], it was expected from personal experience that this would not impact realism much, as this information is not often used for the understanding of the code.

- The participants were not allowed to discuss with each other during the experiment, because it was expected that it would have introduced too much noise in the data.

While developers may normally choose to discuss complex code with colleagues when they are having difficulty, it was expected that the impact on the realism was small, as the exercises are shorter than the expected threshold for when one would ask a colleague developer.

- The participants were instructed to stop with the task when they were 'reasonably certain' they were finished, or when more than 30 minutes had passed. This was intended to reduce the time the participants spend on development factors not directly related to understanding the code, such as refactoring their solution to be more clean, as well as reducing overall time needed.

Of these, only the last is expected to have influenced the results. Some participants mentioned the time pressure of the 30 minute 'deadline' as impacting their behavior. On the other hand, there was only a single case where a participant had to give up due to reaching the time limit. As discussed earlier in Section 6.1.3, there are a few additional cases where the participant gave up, despite not having reached the time limit. It is not likely that the looming deadline was the cause for this, but rather other motivations such as having to leave.

In addition to the expected differences in realism, during and after the experiment several other factors relating to the experimental setting were identified that may have substantially influenced participant performance.

Most notably, the experiment was done in the evening, after the participants had already completed a full work day. This meant they were tired and their performance was worse than would have been during the work day [15]. This is supported by several participants mentioning being tired or less sharp than usual. The effect was expected, but the effect was estimated to be a lot smaller. Several effects can be seen from this.

Firstly, the experiment was designed to take a maximum of 2 hours. This was tested in the pilot experiment with both an experienced and an inexperienced developer, and was found to be realistic. However, more than half of the participants took longer than that, spending up to 30 minutes extra. This is attributed to the fact that the pilot experiments were done during the work day, while the experiment itself was during the evening.

Additionally, nearing the end of the experiment, it was getting later than the participants might have expected beforehand (as they were told the experiment would take a maximum of 2 hours). This likely resulted in participants rushing in the later assignments, as evidenced by conversations with several participants confirming this. The effects of this can be seen in Exercise 5, where the correctness of the participants is relatively low and the high amount of observed mistakes made by the participants that can be seen as sloppy.

It is expected that analysis of the results is not impacted, as the effects have influenced all groups equally. A possibility is that the impact of, for example, tiredness may be larger on more complex code than on simpler code, which would mean analysis is impacted. However, no evidence of this is found in the literature, so this effect is not assumed.

6.5 Document specifics

Some specifics of the project knowledge document may influence the effectiveness of the document and with it, its effect on the performance of the participants who have read it.

Firstly, the quality and contents of the document are likely to be of large influence. If the information in the document is incorrect, badly explained or irrelevant for the participants, no positive effect can be expected. To lessen the possibility of incorrect information, the entire document was checked with the project developer to verify correctness. Additionally, the document is included in this thesis in Appendix A, so the reader can judge the quality of the document themselves.

The document gave a high-level overview of the system, explaining the architecture and general execution flow. On the other hand, the assignments during the experiment focused on lower level details and often gave a specific starting point for solving the task. This meant that architectural knowledge was likely to be of less importance during the experiment. Arguably, the contents of the document are thus less relevant for the specific setting of this experiment, and may be more useful in the regular work setting of the participants. More specifically, the document may not have helped in improving the understandability of the code, but may help in other areas such as maintainability due to increased knowledge of the architecture and the reasoning behind it.

Another possible factor is the extent to which the knowledge of the document was retained by the participants after reading. Although the document was sent shortly (five days) before the experiment, it is possible that some of the knowledge had already faded by the time of the experiment. Additionally, some participants may have read the document more thoroughly than others. We have no data to measure this and it did not come up in the conversations with the participants afterwards. One participant did remark during the experiment that he forgot most of the contents of the document.

6.6 Effects

With the results and their limitations analyzed, there is now enough information to revisit the research questions, as posed in Section 1.2. The research questions concerned the effects of refactoring, an introductory document and their interaction on code understandability.

6.6.1 Refactoring

RQ1 is as follows:

> Does refactoring according to the guidelines of Clean Code have a positive effect on understandability?

As summarized in Table 5.3, in 2 assignments a statistically significant positive effect of refactoring on completion time and in 3 assignments a positive effect on correctness percentage was measured (out of a total of 13 assignments). Looking at the combined data from Figure 5.25a, a statistically significant improvement in the completion times between the group working in the refactored and those working in the original code can be seen,

while little difference in correctness is visible. However, while not statistically significant, some negative effects can be seen, in particular in assignments 2A2 and 4A1.

Overall, this gives an indication that the refactoring according to the guidelines of Clean Code may have improved performance, and thus understandability, during the experiment. However, we must be careful in drawing strong conclusions from these results, due to the caveats mentioned earlier in this chapter. Nonetheless, the generally positive results give opportunity for further research to dive deeper in specific factors and determine their influence on understandability.

6.6.2 Document

RQ2 is as follows:

> Does adding project knowledge have a positive effect on understandability?

As summarized in Table 5.3, in 2 assignments a positive effect of the document on completion time and in 4 assignments a positive effect on correctness was measured. In contrast, in no assignments a negative effect on completion time and in 4 assignments a negative effect on correctness was measured. The combined data shown in Figure 5.25b does not indicate any difference in terms of overall completion time or correctness. Additionally, the participants who had received the document did not believe it helped them during the experiment. Therefore, there is no indication that the document had a positive impact on understandability.

One likely explanation for this, as stated in Section 6.5, may be found in the mismatch between the document and the experiment. The document gave a very general overview of the software and its architecture. On the other hand, the experiment focused on more specific parts of the code, where the general knowledge may not have been useful.

The general overview could have helped the participants in finding the relevant code faster, because they might have better known where to look due to their increased knowledge of the architecture of the system. However, the results do not show significant differences in the reported difficulties in finding the relevant code. While it could be argued that the assignment texts helped the participants greatly in finding the relevant code, the effect is also not visible in assignment 4, where no such guidance was given.

Interestingly, as described in Section 5.2.8, after the experiment some participants who did not receive the document mentioned that they expected to have performed better if they would have received documentation beforehand, mentioning the type of information that was present in the document used in the experiment. By contrast, the participants that did receive the document indicated that they believe they would have been helped by more detailed information than was present in the document.

In addition to the lack of indication of an effect on performance in this experiment, the concept of an architecture document has other pitfalls, such as being quickly outdated and rarely fulfilling the exact information needs of the developer [73]. Therefore, on the basis of this research, no recommendation on drafting a document for the purpose of increasing understandability can be made. Further research needs to be done to determine better ways of using documentation to introduce new developers into a project.

6.6.3 Interaction between refactoring and document

Finally, **RQ3** is as follows:

> Does both refactoring and adding project knowledge have a positive effect on understandability?

With experiments with multiple factors, such as this one, the interaction between the treatments is an interesting point of study [139]. Unfortunately, this experiment did not have enough observations to draw conclusions on the interaction. However, the lack of effect from the document treatment makes it unlikely that such an interaction is significant, as the hypothesized causes for interaction require an effect in the use of an introductory document.

6.7 Threats to validity

Several threats to validity of the experiment can be identified. These will be discussed below, categorized into Conclusion, Internal, Construct and External Validity, as proposed by Cook [32]. The lists of threats as proposed by Wohlin et al. are used as a checklist to identify potential threats [139], with some omitted if they were defined not applicable to this experiment.

6.7.1 Conclusion Validity

Wohlin et al. describe conclusion validity as follows:

> The conclusion validity is concerned with the possibility to draw correct conclusions regarding the relationship between treatments and the outcome of an experiment. [138]

Wohlin et al. define the following threats [139]:

- *Low statistical power*: The experiment has low statistical power, largely due to the small amount of participants. This refrains us from drawing strong conclusions from the results. However, the data, supplemented with qualitative data, does give indications of possible effects. This is discussed more extensively in Section 6.1.1.
- *Violated assumption of statistical tests*: All statistical tests in this experiment have been performed using the Mann-Whitney-Wilcoxon (MWW) test. The choice for a non-parametric test is motivated by the fact that normality of the completion times can not be assumed [59]. The choice for the MWW test is elaborated on in Section 5.2.
- *Fishing and the error rate*: The statistical significance level used throughout the experiment is the commonly used $p < 0.05$ level. It can be argued that the error rate of the experiment is a lot higher, because multiple analyses were performed. However, the analyses are done on different tasks which apply to different parts of the code. Furthermore, the results are supported by additional qualitative data.

