Effects of Refactoring on Productivity in Relation to Code Understandability

A series of controlled experiments

Erik Ammerlaan

Effects of Refactoring on Productivity in Relation to Code Understandability

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Erik Ammerlaan
born in Rozenburg ZH, the Netherlands

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

Exact International Development B.V.
Molengraaffsingel 33
Delft, the Netherlands
www.exact.com

Cover picture: The Merry Family, Jan Steen, 1668. Rijksmuseum, Amsterdam.

Effects of Refactoring on Productivity in Relation to Code Understandability

Author: Erik Ammerlaan
Student id: 4005341
Email: e.ammerlaan@student.tudelft.nl

Abstract

Depending on the context, the benefits of clean code with respect to understandability might be less plain in the short term than is often claimed. This work has studied a system with legacy code in an industrial environment to evaluate if giving 'clean code' to developers would immediately lead to increases in productivity. They were given refactored components and were assigned small coding tasks to complete. Contrary to our expectations, we observed both increases as well as decreases in understandability, showing that immediate increases in understandability are not always obvious. This study suggests that negative effects could have been caused by the fact that the test subjects were used to long methods rather than a decomposed design. Another finding is that unit tests, accompanying refactorings, can lead to more substantial increases in productivity. Secondly, developers tend to implement higher quality solutions when working with clean code. A recommendation to improve the net return on refactoring is to not just refactor to improve its understandability, unless one has additional motives, such as easing maintenance or increasing testability.

Thesis Committee:

Chair:	dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	ing. W. Veninga, Exact International Development B.V.
Committee Member:	dr. ir. F.F.J. Hermans, Faculty EEMCS, TU Delft
Committee Member:	dr. M.T.J. Spaan, Faculty EEMCS, TU Delft

Preface

I proudly present my master's thesis, describing my research on refactoring. This research would not have been possible without the help of some that deserve to be acknowledged. First of all, I would like to thank Andy Zaidman for all his feedback and, most notably, the enthusiasm with which he followed my progress. My thanks go to Wim Veninga for the level of freedom that he gave me at Exact and for his helpful criticism that helped me to stay focused. Next to Andy and Wim, I would like to thank the other committee members for reviewing my work. My thanks go to the management at Exact for the unforgettable experience of travelling to Kuala Lumpur, and to Prem Kumar Ponuthorai for the collaboration while setting up a refactoring training over there. Finally, my thanks go to the system team at Exact for letting me feel part of their team; special thanks go to Jerry de Swart for helping me find my way around the company.

This thesis is only the last stage of a journey. Along the way, many have mentored me or inspired me in one way or another and have brought me to where I am now. I cannot name them all, but they should know that they have my thanks. I would like to thank my family, my mother in particular, for their unconditional and indispensable support over the years.

Erik Ammerlaan
Delft, the Netherlands
June 30, 2014

Contents

List of Figures

List of Code Listings

Chapter 1

Introduction

The cover of this work shows a famous painting by Jan Steen, a Dutch painter from the 17th century, who was known for his portrayal of messy interiors. In fact, 'a Jan Steen household' is a common Dutch expression for describing a cluttered interior. It is well-known that many large software systems in our industry also contain such cluttered interiors, if we consider their source code. R.C. Martin [12] even said that "software systems almost always degrade into a mess." It happens because requirements change, because of pressure to deliver, et cetera. Just as Steen's paintings mainly contain merry faces despite their cluttered surroundings, a messy code base does not immediately seem much of a concern. A low code quality can coexist with a high product quality. What matters most in the end is the external quality of the product as perceived by customers. Hence, 'dirty' code is often tolerated, as it seems more profitable to add new features for customers rather than cleaning up. After all, customers will not see the interior, only the developers will.

However, it has been recognized that dirty code can have a significant impact on the software development process, which is best explained by the metaphor of *technical debt*, as illustrated by Ward Cunningham:

> "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, object-oriented or otherwise." [8]

If unclean code is not cleaned up, developers have to pay interest in the form of the additional time it takes to understand the complex code, leave alone changing it. If only new debt is introduced, then the accumulated interest may become so high that changing the code becomes infeasible. As an example of the impact, it is said that RIM, the company behind Blackberry, shipped a tablet without a native e-mail client because their outdated software architecture could not be changed quickly due to accumulated technical debt [9].

The global technical debt was estimated by Gartner to be $500 billion in 2010 [15]. In their press conference they stated how this problem, hidden from sight, is getting bigger

every year and more difficult to deal with every year. While the association with debt was originally metaphorical, technical debt can lead to actual costs. Furthermore, technical debt was found to impact morale, productivity, quality and risk in the long term [46].

This shows the scope and the urgency of the problem, which has motivated this study. Our study focuses on *code quality*. More specifically, we look for ways to manage the technical debt of a so-called legacy system at Exact through refactoring.

1.1 Context

Exact serves entrepreneurial businesses through a number of software solutions and services. Founded in 1984 in Delft, Exact has since become an international company and a global solution provider. This thesis focuses on *Exact Online* (EOL), Exact's online business software, which is hosted in the cloud as a software as a service (SaaS) with a subscription model. The first version of Exact Online was launched in the Netherlands in 2005 and since then, it has been rolled out to Belgium, Germany, the United Kingdom and the United States of America, serving over 140,000 customers.

Since Exact Online is a SaaS solution, there is only one version of the product in production. Hence, customers automatically benefit from new product updates, which are deployed to production multiple times a week. Thus, Exact Online follows a rolling release development model.

The continuous development of Exact Online has led to a large organizational growth. The past year, the number of developers working on Exact Online doubled to 300 people. The number of agile development teams has increased to 20 teams, located in the Netherlands, Belgium, Malaysia and the United States, that practise round-the-clock development [17].

So, Exact Online has become a large and complex system over the years and it continues to grow. The rapid growth of the system and the number of people working on it, the challenges of global software development, and the specific demands of a cloud solution with regards to scalability and performance, are all factors that have had repercussions on the code quality.

1.1.1 Code of Exact Online

Considering the size and the rapid growth of Exact Online, it is not surprising that quite some technical debt has been accumulated over the years. For the same reason, it is also not trivial to reduce this debt. In fact, large parts of Exact Online have become *legacy code*.

Exact Online has mostly been written in VB.NET, an object-oriented programming language. However, the actual use of object-orientation seems limited. For instance, it is hard to find domain entities in the code that resemble entities from the problem domain. The system is more concerned with data than with entities. Also, large classes of more than 500 lines of code are not uncommon, neither are long methods that seem to fit more with a procedural programming style. The term *procedural object-oriented programming* [4], which describes code that lacks an object-oriented design though it is written in an object-oriented language, rightly applies to Exact Online.

The precise definition of legacy code will be explored further in Chapter 2, but its name implies that the code has somehow been inherited. If the current generation of software engineers were to rebuild Exact Online, they would do it differently with today's knowledge. However, a legacy system cannot be replaced or upgraded except at high cost [10]. Furthermore, legacy code is often very *valuable* and not something one wants to throw away [10]. This is certainly true for Exact Online as it provides value for a large number of customers; in fact, there is a growing demand for it. In practice, developers may feel that having a legacy codebase is 'just the way it is' and something that they just have to put up with. However, there are ways to work more effectively with legacy code.

1.1.2 Refactoring and Unit Testing

One of the problems of legacy code is that it is hard to test. At Exact, there is an automatic test suite of high-level functional tests to guarantee product quality. But implementing unit tests is challenging, as Exact Online was not designed for testability. Unit tests can provide quick feedback to developers when they make changes to the source code, so that they can have confidence in their changes.

Refactoring, changing the code without affecting its external behaviour, can be used to break dependencies in the code to increase testability. Feathers has written extensively about how one can work more effectively with legacy code by using refactoring to make software unit testable [12]. But refactoring can be used to achieve much more than only increased testability. Refactoring can be used to clean up complex code, so that it becomes easier to understand and to maintain.

When refactoring becomes a large activity, one can speak of *software re-engineering* [10]. Re-engineering entails more than just restructuring code. The process of re-engineering starts with *reverse engineering* where higher-level models are reconstructed from the code, such as diagrams and other design documents. In other words, before any refactoring activity is started, a thorough analysis is performed to understand the functionality of a component and the rationale of its existence. Next, a new design is proposed (*forward engineering*), after which the actual refactoring work commences.

Refactoring itself, in its broad sense, is claimed to increase developer productivity, to improve the design of software, to make software easier to understand and to help developers to find bugs [13]. There are those who even advocate *continuous refactoring* [19] to prevent software from degrading. Some agile practitioners that apply Scrum recommend that refactoring should be part of Scrum's 'definition of done' [26]. However, most benefits of refactoring are in the long term, which can make it difficult to measure the value of refactoring. As someone at Microsoft said in an interview [20]: "How do you measure the value of a bug that never existed, or the time saved on a later undetermined feature?"

The argument that the structure of code, with respect to development productivity, matters to even the smallest detail has perhaps been put forward most strongly by Robert C. Martin [26] who published 'Clean Code: A Handbook of Agile Software Craftsmanship' in which he calls on developers to *take care* of their code. He claims that "the code you are trying to write today will be hard or easy to write depending on how hard or easy the

surrounding code is to read", as constantly reading old code is part of any effort to write new code [26].