- *Reliability of measures*: All measures used in the experiment are objective or have been detailed so the reader can assess their reliability (e.g., the assignment texts for the experiment). As completion times were self-reported by the participants, these could be inaccurate. However, the screen recordings allowed for verification and aside from a single case (as described in Section 5.2.1) no significant inaccuracies were found.

- *Reliability of treatment implementation*: Both treatments were applied in the same way too all participants. In the case of the refactoring treatment, the participants were supplied the code during the experiment, so no difference in the application of the treatment is present. Additionally, the submitted code by each participant was checked to verify if they worked in the correct version. In the case of the document, all participants in the relevant groups received the document at the same time. A question asked after the experiment verified whether the participants did actually read the document, resulting in one participant moving groups, as discussed in Section 5.2.1.

- *Random irrelevancies in experimental setting*: The experiment was performed in a single room with all participants and the author present, in an otherwise empty office without additional noise. Because of the error in the code distribution mentioned in Section 5.2.1, half of the participants did have to wait an extra 15 minutes at the start of the experiment. However, there is no indications that this influenced their performance. Additionally, whenever a participant was finished with the experiment or had to leave, they left the room. Other participants could have regarded this as an interruption. However, there is no indication that this influenced performance.

- *Random heterogeneity of subjects*: As discussed in Section 5.1.1, there was a large variation (heterogeneity) in the participants in the experiment. This improves generalizability, but could be a threat when the differences between the participants cause for greater variation than the treatments. This risk was mitigated by balancing the groups, so that each group has approximately equal variation, enabling the comparison between groups.

6.7.2 Internal Validity

Wohlin et al. describe interval validity as follows:

> The internal validity is concerned with factors that may affect the dependent variables without the researchers knowledge. [138]

Wohlin et al. define the following threats [139]:

- *History*, *Testing* and *Instrumentation*: The experiment was performed only once, with all participants performing the experiment at the same time. Additionally, participants all received the same material (i.e., the documents containing the tasks and questions)

during the experiment, only differing slightly in relation to the treatments[4]. As such, there is no risk that differences in time or material influenced the results.

- *Maturation*: It is likely that the participants reacted differently at the end of the experiment, due to learning effects (discussed in Section 6.3.4) and tiredness (discussed in Section 6.4). However, as these effects were concluded to affect all participants similarly, analysis between the groups is not impacted.

- *Statistical regression*: This threat relates to comparing performances between multiple experiments, and is not applicable here.

- *Selection*: Participation in the experiment was voluntary, meaning the participants may generally be more motivated than the general population, making them less representative [139]. In this experiment, this is not a threat to *internal* validity, as this impacts all participants equally and thus comparison between groups is not affected. Its possible threat to *external* validity is discussed in Section 6.7.4.

- *Mortality*: During the experiment, a single participant dropped out entirely and some participants did not answer some tasks, as discussed in Section 6.1.3. As the loss of data was spread across the multiple treatments, the validity of the experiment is not affected.

- *Ambiguity about direction of causal influence*: The threat of difference in performance being caused by other factors than the treatments is mitigated by the presence of the control group(s) and group balance, meaning that the only differences between the groups with and without treatment are the treatments themselves.

- *Interactions with selection*: When different groups behave differently, or interact differently with other threats (such as maturation, meaning that the groups mature at different rates, for example that one group learns faster than the other), the validity of the experiment is threatened. This is mitigated by the balancing of the groups, such that all groups behave similarly.

- *Diffusion of imitation of treatments*, *Compensatory equalization of treatments*, *Compensatory rivalry* and *Resentful demoralization*: These threats relate to situations that might occur when the control group knows they are in the control group, or when they are treated differently. In this experiment, the participants were not aware in which group they were, so this threat was not present. Because of the error in code distribution mentioned in Section 5.2.1, some participants may have figured out there were different groups relating to the code, as the error only affected half of the participants (i.e., those who should have received the refactored code). However, this still

[4]To be more precise: Between the groups with and without refactoring, the task texts differed slightly whenever the names of the code units that they referred to were changed in the refactoring. The participants who had received the document beforehand had a few extra questions compared to the participants who did not receive the document. These questions related to the document, and thus were not relevant for the participants who did not receive it.

did not tell them whether they received the treatment or not, just that the code was different. Conversations with some of the participants after the experiment indicate that participants either did not read too much into the error, or simply did not care at that moment whether they had received refactored code or not.

6.7.3 Construct Validity

Wohlin et al. describe construct validity as follows:

> The construct validity is related to the relationship between the concepts and theories behind the experiment and what is measured and affected [138]

Wohlin et al. define the following threats [139]:

- *Inadequate preoperational explication of constructs*: The constructs used in the experiment are made explicit and substantiated in the rest of this thesis. In particular, the use of completion times and correctness as measures for performance are built on previous research, explained in Chapter 3 and discussed earlier in this chapter. As such, the constructs are sufficiently defined.

- *Confounding constructs and levels of constructs*: This threat occurs when the presence of a construct does not explain the effect, but the level or extent of the construct may. As discussed previously in Section 6.6, this is possibly the case for both treatments. In particular, the content, extent and quality of the refactorings and the document are likely to influence the results. More 'in depth' research to these effects is an opportunity for further research.

- *Interaction of different treatments*: The interaction between the two treatments was a specific point of study, as discussed in Section 6.6.3. As mentioned there, there is not enough data to conclude whether there is an interaction, but it is unlikely that one exists due to the lack of effect from the document.

- *Interaction of testing and treatment*: Participants may have behaved differently than they would have in their day-to-day work due to being in a testing environment, as discussed in Section 6.4. However, as concluded there, the results are likely not impacted, because all groups were impacted equally.

- *Restricted generalizability across constructs*: In this experiment the understandability of the code was measured, using the productivity and correctness of the developer as measures. It is possible that the treatments have unintended (negative) effects on other constructs, such as maintainability or testability. While we have no reason to believe this is the case, the experiment itself does not provide enough evidence to rule out the possibility.

- *Hypothesis guessing*: The participants were told that the experiment related to refactoring and Clean Code, but were not told how they were evaluated or the goals and hypothesis of the experiment. As they also did not know what the treatments were

and whether they received any, it is unlikely that participants guessed the hypotheses and adjusted their behavior accordingly.

- *Evaluation apprehension*: Most of the data from the participants was self-reported, in the form of questions and timekeeping. The participants may have 'embezzled' this data to try to look better. However, there is no indication that this is the case. When the recordings were inspected, such as when inspecting specific assignments or for the analysis done in Section 6.2, the timekeeping data was also checked with the times shown on the screen, and no inconsistencies were found[5]. Additionally, the participants were explicitly told beforehand that individual performance would not be shared, to alleviate some fears they might have had.

- *Experimenter expectancies*: All material used during the experiment was formulated to avoid the effect of steering in the questions. Additionally, all material was checked beforehand on validity and was tested in the pilot experiments. Most of the material is also detailed in this thesis, allowing the reader to judge. During the experiment, all data collection, with the sole exception of the conversations afterwards, was done using the same, pre-written, material for all participants. Furthermore, no hints were given during the experiment.

6.7.4 External Validity

Wohlin et al. describe external validity as follows: "The external validity is related to the ability to generalize the results of the experiments." [138] Wohlin et al. define the following threats [139]:

- *Interaction of selection and treatment*: If the population selected in the experiment is not representative, the results cannot be generalized to a larger population. As described in Section 5.1.1, the participants in this experiment are varied in terms of background and experience. The participants can carefully be regarded as a representative sample for consultants within ALTEN that would work on code such as that was used in the experiment. A factor that needs to be taken into account is that all participants were volunteers, and therefore more motivated than developers would be in general. It is unknown to what extent the participants are representative for the general population of software engineers, so we cannot generalize the results of this experiments to software engineers in general.

- *Interaction of setting and treatment*: If the setting of the experiment is not representative, the results cannot be generalized to general practice. For this reason, the set up of the experiment was kept as realistic as possible, as discussed in Section 6.4. Additionally, the chosen code base was selected to be as representative as possible, as discussed in Section 3.1. Finally, the chosen tasks have been checked with the

[5]The single exception is the inconsistencies of a single participant, as mentioned in Section 5.2.1. The reason for this was not found in false reporting, but in human errors. This did not influence the results, as the times were corrected.

developer of the project to determine whether they are realistic and representative of tasks in regular work. They were deemed representative for short tasks, meaning the results are likely generalizable to all short tasks. The results of this experiment cannot be generalized to larger tasks.

Chapter 7

Related Work

This chapter discusses work related to this study, in order to establish the historical context and the contributions of this study. The scope of this study is to study the effects of refactoring on code quality, specifically understandability.

The topic of refactoring has been widely studied. In order to limit the scope, we will focus on empirical research on the effect of refactoring. Because refactoring is merely a tool that can be applied for a multitude of reasons and performed in a variety of ways, we will organize the studies by the basis the authors used for their refactorings. First, the studies that refactor with a general improvement goal will be examined. After that, studies which focus on refactoring to GoF Design Patterns [51] are discussed. Finally, studies which use Clean Code [82] as a basis are analyzed, finishing with a conclusion.

7.1 General

In 2004 Pizka performed a case study in which a degraded commercial system was refactored over a period of five months [108]. At the end of the refactoring period, the experimenter concluded that refactoring was harder and more time-consuming than expected, and:

> the usefulness of refactoring for restructuring purposes without concrete need is doubtful because it is unclear whether the increased beauty will simplify or aggravate future changes [108]

As a result, the author concludes that "[Refactoring] is of little help for retrospective restructuring of larger code bases for quality reasons.". However, without specifics about the refactorings it is hard to verify this claim.

Some details in the article do allow for comparing this result to today's expectations. The author attributes many of the difficulties found to limited tool support. Because tooling has improved significantly over the past 14 years, these findings are not likely to hold nowadays. In addition, the paper makes some dubious claims that either suggest a large change in technology over the years, or inexperience in refactoring. For example, the paper claims the `Introduce Parameter Object` refactoring could lead to performance penalties up to

290%, while the only functional difference should be the creation of an object, an operation that should be virtually costless. Another example is the claim that refactorings can only be syntactical in nature, and thus being unable to fix weak algorithms. However, earlier in the paper the author mentions the `Substitute Algorithm` refactoring, which describes exactly such a refactoring.

Wilking et al. performed an experiment where they let 12 students create a small program written in C, running on a micro-controller [137]. The participants were divided in two groups: the first group had to regularly refactor their programs, whereas the other group was instructed to regularly document their programs. The authors consider the activities from the second group as a placebo in order to avoid the Hawthorne effect. The activities were controlled by interrupting the participants every 20 minutes, requiring them to either walk through a refactoring checklist or write documentation. Wilking et al. studied maintainability by randomly deleting a line of code from the program, and timing how long it takes for the participant to fix the bug. They found that there was a slight (but not statistically significant) increase in maintainability for the refactored versions. Modifiability was studied by drafting new requirements, and measuring the LOC added and the time it took to complete them. They found that the refactored versions required more LOC and more time, indicating lower modifiability. They did find a decrease in memory usage in the refactored versions.

While the results in general do not seem to disagree with other research, there are some factors that limit the generalizability to this study. In particular, their use of the C language is a point of concern. Most of the principles and material on refactoring focus on object-oriented design, which might not be immediately beneficial for procedural programming like in C[1]. In addition, the realism and generalizability of a random bug generated by removing a random line of code from the system is questionable.

Moser et al. present a case study in which a new software system was developed by one professional developer and three undergraduate students [95]. The development consists of five iterations, and at the end of the second and fourth iteration a user story that explicitly mentions refactoring was written. The authors found that productivity (measured in LOC per hour) was higher in the iteration following the refactoring. In addition, they found that code quality (measured in metrics) was higher right after the refactoring. The results suggest that refactoring improves productivity, but the authors are also clear that the generalizability is limited due to it being a single case study on a small software system. In particular, there is little evidence the higher productivity is caused by the refactorings. For example, the higher productivity in the last iteration can just as likely be attributed to the team rushing to finish the project.

In 2016 Wahler et al. perform a case study where they examine a large refactoring of scientific simulation software [135]. The software was written by scientists without formal software engineering training. As a result, the maintainability of the project was considered very low, as evidenced most strongly by the fact that adding new components to the system was stopped completely because it was considered to be too demanding, even though there

[1] While object-oriented programming is technically possible in C, the authors explicitly state that: "Only non-object oriented programming features were used during this experiment." [137]

was still business desire to do so. The authors back this up with a subjective assessment of the changes needed to add a single component, and an objective assessment using static analysis tools. After a large part of the refactoring was done, adding a new component was estimated to be completed 75% faster. In addition, static analytics shows a large improvement in the amount of duplication and other issues found.

Interestingly, this result seems to directly contradict the results of the earlier study by Pizka [108], while both study refactoring done when the project is already mature. However, in the study by Wahler et al. the refactoring can be seen as supporting a specific type of change (adding new components), while the study by Pizka does not seem to have such a focus. In addition, the refactoring studied by Wahler et al. can be seen as more expertly guided and supported by better tooling.

7.2 Design Patterns

In 1995 Gamma, Helm, Johnson, and Vlissides published their book on Design Patterns [51], in which they describe several reusable architectural patterns that solve common design problems. In their own words:

> Design patterns (...) are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. [51]

Over the years it has become a standard work in practice and has attracted much research [9]. However, the amount of empirical research in the field can be seen as limited [140].

In a controlled experiment, Ng et al. took a clean open source system and performed refactorings in it, specifically to support a certain change task [97]. An experiment was then performed with inexperienced (undergraduate students) and experienced (postgraduate students with at least a year of working experience) participants. They were given either the unrefactored or the refactored version and were asked to perform a change task, which was explicitly supported by the refactoring. The authors found that both groups performed the change task faster in the refactored version, even when taking into account the time it took to perform the initial refactorings.

Their results are interesting, but while Ng et al. have shown that refactoring was beneficial when it explicitly supports a certain change task, we are more interested in a more general improvement. The rationale is that you cannot always predict what change tasks will have to performed in the future: there may be plenty of components in the system that will never be extended upon. In those cases, the investment of proactive refactoring will never get its return in the form of a change task. In this study, we are interested in if such a return can be achieved by in improvement in understandability, using the rationale that code is more often read than written.

A controlled experiment by Prechelt et al. [109] examined four simple systems of which each two different versions exist. In the ‘pattern version’, the functionality was implemented using a design pattern. In the ‘alternative version’ the functionality was implemented in

a simpler fashion, containing only the functionality that was required, as opposed to the (proposed) added flexibility of the design pattern. As such, the authors explicitly research whether the implementation of design patterns can be beneficial, even if not needed for the required functionality (in other words: 'overkill' [109]).

Their experiment spanned two days, starting with a 'pre-test', where every participant completed tasks for two different systems (one pattern version, one alternative version). After the first test, the participants received extensive training about design patterns. Finally, the experiment ended with a 'post-test', where every participant received the two systems they have not seen yet (again one pattern version, one alternative version). The tasks had to be fulfilled on paper and contained a mixture of coding and comprehension tasks. The participants were all professional software engineers. Performance was measured in the time it took to finish the tasks, as a measure of understandability and maintainability.

The authors found that the Observer pattern decreased performance overall, the Visitor and Abstract Factory patterns had little effect and the Decorator pattern improved performance. The results indicate that implementing some patterns might be positive for understandability, while others might be counterproductive.

However, in a replication by Vokáč et al., not all results could be verified [133]. The study was a close replication aiming to increase realism. Thus, the authors performed the same experiment with the same set-up, but having the participants work on computers instead of pen and paper. Their results did not agree on every aspect. Most notably, the effects they measured for the Visitor and Observer pattern were exactly opposite to that of the original study.

Overall, the results over various studies have been shown to be inconsistent. This is best illustrated in Table 14 in the mapping study by Ampatzoglou et al. [9]. This table shows that for multiple combinations of quality attributes with design patterns, inconsistent and contradicting results between studies have been found. This effect is especially visible in the measured effect on understandability, where only one of the six design patterns studied by more than one paper had agreeing results in all papers.

Analytic research suggests that this inconsistency might be explained by deeper underlying trade-offs in the situations where design patterns are employed [61, 60, 8, 27]. These trade-offs are calculated as a function of the complexity of the (proposed) implementation of the pattern or its alternative. However, these studies employ metrics as a measure for understandability, whereas we argued in Chapter 2 that metrics might give an inaccurate view of understandability. Unfortunately, to our knowledge no empirical study exists that checks the results of these studies.

Krein et al. performed a large replication of the experiments mentioned above ([109, 133]), where they performed the original experiments with students, through a web interface at four different sites [71]. Using the measurements from the original experiments and their own new measurements, they were able to explain the contradictory results of the original experiments, and the contradictory results of their experiments with the original ones. In particular, they showed that developer experience and pattern knowledge both enhance the benefits of design patterns during maintenance. Taking these variables into account, the

authors were able to generalize across all three studies. In addition, they also found evidence of motivation being a large influencing factor, but could not prove it due to a lack of data.