It has been recognized that few studies have quantitatively assessed the claimed refactoring benefits [20, 47]. Especially with regard to development productivity, consensus is lacking, which might refrain managers from adopting refactoring [32]. At Exact, management is convinced of the benefits of refactoring, but it is still unclear *how much* benefit refactoring brings, which can make it difficult to determine how many resources should be available for refactoring.

1.2 Problem Statement

There is a gap of knowledge if we consider the claimed refactoring benefits on the one hand and the little empirical evidence on the other hand. Some studies have measured that refactoring can lead to an increase in productivity over multiple weeks, when new features are added to the system (e.g., [37]). However, we could not find any work that studied if refactoring increases *understandability* in such a way, that developers need less time to understand a piece of code.

At Exact, it can happen that a software engineer, when fixing a bug or making a small change, needs multiple hours to understand a piece of code, while it turns out that the required change is but a few lines of code. The complexity of the code and functions with side-effects that do more than their names suggest, are what makes program comprehension time-consuming. To what extent can refactoring reduce that time?

This brings us to our main research question:

Main Research Question.
How much difference in time is there when performing a small coding task on refactored code, compared to if the code had not been refactored?

The fact that we limit ourselves to small coding tasks means that our study does not involve large maintenance tasks. Thus, with small coding tasks, a large part of the time required to finish them will be taken up by code understandability.

The context of this research is Exact. We found that studies that evaluated the effects of refactoring in an industrial environment are rare. Experiments are often conducted with relatively small systems and with test subjects from an educational environment (e.g., [32, 37]). Our research should be a valuable addition to earlier work on refactoring, as it (1) was performed within a large, industrial organization, (2) targeted a large and complex legacy software system, and finally (3) focuses on the, until now, underexposed (with respect to empirical research) relation between refactoring and code understandability.

1.3 Research Methodology

We sought an answer to our main research question through quantitative, empirical research, carried out at Exact. For this, we set up a series of experiments where we let developers at

Exact complete coding tasks on components of Exact Online that we had refactored. This section describes what steps we followed to conduct our research.

First of all, we consulted literature to see how code quality can be defined and what refactoring is. This background knowledge is required to know when it makes sense to refactor. Furthermore, we need to know when refactoring leads to an increase in code quality. Without a notion of code quality we have no basis on which we can say that a refactored piece of code is better than its original form. Therefore, we present our first sub-question:

Research Question 1.
In what ways is code quality described?

With a sufficient knowledge of code quality, we would be able to refactor components of Exact Online in a well-considered way. We recognize the fact that 'refactoring' in general is a very broad term. Not all refactorings are similar, which is something that should be taken into account. We expected that results would depend on the *radicalness* of refactorings, that is to say, the extent to which the original structure of the code was altered. Therefore, we define three sub-questions that categorize our findings with respect to the number of classes that were involved in refactoring. Our second research question focuses on intra-class refactorings, which are often small refactorings such as method extractions.

Research Question 2.
What is the difference in completion time between people that finish a change task on code that has been refactored with small refactorings, such as Extract Method, and those that finish a task on the original code?

The next question aims at refactorings where an additional class was added. In our question we also state that the refactored code is covered by unit tests. Our rationale is that larger refactorings present risk, so that in practice such refactorings must always be accompanied by unit tests.

Research Question 3.
What is the difference in completion time between people that finish a change task on code that has unit tests and that has been refactored so that an extra class was added, and those that finish a task on the original code?

Our last research question aims at large refactorings where multiple classes have been added, where the end result can be considered a redesign of the original functionality so that responsibilities are divided over separate classes. Again, we incorporate unit testing into this question for the same reason as before.

Research Question 4.
What is the difference in completion time between people that finish a change task on code that has unit tests and that has been refactored, so that responsibilities were divided over multiple classes, and those that finish a change task on the original code?

To answer Research Questions 2-4 we made multiple refactorings of Exact Online components that we classified as either small, medium or large. Then, for each refactoring we designed a small coding task that would take no more than one hour to complete. For each refactoring, we ran experiments where test subjects would try to complete the coding task on either the refactored or original component, so that we were able to measure the differences in time between subjects working on original and refactored code.

1.4 Thesis Outline

This thesis has been structured as follows. First of all, Chapter 2 gives an answer to our first research question. It explores different views on code quality.

Next, Chapter 3 gives a thorough account of the refactored components that we created for the purpose of our experiments. We think it is important for any research that draws conclusions based on refactorings to include the decisions made during refactoring, so that the reader can put things into perspective. When to refactor and how far to go can be highly subjective. Too much refactoring can lead to over-engineering [19]. Furthermore, there is the phenomenon of being *Patterns Happy*, which applies to developers that implement design patterns at every opportunity even when there is no need for it, which can lead to an unnecessarily complex design. Hence, we strive for openness about our *modus operandi*.

Chapter 4 presents the experimental results. First we explain the set-up of the experiments, then we introduce the hypotheses that we formulated *a priori* for each experiment. After that, we present our results and validate our hypotheses.

We discuss our results in Chapter 5 where we put things into perspective and answer Research Questions 2-4. After that, Chapter 6 compares our results to other studies and explains how our research complements earlier work. Finally, Chapter 7 answers the main research question and gives an overview of our contributions.

Chapter 2

Code Quality

In this chapter, we describe different views on code quality to give an answer on our first research question. A good understanding of code quality is required to be able to assess when refactoring is justified. The goal of refactoring is to improve the structure of source code, to improve its quality, so that it becomes easier to e.g. understand or maintain code. This chapter describes different aspects of code quality, so that it is possible to reason about improvements to source code.

While some aspects can be measured objectively, other aspects are more subjective. The latter particularly applies to the notion of clean code. Low code quality can have a serious impact on the maintainability of software, for which Baggen et al. [2] show four examples: "when a change is needed in the software, the quality of its source code has an impact on how easy it is: (1) to determine where and how that change can be performed; (2) to implement that change; (3) to avoid unexpected effects of that change; and (4) to validate the changes performed." Before the different aspects of code quality will be discussed, we first look at the formal definition of refactoring.

2.1 Definition of Refactoring

Martin Fowler [13] defines a refactoring as follows:

Definition 1. *"A refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour."*

Examples of refactorings are: the removal of duplication, the simplification of complex logic, and the clarification of unclear code [19]. The definition above highlights the important characteristics of a refactoring. First of all, it preserves the behaviour of software from the perspective of a user. When designing software for a customer, it means that a refactoring does not add immediate value to the product for the customer. However, the benefits of a refactoring *can* make it possible to deliver more value, or to deliver faster. From Fowler's definition we can discern three benefits: improved understandability, maintainability and reusability.

Kerievsky [19] sums up the most common motivations for refactoring:

- Make it easier to add new code.
- Improve the design of existing code.
- Gain a better understanding of code.
- Make coding less annoying.

To know what structures of code have design flaws and could benefit from refactoring, there are a large number of *code smells* that indicate areas that usually need improvement.

Definition 2. *"A code smell is a surface indication that usually corresponds to a deeper problem in the system" [13].*

Code smells act like warning signs; the metaphor is that a smell indicates that something is not quite right. Code smells can apply to methods, classes, namespaces (packages) or entire systems. Note the word 'usually' in the definition. A code smell does not always correspond to an actual problem, so human investigation is in order. Examples of self-explanatory code smells are *Long Method*, *Large Class* and *Duplicated Code*. Other smells such as *Shotgun Surgery* and *Feature Envy* might be less known. A common theme is that the name of a code smell often refers to a metaphor, which makes the meaning of a smell easier to remember. Catalogues of code smells (e.g., [13]) serve to provide a common vocabulary for developers to rapidly communicate about design problems [19].

It is clear that refactoring is about improving characteristics of code from a developer's point of view. It follows that the process of refactoring is a matter of *quality*. Though, what is better understandable to one developer, might not necessarily be more understandable to another one. Therefore, we will further explore the notion of code quality in the following sections.

2.2 A Quantitative View

There are quite some different models that describe code quality in a quantitative way. For example, the Software Improvement Group measures the following properties of source code to assess its quality [2]:

- **Volume** (lines of code)
- **Redundancy** (percentage of duplicated code)
- **Unit size** (lines of code per unit)
- **Complexity** (cyclomatic complexity per unit)
- **Unit interface size** (number of parameters per unit)
- **Coupling** (number of incoming calls per module)

Those properties can be measured with the aid of different metrics. Example metrics have been given within parentheses. As the Software Improvement Group does, those measurements can be aggregated to be able to give a verdict on the quality of a code base, compared to what is average in the software development industry [2].

Software engineers can often calculate code metrics on their code base themselves. For most IDE's, plug-ins that calculate code metrics can be installed if support has not already been integrated in the IDE itself. For .NET, used by Exact, code metrics for cyclomatic complexity (CC), depth of inheritance (DIT), class coupling (CBO) and lines of code (LOC) are available in Visual Studio. Visual Studio also provides a composite metric, called the maintainability index. The maintainability index exists in different variants [49]. Microsoft calculates the index as follows, as a a value between 0 and 100 [31]:

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

Though using the maintainability index to quantify maintainability is an effective approach, one needs to keep its limitations in mind [49]. For example, the metric does not take coupling or cohesion into account. Therefore, it may be worthwhile to use a larger set of metrics to gain more insight. Table 2.1 lists a number of common code metrics [18]. They can be calculated with a tool such as NDepend[1] for .NET.