7.3 Clean Code

In 2009 Martin published his book on Clean Code, describing what he thinks good code should look like. The book is discussed in more detail in Chapter 2. While Clean Code has seen significant adoption in practice, research of the effects of applying Clean Code is limited.

Vasileva and Schmedding presented a case study where the authors introduce the notion of code quality in a course for bachelor students [131]. In this course the students have to develop a small system (about 6000 LoC). The authors created a metrics-based quality model based on selected rules from Clean Code. Vasileva and Schmedding show the results of three iterations of the course. In the first iteration, the tools have not been introduced to the students, serving as a baseline. In the second iteration, the concept and the tools were introduced to the students at the start of the project. In the third iteration, the tutors were instructed to discuss code quality regularly and an incentive was introduced to motivate the students to improve their code.

In the second iteration the code quality did not improve. The authors suggest the focus of the students was not in the code quality and was therefore checked at the end of the project as an afterthought, 'when it was already too late for refactoring' [131]. In the third iteration the scores were notably higher. This suggests that even inexperienced programmers can improve the quality of their code with good guidance and tool support. However, as argued before, metrics do not paint a complete picture of code quality. In addition, some refactorings were performed by the students because the tools said they should, even though they criticized the result. This raises the question whether an improvement not recognized by the authors/maintainers is actually an improvement.

Two previous master theses were found to be very similar to our research [6, 69]. Ammerlaan took a large, legacy, industrial code base and made refactorings of varying size in it [6, 7]. These refactorings were made with Clean Code [82] as a basis. Afterwards, an experiment with 30 professional developers that work in the system on a daily basis was conducted in order to measure the effect of these refactorings on code understandability. The findings were inconclusive: the expected increase in understandability could not be measured, even measuring a decrease in some instances.

Ammerlaan gives a few possible explanations for their results [6]. The author suggests that the lack of effect is mainly due to the habits of the developers, arguing that the developers were used to reading longer procedural methods and might not be used to the workflow of jumping between smaller methods. In addition, they suggest that splitting up methods does not increase understandability per se:

> The flow of method arguments and return values might have been more difficult to understand than a linear flow in a large method. [7]

Koller adopted a very similar approach, where they refactored a single class of an industrial code base [69], also with Clean Code as a basis. An experiment was performed with 14 graduate students and 4 professional developers, again in order to measure the effect of the refactorings on code understandability. The results were similarly inconclusive. The author argues the effect can be explained due the inexperience of their participants, both with larger code bases and the style of Clean Code.

This experiment has been set up so that it builds on the previous research and can thus verify and build on the conclusions drawn. Some key differences are:

- Both studies mention the lack of a realistic working environment as a threat to validity [6, 69]. In this study, the working environment was kept as realistic as possible. However, as mentioned in Section 6.4, the working environment of this experiment was not perfectly realistic either, in particular due to being held in the evening.

- The participants of this experiment do not have the habit of working with procedural code, as was the case in the study by Ammerlaan [6]. Instead, all participants in this experiment were trained in object-oriented programming and most participants practice this daily.

- The participants of this experiment are all professional developers with varying experience with both larger systems and Clean Code. This is in contract with Koller's study, in which the negative results are partially attributed to the lack of experience of the participants.

- While in Ammerlaan's study all developers work on the same project the experiment was performed in [6], in this experiment none of the developers have worked with the code base before. This was also explicitly mentioned by Ammerlaan as a recommendation for future work. Koller does not mention whether they introduced the system explicitly beforehand.

- In both studies, the code bases on which the experiments were performed were criticized on their cleanness and code quality, with both being characterized as legacy code [43, 6, 69]. In this experiment, the code base was considered to be generally clean and can not be characterized as legacy code (see also Section 4.1). While in all studies the refactorings were based on unclean code being refactored with Clean Code and are therefore expected to be similar, it may mean that the difference between both the original and the refactored code is smaller in this study. Additionally, the quality issues solved in the other studies may be more indicative of the code bases as a whole, while in this study the pieces of code with quality issues can be regarded as the exception.

- In the study by Ammerlaan, unit tests are concluded to have a positive impact on productivity: "this study has modestly demonstrated that unit testing can have a positive effect on productivity." [6]. However, in this study, the participants using the unit tests generally took longer to solve the tasks (see also Section 6.2.2). At a glance, we

cannot say what factors caused the improvement in one study, and the decrease in the other. This is an interesting opportunity for further research.

Both studies by Ammerlaan and Koller had both positive and negative effects on productivity and were unable to conclude that refactoring to Clean Code caused immediate benefits. By contrast, this study had no statistically significant negative results, and a general indication for increased performance in the group working in the refactored code was found.

At this stage, it is not clear what factors caused the difference in result. The results do provide an indication that possible explanations for the effects given by Ammerlaan and Koller (e.g., lack of experience, different habits) do not hold in general.

7.4 Conclusions

The mixed and sometimes contradicting results of the works mentioned above makes it challenging to summarize and draw general conclusions about the effect of refactoring. However, some recurring effects can be distinguished:

- When the refactoring directly supports a change task, the effect is positive [97, 135, 69].
- The effect of refactoring is more positive when executed earlier in the project phase [108, 95].
- Refactoring to more extendable but more complicated patterns may harm maintainability when the maintainers are not familiar with the patterns [71, 6].

Chapter 8

Conclusions and Future Work

The goal of this study was to assess the effects of two solutions on the immediate productivity of a developer in a new code base: refactoring following the principles of Clean Code and providing an introductory document to the code. This was done using a controlled experiment, where the participants were split into four groups, each being provided by either the original or refactored code, and either receiving the introductory document beforehand or not. In the experiment, the participants completed small coding tasks and answered multiple choice questions about the functionality of the code. Their completion times and correctness percentages were used as measures for understandability.

This chapter will reflect on the results of this experiment and draw conclusions. Finally, some ideas for future work will be discussed.

8.1 Conclusions

Before drawing our conclusions, we first recap the research questions and the answers given in Section 6.6.

RQ1 is about investigating the relationship between refactoring and understandability. It was stated as follows:

> Does refactoring according to the guidelines of Clean Code have a positive effect on understandability?

As discussed in Section 6.6.1, the results give a good indication that refactoring had a positive effect on understandability.

However, this conclusion comes with a lot of caveats (more extensively discussed in Chapter 6). While the combined data of all assignments shows a positive significant effect, only 4 out of the 13 individual assignments gave a significant positive effect in either completion times or correctness, with the other 9 assignments giving no significant effect in one way or the other. The effect of refactoring on understandability can be influenced by many factors, including the extent and quality of the refactoring, the quality of the code before refactoring, the presence and quality of unit tests, the experience of the participants and the

type and difficulty of the task to be performed. The extent of the influence of these factors is also not agreed on across different studies, as seen in Section 7.3.

For these reasons, we cannot conclude that refactoring according to the guidelines of Clean Code gives an immediate positive effect on understandability *in general*. Further research is needed to determine in which situations refactoring is beneficial, and in which refactoring is not worth it or even detrimental. This research does give a positive indication for the existence of such beneficial situations.

RQ2 is about investigating the relationship between the introductory document and understandability. It was stated as follows:

> Does adding project knowledge in the form of an introductory document have a positive effect on understandability?

In Section 6.6.2, we discussed that no effects of the document were seen in the results. Although again 4 out of the 13 individual assignments show a significant positive effect in either completion times or correctness, the combined data does not show this positive effect. Additionally, the participants that received the document generally did not believe that the document helped them during the experiment. A possible explanation for this can be found in the contents of the document: the document gave a general overview of the code base, while the assignments in the experiment were focused on specific parts of the code.

The lack of effect, in addition with the observations that extensive documentation is often quickly outdated [73] and discouraged by the Agile methodology [35], we do not recommend drafting an introductory document to increase the understandability of a system. However, this does not mean writing such a document is useless: it may have positive effects on constructs not measured in this study, such as maintainability. Further research may reveal better ways of using documentation to improve the understandability of a system.

RQ3 is about the interaction between the refactoring and the document. It was stated as follows:

> Does both refactoring and adding project knowledge have a positive effect on understandability?

Unfortunately, as discussed in Section 6.6.3, little can be said about the interaction between the document and the refactoring treatments, because there is not enough data to properly investigate this. However, with the lack of observed effect from the document treatment, we deem it unlikely that this interaction is significant, as the hypothesized causes for the interaction require a positive effect of the introductory document.