WMC	Sum of the cyclomatic complexity of each member method of the class
LCOM	Lack of Cohesion in Methods
CBO	Coupling Between Objects, the number of distinct classes that are accessed from a class
DIT	Depth of Inheritance Tree
NOC	Number of immediate descendants of a class
NOP	Number of class methods that provide polymorphic behaviour

Table 2.1: Examples of code metrics [18].

In practice, one often is not interested in the precise values for each metric. Ultimately, one rather wants to know if the quality is adequate, what kind of risks exist, and what steps may be taken to eliminate those risks. Therefore, code metrics still require interpretation, which might not be easy considering the wide range of metrics that exist. Fortunately, there are some static analysis tools that go one step further and do the interpretation for you. For example, the static analysis tool inCode[2] can detect code smells in Java, C++ and C, such as God Class, Code Duplication and Feature Envy to name a few. Its larger brother, inFusion[3], can even calculate the changeability, reusability and understandability of a code base. For

[1] http://www.ndepend.com/

[2] https://www.intooitus.com/products/incode

[3] https://www.intooitus.com/products/infusion

.NET, there is NDepend[4]. An interesting question is: how effective are such code smell detection tools at uncovering real problems that impact maintainability?

Yamashita and Moonen observed that code smells are just partial indicators of maintenance difficulties [51]. They used inCode to detect code smells on systems for which they had identified maintenance problems based on the experiences from developers working on the source code. They found that only 58% of the maintenance problems could be explained by occurrences of code smells. Other problems were caused by other code characteristics or by combinations of code smells. Furthermore, code smells could potentially interact with one another, making it important to keep the whole picture into account and to avoid considering code smells in isolation. They concluded that code smell detection is of limited value as a single predictor of maintainability [51].

While currently existing code metrics can give a global assessment of code quality and can help to identify problem areas, it must be acknowledged that they do not yet capture all aspects of code quality. For Exact, a large part of maintenance time was not spent at changing the code, but rather at trying to comprehend the code before the change could be made. There has been quite some research within the scientific community to tools and techniques that can aid in *program comprehension* through static and dynamic analysis [43, 7]. Examples of such techniques are the visualization of large execution traces or the extraction of UML diagrams. Although such techniques can be very helpful in trying to understand a complex system as a whole and to uncover its architecture, I think that, for the largest part, program comprehension at Exact is directly determined by the understandability of the source code. This is in line with Chhabra et al. who say that source code understandability is very important from a maintenance point of view [21].

There have been some proposals for source code understandability measures [6, 21]. For example, Chhabra et al. proposed two measures: code spatial complexity and data spatial complexity [21], where the first one focused on the required effort to comprehend software modules, whereas the second one focused on the efforts needed to comprehend the purpose and uses of variables. Unfortunately, such measures only provide a partial view of understandability. For example, we did not find any measures that took the meaningfulness of function and variable names into account. Obviously, this is a very difficult aspect to measure, but that just shows how difficult it is to measure source code understandability in a complete way. While there are a lot of metrics that can measure the changeability of software, no satisfactory measures for understandability seem to exist yet. So, as Welker said, determining maintainability purely by objective measures can be deceiving [49].

2.3 A Qualitative View

If we recognize that, with the current state of research at the time of writing, software metrics, despite their usefulness, are still limited at describing code quality and that, secondly, the way they should be interpreted is not always obvious, it is not a bad idea to pay some attention to more informal descriptions of code quality.

[4] http://www.ndepend.com/

2.3.1 Aesthetics

Developers can often tell if code is 'good' or 'bad' at a glance. It could be said that code quality is also a matter of *aesthetics*. Börstler et al. demonstrated this when they compared two example programs that implemented a Date class, and called one program the *Beauty* and the other program the *Beast* [5]. The Beauty had been decomposed into small classes and methods. The authors listed four reasons why they thought the Beauty was beautiful: firstly, key concepts in the problem domain were modelled as separate classes, secondly, the interfaces and implementations were small, thirdly, carefully chosen identifier names improved readability and finally, the decomposition supported independent and incremental comprehension, development and testing. In contrast, the Beast did not have any of those characteristics. It had been written as one large, monolithic method. The only advantage of the Beast was that all code was in one place, so that there were fewer lines of code in total. The authors aptly illustrated the differences by noting that, in the Beauty, the complexity and thinking had gone into the design, so that the units of code had become simple [5]. Whereas in the Beast, the design was trivial, but the code was complex. In the end, it is better to have a complex design and simple code rather than the other way around, because the design only needs to be made once and its complexity can be mitigated by providing diagrams of the design, whereas complex code demands its price every time it needs to be read.

2.3.2 Clean Code

A common practice among developers is to call code 'clean' or 'dirty'. What is clean code? If we search for mentions of the term 'clean code', it appears that the meaning has changed over time.

Early mentions

In the '80s and early '90s clean code could mean code that was optimized to achieve high performance on supercomputers [41]. Later in the '90s and 2000s we see the meaning coming closer to what we presently associate with clean code. In 2000, Edwards [11] said: "Clean code is easy to read; this lets people read it with minimum effort so that they can understand it easily." Clean code is very much related to *understandability*. The XP community has mentioned that the goal of test-driven development (TDD) is 'clean code that works' as quoted in Kent Beck's Test-Driven Development By Example from 2003 [3]. The famous TDD cycle 'Red, Green, Refactor' contains an explicit refactoring step, so that after having created code that works, the code is cleaned up [3]. However, despite the grown attention to clean code, a clear definition of clean code was still lacking.

Handbook of Clean Code

The definition of clean code became clearer when Robert C. Martin published his 'handbook of agile software craftsmanship', called *Clean Code* [26]. While it does not give one defi-

nition of clean code (it does give several), the term clean code has been strongly associated with the rules described in this book ever since.

In Clean Code, several authors that are well respected in the software community, described the characteristics that they associate with clean code [26]. If we extract some common themes, we could say that clean code is elegant, simple and direct, reads like a story. Perhaps most importantly, if we take maintainability into account, clean code can be understood and changed by someone else than the original author. Ward Cunningham expressed it like this [26]:

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

What this means is that clean code has no surprises for its reader, the intent of the code should be obvious and straightforward. Clearly, in this case understandability would be optimal. From these descriptions, we can see that the definitions of clean code are very similar, but that they can vary. Clean code could refer to an ideal, optimal scenario that may not be reached easily. That optimum might even be subjective and could be different from person to person. However, clean code is still something that we can aim for and come close to.

Some of the rules of clean code for functions are the following [26]:

- *Small!* "The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that [26]." While this may sound vague, because it does not mention any numbers, it means that one should be challenged to make functions smaller. A developer should look at his own work critically to see if a function can be decomposed further.

- *Functions should do one thing.* If taken literally, one could interpret this rule as if it is only allowed to have one statement within a function, which would be pointless. Martin uses an alternative description by saying that statements within a function should be at the *same level of abstraction*.

  A single level of abstraction means, for example, that business logic should not be mixed with the manipulation of a string buffer. Mixing low-level and high(er)-level details within the same function should be avoided. Details should rather be moved to private helper functions.

- *Use meaningful names.* Functions and variables should have meaningful names. Though this rule sounds like an undisputed truth, it is often broken in practice.

- *No side-effects.* Functions should do what their names suggest and nothing more. Functions that change the system state without that being clear from the description of that function, can easily lead to bugs.

There are many more rules in Clean Code. Of course there is a relation between some rules and code metrics. For example, if functions are small, then they cannot have a deep

nesting structure, so that their cyclomatic complexity would be low. But this relation does not have to exist. After all, meaningfulness of identifiers is something that is hard to measure as we saw previously, but it can certainly impact code understandability and cleanness.

2.3.3 S.O.L.I.D. Principles

Important principles of agile object-oriented design are the five S.O.L.I.D. principles, coined by Robert C. Martin [25]. These principles have precise definitions, so they do not suffer from the vagueness that surrounds the notion of clean code. Following these principles should result in more maintainable code. While clean code is mostly about understandability, the S.O.L.I.D. principles focus more on maintainability. We will present the definitions of those principles here, as they will be referred to later on.

Definition 3. *Single Responsibility Principle (SRP) [25]: "A class should only have one reason to change."*

Definition 4. *Open-Closed Principle (OCP) [25]: "Software entities should be open for extension, but closed for modification."*

Definition 5. *Liskov Substitution Principle (LSP) [25]: "Subtypes must be substitutable for their base types."*

Substitutability means that functions that expect the base type as a dependency, should still work correctly if they receive a subtype instead. The LSP also relates to the theory of *design by contract*: subclasses should "require no more, and promise no less" [30].

Definition 6. *Interface Segregation Principle (ISP) [25]: "Clients should not be forced to depend on methods that they do not use."*

Definition 7. *Dependency Inversion Principle (DIP) [25]: "a. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
b. Abstractions should not depend on details. Details should depend on abstractions."

An alternative interpretation of the DIP is the heuristic "Depend on abstractions [25]."

2.3.4 Legacy Code

If we consider clean code to be on one side of the spectrum, then we could say that we have *legacy code* on the other side. Legacy code is a rather informal term that is often used to describe code that is difficult to change and to understand [12]. Often, violations of the S.O.L.I.D. principles are everywhere in legacy code. The term 'legacy' implies that the code is old, written by a previous generation of programmers, which the current programmers have to deal with. Of course, new code can be written in the same hard-to-understand way, so that even new code that exhibits similar characteristics could be considered legacy code. Functions are often large and complex as the monolithic 'Beast' from Börstler et al. [5]. Because of this complexity, changes to legacy code can often have unintended side-effects. Therefore, changing legacy code is risky.

However, whereas the term legacy code could indicate that the code is old, it does not imply that the software product itself is a legacy system. Legacy code can be part of systems that have been used for a large number of years and that have been constantly updated to new business requirements, i.e., the needs of their users or customers. Therefore, a legacy code base that has grown over the years, can represent a lot of business value. If a system with legacy code is still actively developed, the risk that is involved in changing the software, could become a burden. Furthermore, the low understandability of the complex code could considerably slow down development.

Code Without Tests

Feathers defined legacy code in a rather black-and-white way as *'simply code without tests'* [12]. This describes the worst scenario. Without tests, any change to the code could potentially break the system [26]; there can be no sense of security. Every time a change is made, the application needs to be run to make sure that it still works [12]. Covering legacy code with automated tests is often not possible straight away, as legacy code often lacks testability. To improve testability, the code should be changed to allow for testing. Yet, changing the code is the very thing one needs tests for. Feather called this the legacy code dilemma [12].

Definition 8. *The Legacy Code Dilemma [12]: "When we change code, we should have tests in place. To put tests in place, we often have to change code."*

To change functionality in such a situation, firstly, one could break the dependencies that impede testability by doing small refactorings, then one could write tests for the functionality under test, then one could change the functionality and perform larger refactorings while using the tests for verification [12].

Characteristics

Feather's definition, if taken literally, would imply that if some tests are written for legacy code, then it is no longer legacy code. However, even from Feather's work we can infer that there are a lot of other characteristics that legacy code often has. Even with tests, complex code that is difficult to understand could still have an impact on software maintainability.

Here we present a summary of typical characteristics of legacy code [12]:

- "In a well-maintained system, it might take a while to figure out how to make a change, but once you do, the change is usually easy and you feel much more comfortable with the system. In a legacy system, it can take a long time to figure out what to do, and the change is difficult also."
- Legacy code has large, complicated classes and large methods.
- When changing legacy code, programmers often do not create new classes but keep the large structures in place. Feather calls this phenomenon *conversation scrutiny* and gives the following example: "When I'm looking at four or five classes that have

about a thousand lines of code apiece, I'm not thinking about adding new classes as much as I'm trying to figure out what has to change."

- When one thinks he has successfully made a change, it turns out that the same change needs to be made in many more places, because there are so many places with similar code within the system.
- All legacy code has bugs, usually in direct proportion to how little it is understood.

From the foregoing we can understand that even if tests are added to a legacy system, not all problems are solved directly. Tests allow change [26], but without refactoring the internal structure of legacy code to improve understandability, maintenance might still be impacted.

2.4 Application to Exact Online

So far we discussed code quality from various perspectives. As an answer to our first research question, we can say that code quality is sometimes described by different (aggregations of) code metrics. These can be useful to give a quick overview of the health of a code base. At the same time, we saw that they cannot capture every aspect that developers associate with code quality. Code quality is often also described in informal terms such as 'clean' and 'dirty'. This can be compared to being able to recognize good art from bad [26]. Developers have formed principles and guidelines that are associated with high quality code, which can be used to determine the level of code quality in a qualitative way.

As we made clear in Chapter 1, Exact Online qualifies as legacy code. Some parts of the code are very old and difficult to understand. Because the system has grown so much, a lot of complexity has been added over the years.

Fortunately, there is a large set of functional tests that can automatically test the web user interface of Exact Online. They act as a regression test suite so that Exact can roll out new functionality often, while the developers can be sure that new functionality will not break existing functionality. Moreover, more and more unit and integration tests are added. However, considering the enormous size of Exact Online, increasing the coverage of the unit test suite can only be a long term process.

While Exact Online does have tests, it still has many other characteristics of legacy code. It takes a lot of time to understand code, before it is possible to change it. As can be expected from legacy code, one could spot many design flaws and code smells within the code base.

The next chapter will describe refactorings that have been performed on the legacy code for this research. For all refactorings, the rules from *Clean Code* [26] were used as guidelines. Also other works on refactoring and design patterns were heavily used for inspiration and guidance [25, 14, 13, 19].

Software metrics had not been used to guide the refactorings, for a number of reasons. First of all, improvements to legacy code might result in only small, or even negligible differences in measurements. Considering the huge size of a legacy code base, refactorings by itself are only "baby steps"; they often lead to a design that is simply just a few steps

more maintainable than the design was before [12]. Breaking down a large class just to make it easier to work with can make a significant difference in applications [12], even though it might not drastically improve code metrics.

Of course, one could calculate software metrics for the refactorings described in this work. In many cases one would find a reduced cyclomatic complexity, but nothing earth-shattering. However, in the eye of the beholder the difference between the original and refactored code would be much bigger. It is the difference between dirty and clean code, night and day; the Beauty and the Beast, to paraphrase Börstler et al.[5].

Chapter 3

Refactorings

This chapter describes the refactorings that have been used in the experiments that will be described in the next chapter. It explains what the differences between several refactorings are, what choices were made, and how these refactorings were carried out.

3.1 Classification

It is important to make a distinction between different types of refactoring. The goal of the research was to measure the increase (or decrease) of productivity of developers if they worked with refactored code. We expected that results would depend on the radicalness of refactorings. By simply renaming identifiers or extracting methods within a class, the code would be much more similar to the original code, than if a class was split up into multiple classes by applying design patterns. Furthermore, design patterns may range in complexity. Whereas the *Singleton* pattern is considered 'a very simple pattern' [26], other patterns might be less well known.

Name	↑ **#classes**	**Description**	**#targets**	**Unit tests**
Small	0	Rename, Extract Method, Introduce Explaining Variable	3	
Medium	1	Extract Class, Adapter	1	✓
Large	> 1	Dividing responsibilities, Redesign	2	✓

Table 3.1: Classification of refactorings.

Table 3.1 describes the classification that was used for the refactorings. The classification was mostly based on the number of classes that would be added by the refactoring, as a measure for radicalness. Therefore, the refactorings could be conveniently classified as small, medium or large. In total, there were three small refactorings, one medium refactoring and two large ones. The small refactorings mainly consisted of clean-up changes such as renaming and extracting methods, without affecting the number of classes. In general, when the number of extracted methods starts to grow, it might be a good idea to apply Extract Class to move some methods to their own class. By our classification, this would be called a medium refactoring. In our specific refactoring, we used the Adapter pattern.

The large refactorings involved a significant number of refactoring steps, so that the end result could be considered a redesign, where responsibilities had been divided over multiple, collaborating classes.

The medium and large refactorings were covered by unit and/or integration tests before refactoring so that it was possible to verify their correctness. The small refactorings were of low-risk, so we did not write specific unit tests for those. Since the results for each experiment depend on the nature of the refactoring in question, the remainder of this chapter will describe the refactorings in more detail.

3.2 Limiting Conditions

There were some limiting conditions that had to be taken into account while refactoring. First of all, the public interface of classes had to stay the same. Projects in Exact Online were set up in such a way, that it was not possible to find references to a method or class outside its own namespace. This had important implications. It was not possible to get a full list of components that called a certain public method. As a compromise, it was possible to do a full text search through the code base, but this was slow and could easily lead to mistakes. This also meant that the Rename tool in the IDE for renaming variables and functions, which would normally update all references, could not be used. Consequently, the public interface had to remain the same so that depending classes would remain compatible with a class after refactoring.

Secondly, we obviously had to adhere to Exact's code style guidelines. While most of those guidelines in fact promoted code cleanness, some were debatable. For instance, private class fields had to be prefixed with an underscore character. While this could be convenient in large, legacy classes, classes and functions should rather be small enough so that prefixes are not necessary [26]. After all, if a class is cohesive, there can be little confusion about what the instance variables are. Furthermore, it is argued that the more people read code, the more they learn to ignore the prefix to skip to the meaningful part, so that the prefix becomes practically useless [26].

Another convention was to mark interfaces with a prefix `I`, which can be seen as a distraction [26]. Whereas it is good practice to depend on abstract classes and interfaces [25], a prefix makes it wrongfully look like an exception. Another downside is that it encourages developers to give an interface and its implementation the same name (minus the `I`), whereas it would be better to give a more abstract name to the interface and a more specific name to the implementing class. The advantages of a more abstract interface name are, firstly, that the interface name does not expose the inner workings of the concrete class, which should be irrelevant to users of the interface, and secondly, that it leaves room for additional implementations in the future.