The aim of this research is to assess methods which ALTEN can use to aid developers in understanding a new project. This study gives an indication that refactoring according to the guidelines of Clean Code can give an improvement in understandability. In practical terms, this means that an explicit ‘refactoring phase’, performed by the existing developers before the new developers join, may be successful in improving the onboarding of said new developers. However, this study is not able to provide specific recommendations of when this phase is likely to be successful, and when it is not. This is caused by the large amount

of possible factors influencing the effectiveness of the refactoring. Herein lie opportunities for further research.

Additionally, no positive effect on understandability, of drafting and providing an introductory document for this purpose, was found. This means we do not recommend writing such a document for the sole purpose increasing understandability. Further research may find other forms of documentation that are better suited for this.

8.2 Future work

The open-ended conclusions leaves plenty of opportunity for future research. Here, we list some recommendations for possible directions of future work.

- Similar studies to this one, such as the studies performed by Ammerlaan and Koller, give conflicting results [6, 69]. This effect is visible across the entire body of work on refactoring, as discussed in Chapter 7. Future work could research the similarities and differences between different studies, in order to find the specific factors that cause some refactorings to fail to give the expected effects, and some to succeed.
- As described in Chapter 2, metrics are currently unable to give satisfactory measurements of understandability, with promising progress being made with textual metrics. Further work in measuring understandability using textual metrics could help improve our understanding of understandability and shine light on the most important factors that make up understandability.
- In Section 6.2.2, the impact of various solving strategies is discussed. Future studies can research the various strategies developers may take in solving coding tasks, and how this impacts their productivity.
- This study unsuccessfully used an introductory document to aid new developers starting on new projects. Future research could study the specific information needs to new developers and draft a framework on how this information can be best provided.
- In this study only the short term impact of the treatments was measured. A logical extension is to measure the impact on the longer term. In particular, this allows for measurement of other factors such as maintainability and testability.

Bibliography

[1] Krishan K Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. An integrated measure of software maintainability. In *Reliability and maintainability symposium, 2002. Proceedings. Annual*, pages 235–241. IEEE, 2002.

[2] Rafa E Al-Qutaish. Quality models in software engineering literature: an analytical and comparative study. *Journal of American Science*, 6(3):166–175, 2010.

[3] Ghazi Alkhatib. The maintenance problem of application software: An empirical analysis. *Journal of Software Maintenance: Research and Practice*, 4(2):83–104, 1992.

[4] Eric Allman. Managing technical debt. *Communications of the ACM*, 55(5):50–55, 2012.

[5] Sven Amann, Sebastian Proksch, and Sarah Nadi. Feedbag: an interaction tracker for visual studio. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–3. IEEE, 2016.

[6] Erik Ammerlaan. Effects of refactoring on productivity in relation to code understandability. Master's thesis, Delft University of Technology, 2014.

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

[8] Apostolos Ampatzoglou, Georgia Frantzeskou, and Ioannis Stamelos. A methodology to assess the impact of design patterns on software quality. *Information and Software Technology*, 54(4):331–346, 2012.

[9] Apostolos Ampatzoglou, Sofia Charalampidou, and Ioannis Stamelos. Research state of the art on gof design patterns: A mapping study. *Journal of Systems and Software*, 86(7):1945–1964, 2013.

[10] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology*, 64:52–73, 2015.

[11] Bente CD Anda, Dag IK Sjøberg, and Audris Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, 35(3):407–429, 2009.

[12] Elvira Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Matthias Galster, and Paris Avgeriou. A mapping study on design-time quality attributes and metrics. *Journal of Systems and Software*, 127:52–77, 2017.

[13] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, 2012.

[14] Marc Bartsch and Rachel Harrison. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Journal*, 16(1):23–44, 2008.

[15] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering*, pages 1–37, 2018.

[16] Hans Christian Benestad, Bente Anda, and Erik Arisholm. Assessing software product maintainability based on class-level structural measures. In *International Conference on Product Focused Software Process Improvement*, pages 94–111. Springer, 2006.

[17] Dennis Bijlsma, Miguel Alexandre Ferreira, Bart Luijten, and Joost Visser. Faster issue resolution with higher technical quality of software. *Software quality journal*, 20(2):265–285, 2012.

[18] Barry W Boehm. Improving software productivity. In *IEEE Computer*, pages 43–57, September 1987.

[19] Barry W Boehm and Philip N. Papaccio. Understanding and controlling software costs. *IEEE transactions on software engineering*, 14(10):1462–1477, 1988.

[20] Barry W Boehm, John R Brown, and Mlity Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976.

[21] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. "I know it when I see it": Perceptions of code quality. In *ACM Conference on Innovation and Technology in Computer Science Education, Bologna*, pages 389–399. ACM Digital Library, 2017.

[22] Lionel C Briand and Jürgen Wüst. Empirical studies of quality models in object-oriented systems. In *Advances in computers*, volume 56, pages 97–166. Elsevier, 2002.

[23] F Brooks. No silver bullet: Essence and accidents of software engineerin. *Computer*, 20(4):10–19, 1987.

[24] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[25] Oscar Callau and Eric Tanter. Programming with ghosts. *IEEE software*, 30(1): 74–80, 2012.

[26] Andrea Capiluppi, Maurizio Morisio, and Patricia Lago. Evolution of understandability in oss projects. *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, 2004.

[27] Sofia Charalampidou, Apostolus Ampatzoglou, Paris Avgeriou, Seren Sencer, Elvira-Maria Arvanitou, and Ioannis Stamelos. A theoretical model for capturing the impact of design patterns on quality: the decorator case study. In *Proceedings of the Symposium on Applied Computing*, pages 1231–1238. ACM, 2017.

[28] Quentin Charatan and Aaron Kans. *Java in two semesters*. McGraw-Hill Education, 2nd edition, 2009.

[29] Robert N. Charette. Why software fails [software failure]. *IEEE Spectrum*, 42(9): 42–49, Sept 2005. ISSN 0018-9235. doi: 10.1109/MSPEC.2005.1502528.

[30] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[31] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[32] Campbell D.T Cook, T.D. *Quasi-experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company, Boston, 1979.

[33] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[34] Ward Cunningham. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[35] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75. ACM, 2005.

[36] Florian Deissenboeck, Elmar Juergens, Klaus Lochmann, and Stefan Wagner. Software quality models: Purposes, usage scenarios and requirements. In *Software Quality, 2009. WOSQ'09. ICSE Workshop on*, pages 9–14. IEEE, 2009.

[37] Salih Demir, James H Garrett Jr, Burcu Akinci, Omer Akin, and Mark Palmer. A semantic web-based approach for representing and reasoning with vocabulary for computer based standards processing. In *Proceedings of the 2010 International Conference on Computing in Civil and Building Engineering (ICCCBE), Nottingham*, 2010.

[38] R. Geoff Dromey. A model for software product quality. *IEEE Transactions on software engineering*, 21(2):146–162, 1995.

[39] Kasper Edwards. When beggers become choosers. *First Monday*, 5(10), 2000.

[40] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[41] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[42] Len Erlikh. Leveraging legacy system dollars for e-business. *IT professional*, 2(3): 17–23, 2000.

[43] Michael Feathers. *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[44] Norman E Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149–157, 1999.

[45] Charles Fishman. They write the right stuff. *Fast Company*, 6(95):1997, 1996.

[46] RK Fjeldstad and WT Hamlen. Application program maintenance study - reports to our respondents. *Proc. Guide*, 1982.

[47] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1st edition, 1999.

[48] Martin Fowler. Codesmell, 2006. URL `https://martinfowler.com/bliki/CodeSmell.html`. Accessed May 1, 2019.

[49] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2nd edition, 2018.

[50] Davide Fucci, Burak Turhan, and Markku Oivo. On the effects of programming and testing skills on external quality and productivity in a test-driven development context. In *Proceedings of the 19th international conference on evaluation and assessment in software engineering*, page 25. ACM, 2015.

[51] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[52] Golara Garousi, Vahid Garousi-Yusifoğlu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57:664–682, 2015.

[53] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and software Technology*, 46(5):337–342, 2004.

[54] Yossi Gil and Gal Lalouche. On the correlation between size and metric validity. *Empirical Software Engineering*, 22(5):2585–2611, 2017.

[55] Robert B Grady. *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc., 1992.

[56] Maurice H. Halstead. *Elements of Software Science*. Elsevier Science Inc., New York, NY, USA, 1977. ISBN 0444002057.

[57] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[58] Jim Robert Highsmith. *Agile project management: creating innovative products*. Pearson Education, 2nd edition, 2009.

[59] Myles Hollander, Douglas A Wolfe, and Eric Chicken. *Nonparametric statistical methods*, volume 751. John Wiley & Sons, 3rd edition, 2013.