3.3 Small: Extracting Methods

We chose three targets to refactor in a small way by using simple refactorings such as renaming variables and extracting methods. Those are relatively low-risk refactorings that

do not take much time to carry out. Therefore, such small refactorings could be done all the time by developers during their tasks, as an application of the so-called *Boy Scout Rule* [26], "Always leave the campground cleaner than you found it", a metaphor to promote continuous improvement of source code to get to a cleaner code base. In our research we studied how much benefit could be achieved by such small refactorings, but first we will look at the three targets that were chosen.

The targets that were chosen as candidates for refactoring came from different namespaces (packages) to avoid being biased towards a certain kind of business or application logic. The following targets were chosen:

- `Exact.Project.Jobs\RejectionNotifer.vb`
- `Exact.Purchase.Core\PurchaseOrderTools.vb`
- `Exact.Contracts.Core\ContractProlongation.vb`

3.3.1 RejectionNotifer

The first target, RejectionNotifier, was a class that described a background process that sent out notification e-mails to users when one of their entries was rejected. The method `SendEmails()` originally contained 32 lines of code and had a very deep nesting structure with a cyclomatic complexity of 11. We applied the refactoring *Extract Method* [13] multiple times so that the code was transformed to the code shown in Listing 3.1.

The important things to note are the method calls printed in bold, which point to the extracted methods. If one would replace those method calls with the contents of those methods, then that would resemble the structure before refactoring. The names of the new methods act as helpful descriptions. The complexity is now distributed over multiple methods. We also extracted methods that wrote to the logs. One could argue that it does not make sense to extract a single line of code to a new method. On the other hand, by doing so, `SendEmailsForDivision(division)` now reads more like a story or recipe. If the method would directly write to the error log (inline), then the method would no longer have a single level of abstraction [26], because suddenly one needs to understand lower-level concepts such as `env.Logging` and `ProcessLogTypes`. By keeping methods at a single level of abstraction, there is a descent into details considering methods from top to bottom, with each private method calling the next private method [26]. A second reason, in the case of logging errors, is that it is advised in *Clean Code* to extract the bodies of try/catch blocks into their own methods as error handling or recovery is a detail that should not complicate the main method [26].

Besides *Extract Method*, the refactoring *Introduce Explaining Variable* [13] was used at lines 13 and 14 to introduce the variable `shouldSendEmails`. This should make the if condition at line 14 easier to understand. Both the introduced explaining variable as well as the extracted logging methods are good examples of refactorings that do not show any improvements in code metrics because there is no change in complexity or coupling. On the contrary, the metrics might be worse because there are more lines of code. Yet, intuitively one expects that introducing explaining variables improves maintainability and understandability.

Listing 3.1: `SendEmails()` and the methods extracted from it.

```
1
2       Private Sub SendEmails()
3         Dim divisionList As Object() = GetListOfDivisionsToProcess()
4         If divisionList IsNot Nothing AndAlso _urlToCustomerPortal IsNot Nothing
              Then
5           For Each division As Integer In divisionList
6             SendEmailsForDivision(division)
7           Next
8         End If
9       End Sub
10
11      Private Sub SendEmailsForDivision(division As Integer)
12        env.SwitchDivisionContext(division)
13        Dim shouldSendEmails As Boolean = CBool(env.Setting(SettingType.Division, "
              ProSendEmails"))
14        If shouldSendEmails Then
15          Dim persons As Object() = GetListOfPersonsPerDivision(env)
16          If persons IsNot Nothing AndAlso persons.Length > 0 Then
17            SendEmailsToAll(persons)
18          End If
19          LogSuccessForDivision(division)
20        End If
21      End Sub
22
23      Private Sub SendEmailsToAll(persons As Object())
24        Dim mailer As New ExactSmtpMail(env)
25        For Each person As Person In persons
26          Try
27            SendMailTo(person, mailer)
28            env.Setting(SettingType.Division, "ProLastRejectionNotification") = env
                  .DateTime.Now
29          Catch ex As Exact.EMail.Core.SmtpException
30            LogErrorDuringSendingTo(person)
31          End Try
32        Next
33      End Sub
34
35      Private Sub SendMailTo(person As Person, mailer As ExactSmtpMail)
36        Dim message As New MailMessage
37        With message
38          .IsBodyHtml = False
39          .From = New MailAddress(person.SenderEmail, person.DivisionName)
40          .To.Add(New MailAddress(person.Email, person.FullName))
41          .Subject = term.String(54696, "Notification of rejected entries.")
42          .Body = String.Format(term.String(54697, "This is an automated message to
                notify you that some of your entries were rejected. " &
43          "Please login to {0} and correct and resubmit these entries as soon as
                possible."), _urlToCustomerPortal)
44        End With
45        mailer.Send(message)
46      End Sub
47
```

```
48
49      Private Sub LogSuccessForDivision(division As Integer)
50        env.Logs.LogProcess(ProcessLogTypes.Info, "All notifications of rejected
              entries processed for division " & division.ToString)
51      End Sub
52
53      Private Sub LogErrorDuringSendingTo(person As Person)
54        env.Logs.LogProcess(ProcessLogTypes.Error, "Can't send rejection
              notification from address " _
55            & person.SenderEmail & " to address " & person.Email & " for division:"
                  & env.Division)
56      End Sub
```

3.3.2 PurchaseOrderTools and ContractProlongation

The refactorings performed in `PurchaseOrderTools` and `ContractProlongation` were very similar in nature to the ones on the `RejectionNotifier`. In both cases, one long method was targeted so that multiple helper methods were extracted from it, of which each one contained no more than 10 lines. Furthermore, many variables were renamed in `ContractProlongation`. This legacy component contained a lot of variables in Hungarian notation. For example, `objProrataCalc` and `oContrLn.blnUseProrata` were respectively renamed to `proRataCalculator` and `controlLine.ShouldUseProrata`.

Finally, there were a lot of inline comments to document the branches in if/else structures. Because those blocks were extracted to their own methods, the method name could give a description so that there was no longer a need for inline comments. Robert. C. Martin advises to not write a comment if a function call can be used instead [26]. The disadvantages of inline comments compared to function calls are that they are not checked by a compiler. Inline comments can become out-of-date if the code around it changes and the comments are not updated [26]. Therefore, inline comments can be unreliable. In some cases where a larger description was still helpful, XML comments (comparable with JavaDoc for Java) were added to document the extracted methods.

3.4 Medium: Adapting to Legacy Code

Not all legacy code is the same. While some legacy code can be refactored to better readable code within a relatively short amount of time, refactoring other parts might equal rewriting the entire system. One might think that this would impact the extent to which classes that depend on extreme legacy code can be refactored. The refactoring described in this section shows how one can work around this.

3.4.1 Original Code

This refactoring focused on the class `FinancialEntryTools`, on the method `ValidateDelete()` in particular. This method checked if it was allowed to delete a financial entry. There were three cases where this was not allowed, and for each case a specific error message was returned. The code is shown in Listing 3.2.

Listing 3.2: `ValidateDelete()` in class `FinancialEntryTools` before refactoring.

```
1  Public Function ValidateDelete() As ErrorMessage
2    'Basic checks
3    With bcFinancialEntry
4      'Check 1: Status = processed?
5      If CType(.Prop("Status").Value, GLTransactionStatuses) =
           GLTransactionStatuses.Processed Then
6        Return New ErrorMessage(env.Term.String(9085, "Already processed"))
7      End If
8
9      'Check 2: Period open? Only when entry is not draft
10     If CType(.Prop("Status").Value, GLTransactionStatuses) <>
           GLTransactionStatuses.Draft Then
11       If Not FinTools.IsFinPeriodOpen(env, CInt(.Prop("Division").Value), CInt(.
             Prop("ReportingYear").Value), CInt(.Prop("ReportingPeriod").Value),
             CStr(.Prop("Journal").Value)) Then
12         Dim sb As New StringBuilder
13         sb.Append(env.Term.String(3925, "Period is closed"))
14         sb.Append(": ")
15         sb.Append(CStr(.Prop("ReportingYear").Value))
16         sb.Append(" - ")
17         sb.Append(CStr(.Prop("ReportingPeriod").Value))
18         Return New ErrorMessage(sb.ToString)
19       End If
20     End If
21
22     'Check 3: Check source
23     If CInt(.Prop("Source").Value) >= 100 Then
24       Dim sourceOk As Boolean = False
25       If CInt(.Prop("Source").Value) = GLTransactionSources.PurchaseInvoice Then
26         'For now this property is only used for source PurchaseInvoice
27         If CInt(.Prop("AllowDeleteSourceGreaterThan100").Value) <> 0 Then
28           sourceOk = True
29         End If
30       End If
31       If CInt(.Prop("Source").Value) = GLTransactionSources.PaymentInstrument
             Then
32         If CInt(.Prop("AllowDeleteSourceGreaterThan100").Value) <> 0 Then
33           'It's allowed to delete transactions created for payment instruments
34           'in case the status change of the payment instrument is undone.
35           sourceOk = True
36         End If
37       End If
38       If CInt(.Prop("Source").Value) = GLTransactionSources.VatMarginToTaxReturn
             Then
39         If CInt(.Prop("AllowDeleteSourceGreaterThan100").Value) <> 0 Then
40           sourceOk = True
41         End If
42       End If
43       If Not sourceOk Then
44         Dim sSource As String = FinCache.GLTransactionSource.
               TranslatedDescription(env, CInt(.Prop("Source").Value))
45         If String.IsNullOrEmpty(sSource) Then
```

```
46            sSource = env.Term.String(1067, "Unknown")
47          End If
48          Return New ErrorMessage(env.Term.ConstructTerm(5809, "Source", 0, ":" &
                sSource))
49        End If
50      End If
51    End With
52
53    Return Nothing
54  End Function
```

In lines 12-18, which have been greyed out, an error message was constructed and returned. By applying *Extract Method*, these lines were extracted to the new method `CreateErrorForClosedPeriod()`. Likewise, lines 44-48 were extracted to the new method `CreateErrorForIncorrectSource()`.

The expressions that really prevented this code from being clean, were the calls to the different properties of the business component `bcFinancialEntry`, such as the 'Status' property at line 5. In clean code, one would expect to be able to get an attribute of a business component with a single call. However, as the sequence diagram in Figure 3.1 shows, three calls were required. First, an `ERProperty` object had to be retrieved, then its value was accessed and returned as an `Object`, and then it had to be cast to the proper type, corresponding to that property. This was a violation of the *Law of Demeter* [26]. It obviously made the code harder to understand; if not, the comment "*Check 1: Status = processed?*" would not have been necessary. Furthermore, it should not have been the responsibility of `FinancialEntryTools` to transform properties of a business entity into a workable form.

This way of retrieving properties from business components was not restricted to just the method in Listing 3.2; similar statements could be found throughout the complete source of Exact Online. To understand why this was the case, it helps to have a rough understanding of the data persistence layer of Exact Online.

3.4.2 Data Persistence in Exact Online

The first version of Exact Online was launched in 2005 and development had started some years in advance. The current data persistence layer of Exact Online dates back to that period. The layer is called 'Repository', but it should not be confused with the Repository design pattern. With the Repository pattern, a Repository class represents a collection of business entities, to which objects can be added or removed [14]. Behind the scenes, these business entities are persisted into a data source. Also, it is possible to query the Repository class to return one or multiple business entities. In this way, a Repository accepts and returns business entities, while the business entities themselves do not contain any persistence logic.

The 'Repository' in Exact does not follow that pattern. Instead, it contains *Active Records* [14], which contain both the data of a business entity, as well as the persistence logic. In Exact Online terminology, these Active Records are called *business components* and extend the abstract class `BusinessComponent`. Generally, the data of an Active Record can be accessed through public variables or getters and setters [26]. Thus, the attributes

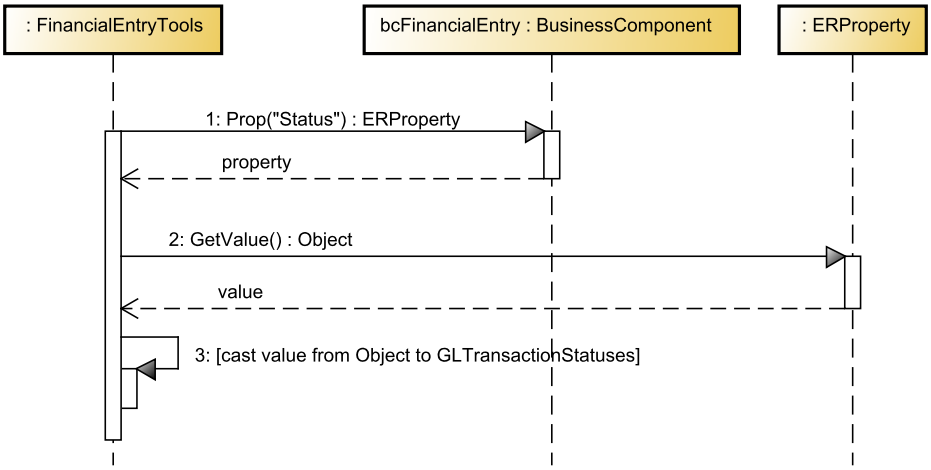


Figure 3.1: Getting the status property from a financial entry.

belonging to a business entity are hardcoded. In Exact Online, on the other hand, the data are stored in a hash map. The data can be accessed by calling the property `Prop(name : String)` and supplying the name of the desired attribute. Furthermore, a business component implements methods such as `Delete()` and `Update()` and contains validation logic.

A rather confusing aspect of working with business components, is that one instance of a business component does not necessarily correspond to a single entity during its lifetime. Changing the property `ID` of an instance has the effect that a database query is executed to retrieve the record belonging to that primary key, and that the instance itself is populated with those data. In other words, there is no strong mapping between a business object and a database record; the record that a business object points to, can be changed on the fly. In database terms, it can be said that a business component tries to work both on table as well as record level.

Getting back to our example, we can now understand why it was necessary that properties of a financial entry had to be casted to its proper type. The abstract class `BusinessComponent` contained a hash map from attribute names (`String`) to `ERProperty` objects, containing the value. Since this structure was specified on a high level for all business components, the classes could only return property values as generic `Objects`, as the specific types depended on the implementations of subclasses.

This might pose the question: is it possible to properly refactor a method such as the one in Listing 3.2 to clean code, while it depends on a suboptimal data persistence layer? Obviously, refactoring the entire data persistence layer was no option; it would have been a Herculean task that would affect the entire code base.

3.4.3 Preferred Interface

The solution was to look at the code from another perspective, to focus on what the code would look like in an ideal world. The comments in the original code already paved the way

for this. These comments had probably been written, because the code failed to convey the same message as easily as the comments did. In contrast, we chose to focus on how the code could have been made just as understandable as the comments. One of the many principles of clean code is to "don't use a comment when you can use a function or a variable [26]".

In an ideal world, instead of

```
'Check 1: Status = processed?
If CType(.Prop("Status").Value, GLTransactionStatuses) =
    GLTransactionStatuses.Processed Then
```

we would rather have a financial entry as a proper business entity with a method `IsProcessed() : Boolean` that we would check, so that we would write the statement shown underneath.

```
If financialEntry.IsProcessed() Then
```

We would write something similar for the second step. For the third step, we would rather give the business entity a method `SourceAllowsDeletion() : Boolean`, so that a financial entry could check its own internal data to determine if it might be deleted. This would increase encapsulation. Figure 3.2 shows the public interface that we would prefer for a financial entry.

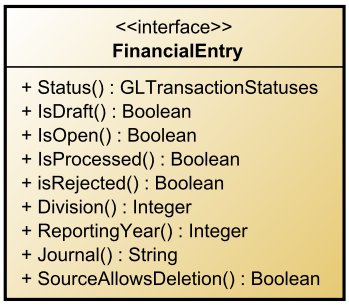


Figure 3.2: Preferred interface for a financial entry.

3.4.4 Applying the Adapter Pattern

Since we did not want to refactor the data persistence layer, we chose to use the *Adapter* pattern to adapt the legacy business component to the preferred interface. One of the reasons why one would normally use an adapter, is being unable to change the interface of a class because it is third party code [25]. In such cases, an adapter makes it possible to call the adapted class via the preferred interface.

At the beginning of Section 3.4 we stated that not all legacy code is the same. In fact, we could view extreme legacy code, such as the data persistence layer, as unchangeable 'third party code'. With this reasoning, we utilized the Adapter pattern as a bridge between the legacy code and the clean interface from Figure 3.2 that we were after.

Figure 3.3 and 3.4 show the situation as it was before and after refactoring, respectively. In the original situation, the class `FinancialEntryTools` went straight to the business component in Exact.Repository to access the properties in the cumbersome way that was

discussed earlier. After refactoring, `FinancialEntryTools` depended on our preferred interface (renamed to `IFinEntry` to adhere to code style guidelines), which was implemented by the adapter `FinEntryAdapter`. The adapter simply held a reference to the original business component, to which all calls were delegated. In this way, the casting of properties was hidden inside the adapter.

The benefit of introducing the adapter was that we could now rewrite `ValidateDelete()` in class `FinancialEntryTools` to the code shown in Listing 3.3. It should be evident that the logic in these 9 lines[1] is much more understandable than it was in the original 42 lines of code (see Listing 3.2). It should now be very easy for anyone to tell under which circumstances a financial entry may not be deleted.

Listing 3.3: `ValidateDelete()` in class `FinancialEntryTools` after refactoring.

```
1  Public Function ValidateDelete() As ErrorMessage
2    Dim errorMessage As ErrorMessage = Nothing
3
4    If _financialEntry.IsProcessed() Then
5      errorMessage = CreateErrorForProcessedEntry()
6    ElseIf Not (_financialEntry.IsDraft() OrElse FinancialPeriodIsOpen()) Then
7      errorMessage = CreateErrorForClosedPeriod()
8    ElseIf Not _financialEntry.SourceAllowsDeletion() Then
9      errorMessage = CreateErrorForIncorrectSource()
10   End If
11
12   Return errorMessage
13 End Function
```

Finally, it is also interesting to look at the implementation of `SourceAllowsDeletion()` in `FinEntryAdapter`, as much of the logic that used to be under 'Step 3' was moved to that method, to keep that logic close to the data. As Listing 3.4 shows, it is very clear from the return statement when the source of a financial entry does not prevent it from being deleted. This was accomplished by a repeated use of the refactoring *Introduce Explaining Variable* [13].