[60] Nien-Lin Hsueh, Peng-Hua Chu, and William Chu. A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software*, 81(8): 1430–1439, 2008.

[61] Brian Huston. The effects of design pattern application on metric scores. *Journal of Systems and Software*, 58(3):261–269, 2001.

[62] ISO/IEC25010:2011. Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models. Standard, International Organization for Standardization, 2011.

[63] ISO/IEC9126-1:2001. Software engineering – product quality – part 1: Quality model. Standard, International Organization for Standardization, 2001.

[64] Ronald Jabangwe, Jürgen Börstler, Darja Šmite, and Claes Wohlin. Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. *Empirical Software Engineering*, 20(3):640–693, 2015.

[65] Paul Jansen. The TIOBE Quality Indicator: A pragmatic way of measuring code quality, 2012. URL `https://www.tiobe.com/content/paperinfo/TIOBEQualityIndicator.pdf`.

[66] Capers Jones. The economics of software maintenance in the twenty first century. *Unpublished manuscript.*, 2006.

[67] Stephen H Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[68] Joshua Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[69] Henning Grimeland Koller. Effects of clean code on understandability: An experiment and analysis. Master's thesis, University of Oslo, 2016.

[70] Phil Koopman. A case study of Toyota unintended acceleration and software safety, 2014. Presentation.

[71] Jonathan L Krein, Lutz Prechelt, Natalia Juristo, Aziz Nanthaamornphong, Jeffrey C Carver, Sira Vegas, Charles D Knutson, Kevin D Seppi, and Dennis L Eggett. A multi-site joint replication of a design patterns experiment using moderator variables to generalize across contexts. *IEEE Transactions on Software Engineering*, 42(4): 302–321, 2016.

[72] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Pearson Education India, 1998.

[73] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

[74] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[75] Nancy G Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[76] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[77] Bennet P Lientz and E Burton Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, 1981.

[78] Jin-cherng Lin and Kuo-chiang Wu. A model for measuring software understandability. In *Computer and Information Technology, 2006. CIT'06. The Sixth IEEE International Conference on*, pages 192–192. IEEE, 2006.

[79] Jin-Cherng Lin and Kuo-Chiang Wu. Evaluation of software understandability based on fuzzy matrix. In *Fuzzy Systems, 2008. FUZZ-IEEE 2008.(IEEE World Congress on Computational Intelligence). IEEE International Conference on*, pages 887–892. IEEE, 2008.

[80] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):31, 2014.

[81] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2003.

[82] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[83] Robert C Martin. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017.

[84] Robert C Martin and Micah Martin. *Agile principles, patterns, and practices in C#*. Pearson Education, 2006.

[85] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 2(4):308–320, 1976.

[86] Jim A McCall. Factors in software quality. *US Rome Air development center reports*, 1977.

[87] Steve McConnell. *Code complete*. Microsoft Press, 2nd edition, 2004.

[88] Federio Mena Quintero, Miguel de Icaza, and Morten Welinder. Gnome programming guidelines, 1999. URL `http://web.archive.org/web/20000302060239/http://developer.gnome.org/doc/guides/programming-guidelines/good-code.html`. Accessed May 1, 2019.

[89] T. Mens and T. Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126 – 139, feb 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.1265817.

[90] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. In *Proceedings LDTA'03: Third Workshop on Language Descriptions, Tools and Applications*, 2003.

[91] Microsoft. Maintainability index range and meaning, 2007. URL https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/. Accessed May 1, 2019.

[92] Microsoft. Extension methods (C# programming guide), 2015. URL https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods. Accessed May 1, 2019.

[93] Microsoft. Code metrics values, 2018. URL https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values. Accessed May 1, 2019.

[94] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer: an investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35. IEEE Press, 2015.

[95] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer, 2008.

[96] Meiyappan Nagappan and Emad Shihab. Future trends in software engineering research for mobile apps. In *Software analysis, evolution, and reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 5, pages 21–32. IEEE, 2016.

[97] TH Ng, Shing Chi Cheung, Wing-Kwong Chan, and Yuen-Tak Yu. Work experience versus refactoring to design patterns: a controlled experiment. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 12–22. ACM, 2006.

[98] John T Nosek and Prashant Palvia. Software maintenance management: changes in the last decade. *Journal of Software Maintenance: Research and Practice*, 2(3): 157–174, 1990.

[99] Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017.

[100] Marcio FS Oliveira, Ricardo Miotto Redin, Luigi Carro, Luís da Cunha Lamb, and Flávio Rech Wagner. Software quality metrics and their impact on embedded software. In *Model-based Methodologies for Pervasive and Embedded Software, 2008. MOMPES 2008. 5th International Workshop on*, pages 68–77. IEEE, 2008.

[101] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344. IEEE, 1992.

[102] Merriam Webster Online. Understand, 2019. URL https://www.merriam-webster.com/dictionary/understand. Accessed May 1, 2019.

[103] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for smell detection. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[104] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, 44(10):977–1000, 2018.

[105] Thomas Panas, Dan Quinlan, and Richard Vuduc. Tool support for inspecting the code quality of HPC applications. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications*, page 2. IEEE Computer Society, 2007.

[106] David Lorge Parnas. Software aging. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 279–287. IEEE, 1994.

[107] James W Paulson, Giancarlo Succi, and Armin Eberlein. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, 30(4):246–256, 2004.

[108] Markus Pizka. Straightening spaghetti-code with refactoring? In *Software Engineering Research and Practice*, pages 846–852, 2004.

[109] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brossler, and Lawrence G. Votta. A controlled experiment in maintenance: comparing design patterns to simpler solutions. *IEEE transactions on Software Engineering*, 27(12):1134–1144, 2001.

[110] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019. URL https://www.R-project.org/.

[111] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[112] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377. IEEE Computer Society, 2009.

[113] Martin P Robillard, Wesley Coelho, and Gail C Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on software engineering*, 30(12):889–903, 2004.

[114] Eran Rubin and Hillel Rubin. Supporting agile software development through active documentation. *Requirements Engineering*, 16(2):117–132, 2011.

[115] Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[116] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: how far are we? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 417–427. IEEE Press, 2017.

[117] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018.

[118] E. C Sharp, L. G Pelletier, and C. Lévesque. The double-edged sword of rewards for participation in psychology experiments. *Canadian Journal of Behavioural Science/Revue canadienne des sciences du comportement*, 38(3):269, 2006.

[119] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2008.

[120] Dag IK Sjøberg, Bente Anda, and Audris Mockus. Questioning software maintenance metrics: a comparative case study. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 107–110. ACM, 2012.

[121] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[122] D Srinivasulu, Adepu Sridhar, and Durga Prasad Mohapatra. Evaluation of software understandability using rough sets. In *Intelligent Computing, Networking, and Informatics*, pages 939–946. Springer, 2014.

[123] Bob Stein. A point about polygons. *Linux Journal*, 1997(35es):14, 1997.

[124] Maikel Steneker. Towards an empirical validation of the TIOBE Quality Indicator. Master's thesis, Eindhoven University of Technology, 2016.

[125] Jamie Stevenson and Murray Wood. Recognising object-oriented software design quality: A practitioner-based questionnaire survey. *Software Quality Journal*, 26(2): 321–365, June 2018.

[126] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.

[127] StyleCopAnalyzers. Sa1124: Donotuseregions, 2015. URL `https://github.com/DotNetAnalyzers/StyleCopAnalyzers/blob/master/documentation/SA1124.md`. Accessed: May 1, 2019.

[128] Edith Tom, AybüKe Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[129] Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. "automatically assessing code understandability" reanalyzed: Combined metrics matter. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 314–318, 2018.

[130] Arie van Deursen. Program comprehension risks and opportunities in extreme programming. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 176–185. IEEE, 2001.

[131] Anna Vasileva and Doris Schmedding. How to improve code quality by measurement and refactoring. In *Quality of Information and Communications Technology (QUATIC), 2016 10th International Conference on the*, pages 131–136. IEEE, 2016.

[132] Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. O'Reilly Media, Inc., 2016.

[133] Marek Vokáč, Walter Tichy, Dag IK Sjøberg, Erik Arisholm, and Magne Aldrin. A controlled experiment comparing the maintainability of programs designed with and without design patternsa replication in a real programming environment. *Empirical Software Engineering*, 9(3):149–195, 2004.

[134] Anneliese von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[135] Michael Wahler, Uwe Drofenik, and Will Snipes. Improving code maintainability: A case study on the impact of refactoring. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 493–501. IEEE, 2016.

[136] Kurt D Welker. The software maintainability index revisited. *CrossTalk*, 14:18–21, 2001.

[137] Dirk Wilking, Umar Farooq Kahn, and Stefan Kowalewski. An empirical evaluation of refactoring. *e-Informatica*, 1(1):27–42, 2007.

[138] Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical research methods in software engineering. In *Empirical methods and studies in software engineering*, pages 7–23. Springer, 2003.

[139] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[140] Cheng Zhang and David Budgen. What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering*, 38(5):1213–1231, 2012.

[141] Junji Zhi, Vahid Garousi-Yusifoğlu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175–198, 2015.

[142] Yuming Zhou, Hareton Leung, and Baowen Xu. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. *IEEE Transactions on Software Engineering*, 35(5):607–623, 2009.

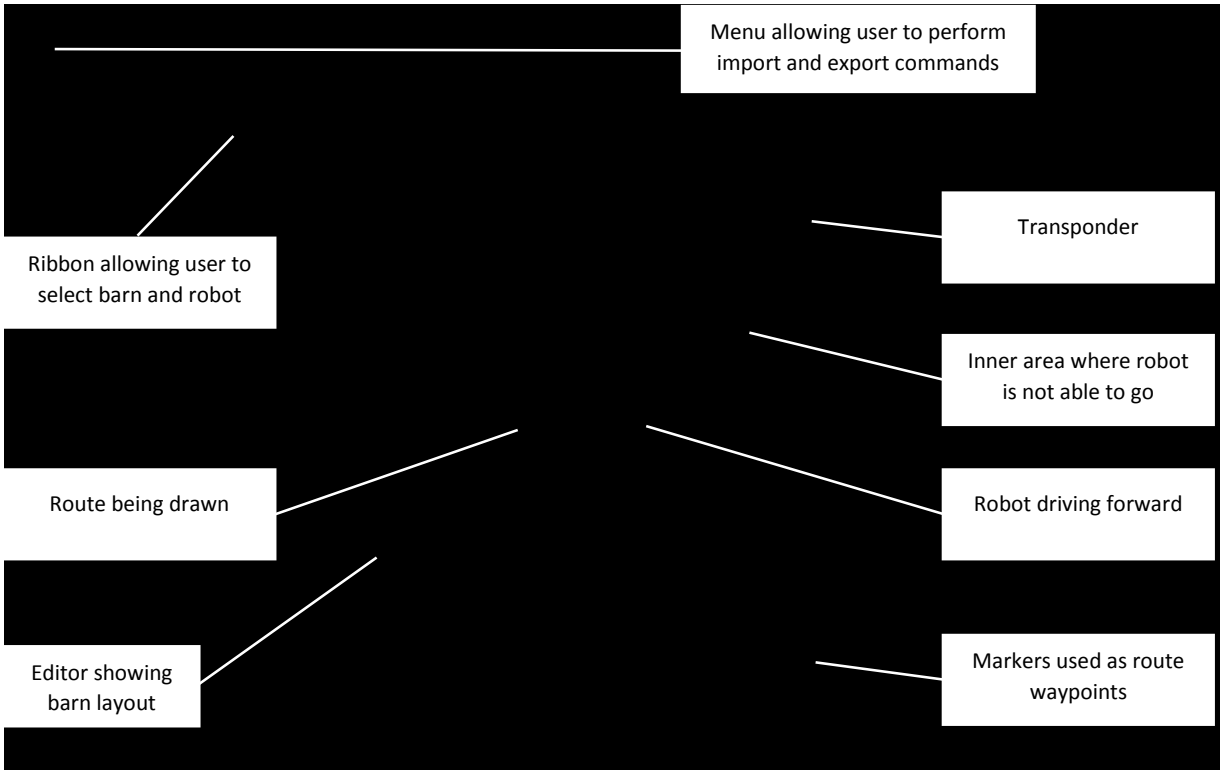
[143] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

Appendix A

Project knowledge document

In this appendix the project knowledge document is presented, as sent to the participants. Some sensitive information has been redacted.

[redacted] is a software tool for farmers, allowing them to draw the routes their AGVs (automated guided vehicles) should follow in their barns. These AGVs might for example be slurry robots, driving through the barn to clean the floor of manure.



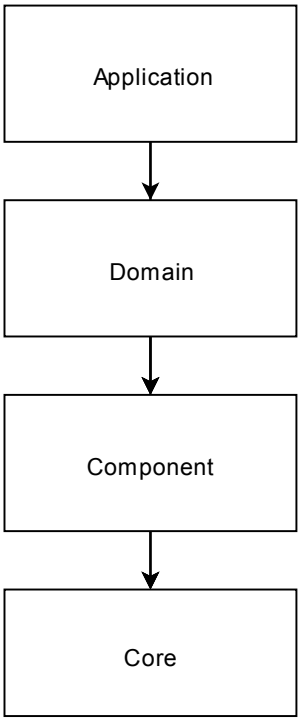
In the software, the user is able to load in a floor plan, specified in XML. The layout of the barn is then shown, after which the user is able to draw routes for the robots belonging to the barn. After the user is done planning out the routes, they are able to export these routes to an XML file. This file can be read by the robots, allowing them to drive the route in the real world.

GUI

The GUI of the application is built using Windows Presentation Foundation (WPF), using packages MahApps.Metro and Fluent.Ribbon for the metro- and ribbon-themed interface. The interface consists of two main parts:

- The ribbon on top allows for switching between loaded barns and switching between robots within a barn.
- The main area of the interface is where the layout of the barn is drawn and where the user can draw the routes for the robots. Drawing a route is done by first clicking anywhere in the layout to start the route, followed by drafting a path between the placed markers and finally ending at the starting point.

Architecture



The code is split into multiple projects with four main namespaces, in order of abstraction: Application, Domain, Component, Core. Dependencies are always from specific to abstract. For example, project in Domain can depend on projects in both the Component and Core namespaces, but projects in Core should not depend on any project in another namespace. An extensive overview of the project dependencies can be seen on the final page in this document. An arrow from A to B means that project A depends on project B. Note that in this overview the dependencies are transitive for simplicity. For example, the connection between Application.Main and Core.IO is not drawn, because an indirect connection (through Domain.IO) is already drawn.

The **Application** namespace is the entry point of the application, containing the main view and its model. It is responsible for initializing the application and managing the various GUI components.

It contains two projects: Application.Main and Application.Version. The Application.Version project contains no functionality and serves as an artifact for the build process. The Application.Main contains the functionality as described above: it defines the main GUI style and layout and manages the application.

The **Component** namespace contains two projects that should have general purpose (i.e., not specific to this application). The Component.Ribbon defines some extensions to the Fluent.Ribbon package used to the ribbon layout. The Component.SpatialCalculus has its origins in another project and contains geometrical calculations.

The application business logic is contained in the **Domain** and **Core** namespaces. The Core namespace contains functionality that is independent of the project domain, while the Domain namespace contains domain specific functionality. This makes the Core namespace easily re-used for other applications. As an example, the Core.Data project contains classes such as Point2D and Line, while the Domain.Data project contains classes such as Barn and Transponder.

The Core namespace contains four projects:

- Core.Controls contains general utility functionality for use in the WPF-code: command classes, logic for file dialogs and WPF ValueConverter classes.
- Core.Data contains a general base to work on, mostly geometry-related functionality.
- Core.IO contains general utility classes for doing IO.
- Core.Utils contains two utility classes for assemblies and files.

The Domain namespace contains four projects:

- Domain.Controls contains specific command classes for the GUI and defines all shapes that can be drawn in the application.

- Domain.Data contains the application business logic, with classes to construct and represent the various domain objects.
- Domain.IO contains the import and export functionality of layout and routes, respectively.
- Domain.Wpf contains the definitions in XAML of the various GUI components, along with related functionality. It also contains factory classes to construct the shapes drawn in the GUI.