Listing 3.4: `SourceAllowsDeletion()` in class `FinEntryAdapter`.

```
1  Public Function SourceAllowsDeletion() As Boolean Implements IFinEntry.
       SourceAllowsDeletion
2    Dim allowedPurchaseInvoice As Boolean =
3      Source = GLTransactionSources.PurchaseInvoice AndAlso
           AllowedToDeleteSourceGreaterThan100
4    Dim allowedPaymentInstrument As Boolean =
5      Source = GLTransactionSources.PaymentInstrument AndAlso
           AllowedToDeleteSourceGreaterThan100
6    Dim allowedVatMarginToTaxReturn As Boolean =
7      Source = GLTransactionSources.VatMarginToTaxReturn AndAlso
           AllowedToDeleteSourceGreaterThan100
8
9    Return Source < 100 OrElse allowedPurchaseInvoice OrElse _
10         allowedPaymentInstrument OrElse allowedVatMarginToTaxReturn
11 End Function
```

[1] excluding blank lines and comments

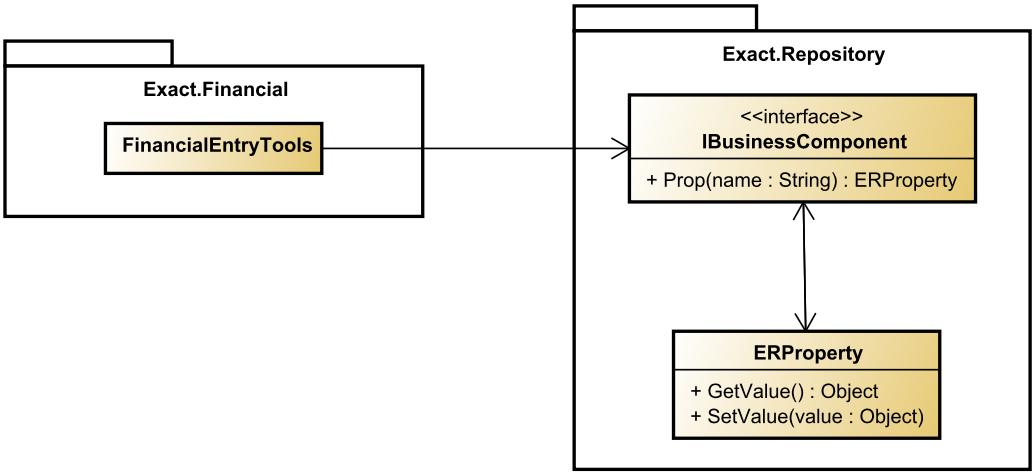


Figure 3.3: Before refactoring `FinancialEntryTools`.

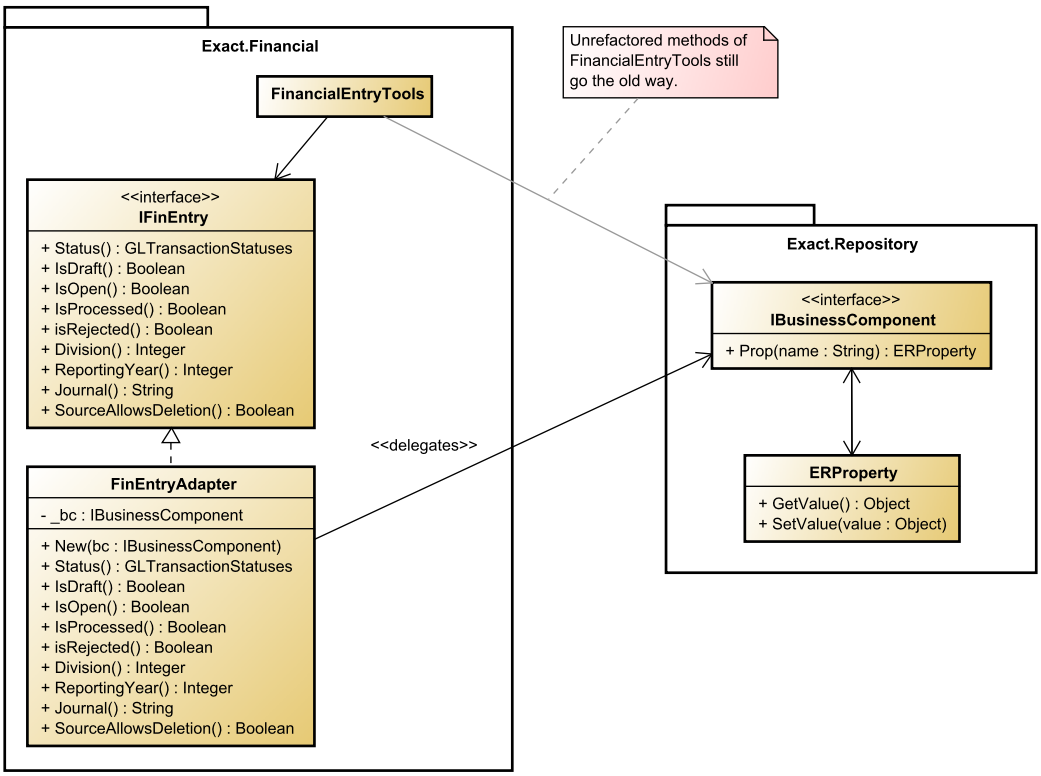


Figure 3.4: After refactoring `FinancialEntryTools`.

To conclude, this refactoring led to much cleaner and better understandable code, though at the expense of an added wrapper class. The Adapter pattern can be useful for moving away from parts of legacy code. Kerievsky [19] said about adapters that "they let me decide how I want to communicate with other people's code". We could say in a similar way that adapters let us decide how we want to communicate with legacy code. Whether or not this refactoring would lead to an expected increase of productivity for developers working on this code, was explored in the experiments that will be discussed in the next chapter.

3.5 Large: Dividing Responsibilities

3.5.1 Ubiquitous Code Smells

If one looks at the source code of Exact Online from a distance, one can easily spot three main code smells that affect most of the system: *Long Method*, *Large Class* and *Conditional Complexity* [13, 19]. We already saw small refactorings that dealt with long methods and complexity in the foregoing. In the coming sections, we will see refactorings that specifically target large classes with many responsibilities.

Additionally, there is one smell, whose name might be less familiar, namely *Inappropriate Static* [26], that really hits the nail on the head considering the way Exact Online has been designed. Many functions that contain business logic reside in large classes with static functions. To understand why there are many static functions within the code base of Exact Online, we need to look at the request flow to see how the use of static functions emerged from that.

Request Flow in MVC Frameworks

Nowadays, new web applications are often built on a framework that implements the Model-View-Controller (MVC) pattern. If we look at the trending repositories on GitHub[2] this month for PHP, the most popular web programming language, based on activity on GitHub and StackOverflow [22], then the top 10 includes no less than three MVC frameworks. In the proprietary domain, Microsoft provides the ASP.NET MVC framework. The flow of a request in popular MVC frameworks roughly looks like the flow shown in Figure 3.5.

All HTTP requests from users are redirected by the web server to a single entry point. That point is called HttpApplication in ASP.NET MVC, but it may be called differently in other MVC frameworks (e.g., in Symfony it is called HttpKernel). It represents the 'main' function of the web application. The HttpApplication will construct the objects that are important for the life cycle of the application or the request. Those objects could construct further objects themselves. Thus, the state of the application is constructed from top to bottom.

The Routing component (sometimes called Router) will parse the URL and other properties of the HTTP request to determine what page the user actually requests. Based on that information, the right Controller and Action (method of a controller) are called that correspond to a page. From there, the Controller uses data from the Model to render a View

[2] https://github.com/trending?l=php&since=monthly

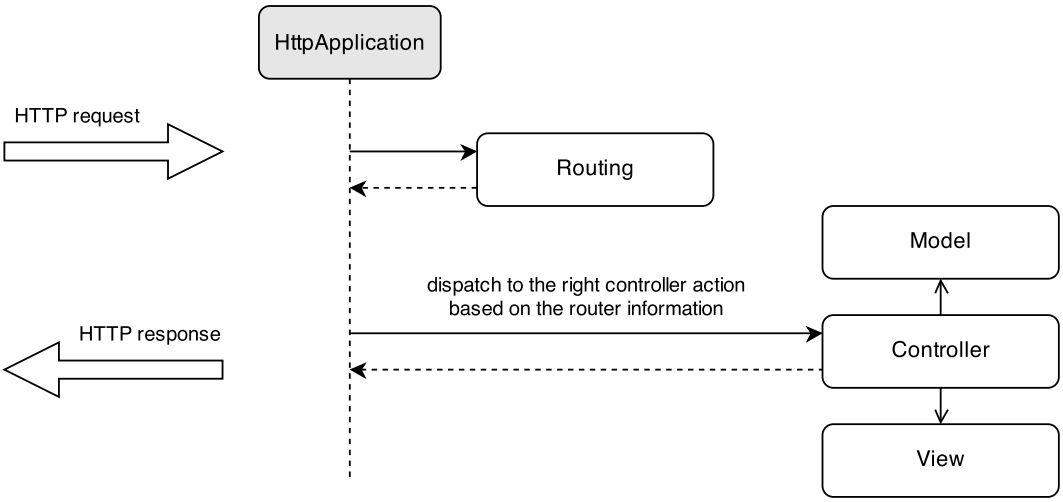


Figure 3.5: The flow of a request in a typical MVC framework.