Typical execution flow

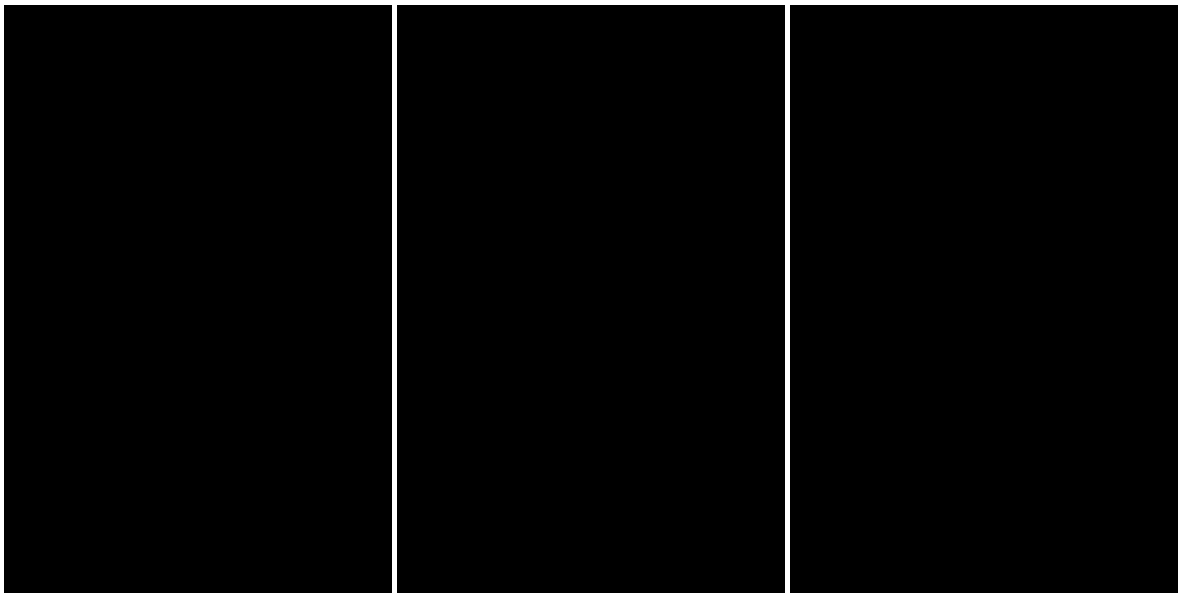
The entry point of the application is the MainWindow.xaml in the Application.Main project, which in turn constructs the MainViewModel class. This class is responsible for managing the GUI components and handling of user events, such as importing and exporting files.

The user can import a barn layout. This is handled by the BarnImporter class in the Domain.IO project. This class reads in the XML file containing the layout, deserializing it into surrogate classes containing the data. From these surrogate classes, the actual domain objects are constructed.

The barn is represented by the Barn class in Domain.Data. A Barn consists primarily of a barn layout (represented by the BarnLayout class) and a collection of robots (represented by the IRobot interface).

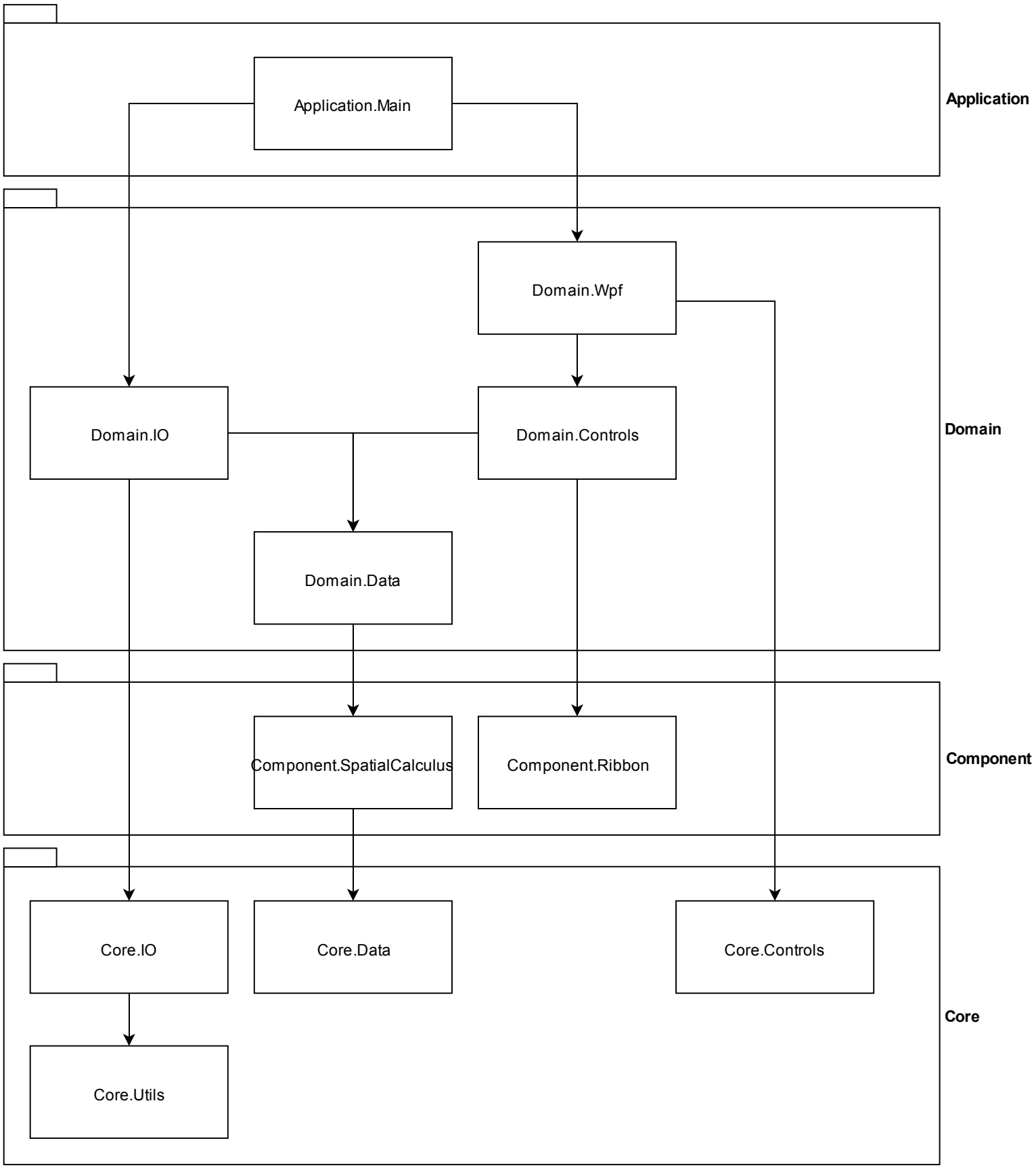
The most complicated part of constructing the Barn lies in constructing the BarnLayout. The XML input specifies the layout only by providing the coordinates at which walls are located, but for use in the application it is necessary to group the walls into sets that enclose an area. When the collection of walls in the BarnLayout is updated, an algorithm finds the contours and creates polygon areas from this (Area class in Domain.Data). Additionally, distinction is made between the outer most area (representing the borders of the barn, where no activity can take place outside of) and the inner areas (enclosures inside the barn where the robots are not able to go). Aside from the areas, a BarnLayout also contains a list of transponders: devices placed in the floor of the barn to help robot navigation. Some transponders may also serve as home transponders: the home base for a robot where they must start and end their routes.

The robots are only allowed to travel horizontally and vertically in the layout, apart from the first and final part of the route: moving to and from the home transponder. To facilitate the drawing of routes, action markers are placed in the application. The generation of these markers is performed in three steps. First, markers are placed around every corner in the walls, making sure they are exactly half the width of the robot's scraper away from the wall horizontally and vertically. This is done to allow the robots to drive exactly around them, and to allow the robot to follow a wall exactly. Secondly, extra markers are drawn so that all markers placed in the first step are reachable from one another with only horizontal and vertical movement. Thirdly, gaps between markers are filled for more freedom of movement. An example of this generation is shown below.



All drawing is facilitated by a slightly modified copy of the OxyPlot library, referenced in the application as Component.Editor and Component.Editor.Wpf. Every domain object that should be drawn has a corresponding Drawable class (e.g., DrawableWall, DrawableRoute) and are constructed using factory classes (e.g., DrawableWallFactory, DrawableRouteFactory). The factory classes are generally responsible for selecting the appropriate objects from the Barn and defining the style. The Drawable classes inherit from the corresponding shapes from the Component.Editor. Their functionality ranges from simply defining the translation between the location of the domain object and the points the shape should consist of (e.g. DrawableWall) to containing extensive functionality to control validation and drawing (e.g. DrawableRoute).

Whenever the user has drawn one or more routes, the corresponding route data can then be exported to another XML file. This is handled by the BarnExporter class in the Domain.IO project. The domain objects are transformed into surrogate classes containing the relevant information for the export. The surrogates are then serialized to XML and saved to disk.


Application
Application.Main
Domain
Domain.Wpf
Domain.IO
Domain.Controls
Domain.Data
Component
Component.SpatialCalculus
Component.Ribbon
Core
Core.IO
Core.Data
Core.Controls
Core.Utils