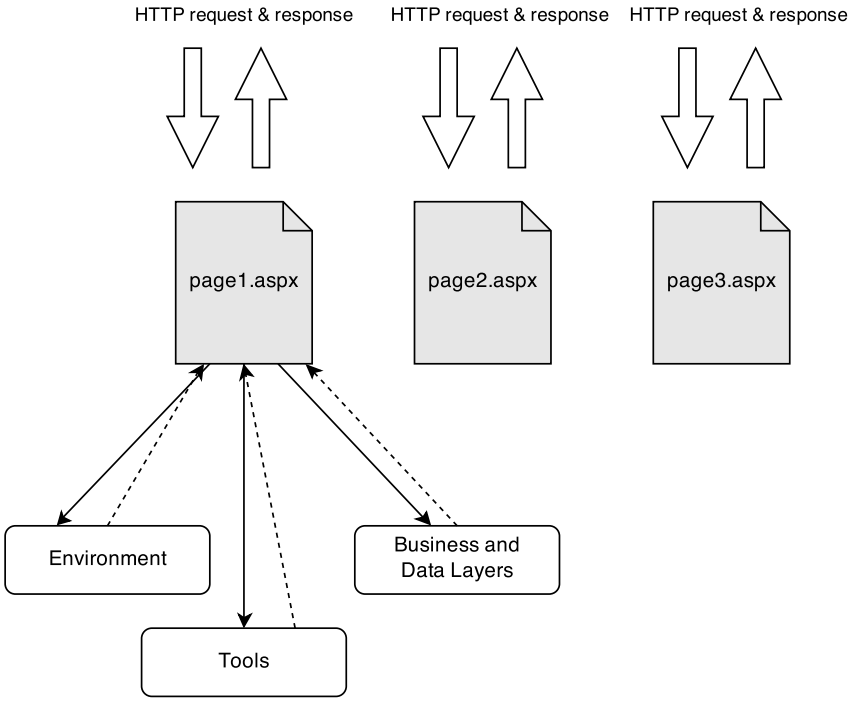


Figure 3.6: The flow of a request in Exact Online as a typical ASP.NET application.

that will be sent as a response. The interesting thing is that in this top-down approach, the Controllers containing the logic for a certain web page, are loosely coupled to the rest of the system. The dependencies that a Controller requires are either injected into it by the 'main' that constructs it, or the 'main' can inject a dependency injection container into the Controller that can provide the required dependencies. In other words, the construction of objects takes place on a high level, in line with the *Dependency Inversion Principle* [25], which implies that one should depend as much as possible on abstract classes and interfaces instead of concrete classes.

Request Flow in 'Traditional' Web Applications

While some of the latest functionality in Exact Online has been built on top of ASP.NET MVC, the largest part (the legacy code) is built on plain ASP.NET. In traditional web applications, the URL of a request directly corresponds to a certain file on the web server that renders the page. This means that, in contrast to the single point of entry in the case of MVC, traditional web applications have multiple entry points. The pages, of which the names end with `.aspx` for ASP.NET, are primarily responsible for rendering the view. However, since a page is an entry point, there is no sense of an application yet. In other words, the generation of high-level components needs to be triggered from the individual pages. Instead of a top-down approach, the application starts at a low-level component, i.e., the page, then constructs the high-level components to use, and then the flow returns to the page that uses higher-level components to construct the view. Although this might sound like a questionable approach, one should not forget that we are concerned with legacy code and that such practices used to be commonplace.

Each page in Exact Online will construct an `Environment` object. The environment takes care of a lot of core functionality: it controls the user session, it handles security by making it possible to check if a user is authorized for certain actions, it also provides logging functionality, et cetera. Since the Environment is so large and has so many responsibilities, it can be called a *God Class*: a class that tries to do everything. It is also a *dependency magnet* [26]: many other classes depend on it, because it is often the only way to reach important services (such as session information). Then there is also a factory that can construct instances of business components from the data persistence layer that was described in Section 3.4.2.

Now, if one wants to reuse some application logic across multiple pages, what options are there? Pages lack scope. Because all pages are entry points, you could essentially consider them to have a global scope. The only context is given by the Environment. So to reuse functionality there are three options:

- Add functions to `Environment`, so that they can be accessed from anywhere. Obviously this would be wrong for a lot of reasons.
- Create a new class and make the pages that need it, instantiate it.
- But then one could just as well create a class with static functions, so that explicit class construction is not necessary. Furthermore, such a class could even share static properties between requests.

In such a situation, creating classes with static functions seems like a natural solution. For Exact Online this meant that a lot of utility classes (e.g., `PurchaseTools` and `PriceTools`) were created with static functions. It is only one step further to start calling other static classes from static functions. The end result is many large static functions, often calling other static functions, passing a lot of data like integers and strings around the system (a smell called *Primitive Obsession* [13]). The static context implies a global context where the lines between responsibilities begin to fade. Despite the use of an object-oriented programming language, it means that large parts of the system had actually been coded with a procedural paradigm in mind.

Towards More Object-orientation

The two large refactorings that will be presented in the following sections, resulted in a redesign where multiple classes were added to the system. The idea was to go from a situation with large classes and methods to a situation where there were multiple small classes with small methods that collaborated together to provide a certain service to the rest of the system.

3.5.2 Refactoring ERDomain

The class `ERDomain` is a typical example of a large class with too many responsibilities. It is instructive to look at the history of the component, to see how it had become more complex over time.

Analysis

Figure 3.7 shows the *raison d'être* for this class, namely to provide a validation service to the rest of the system. The shared function ERDomain.Validate(env : Environment, domain : String, value : Object) can be called from anywhere in the system to have a value validated according to a specified domain, which can be considered a set of validation rules. This is a large function of 87 lines of code with a cyclomatic complexity of 48. However, this function is not the target of the refactoring, because it hardly changed over time. Rather, a lot of changes were made (and bugs were found) in the other methods of `ERDomain`. Since one can call the validation method with only the name of a domain, the method needs to load the actual domain object with validation rules. Therefore, there was a helper method called GetDomainByName(). In time, the methods by which to get domain objects became more complex. Looking at the history of ERDomain in TFS[3], we could distinguish three important points in time.

1. Initially, it was only possible to get domain objects by their name. Figure 3.8 shows how domains were retrieved from the database. After retrieval, the domains were cached in a dictionary as $(name, domain)$ pairs. The dictionary provided fast $O(1)$ read access. The cache was shared among requests. Because of concurrency it was important that any additions to the cache would be executed within a critical section.

[3] the version control system in Microsoft Visual Studio

2. After a while the feature was introduced to load domains by their ID instead of their name. The flowchart in Figure 3.9 is very similar to the previous one; this reflects the changes to the code: the function GetDomainByName() was simply duplicated as GetDomainByID(). The reason for that is that a second dictionary needed to be added as a cache. The original dictionary contained $(name, domain)$ pairs. To be able to additionally look-up domains by their ID in $O(1)$ time, a second dictionary with $(id, domain)$ pairs was added as well.

3. The final change, shown in Figure 3.10 made the class much worse. An option was added to be able to read domains from XML files instead of from the database. This introduced even more complexity and responsibilities to `ERDomain`. Because of the duplication introduced earlier, the logic for deciding which data source to use, was also added at two locations.

The description and the flowcharts already describe so much functionality, that it can be hard to imagine that all that functionality was contained within one class. The most important code smells that were found in `ERDomain` were *Duplicated Code* and *Large Class*. Moreover, it violated the *Single Responsibility Principle* (see Section 2.3.3). We saw that the class had changed for multiple reasons: to extend the caching mechanism and to be able to retrieve domains from another data source. Furthermore, especially the last change had a large impact on the structure of the code; a clear violation of the *Open-Closed Principle*.

We identified the following responsibilities in `ERDomain`:

- validation;
- caching;
- data retrieval from the database;
- data retrieval from XML files.

Furthermore, the class had to deal with concurrency. When dealing with concurrency it is advisable to keep critical sections as small as possible [26]. Since quite some statements were executed within the critical sections and because the class was so large, it was difficult to have a clear overview of what statements were executed in a critical section. In Clean Code it is even advised to keep concurrency-related code separate from other code [26].

We analysed `ERDomain`, a large class with a lot of responsibilities. Next, we present a series of refactorings that changed ERDomain to a group of classes with their own distinct responsibilities.

Refactoring

As a first step of refactoring, the large validation method in `ERDomain` was moved to its own class `ERDomainValidator`. We noted that it had hardly changed over time, while the rest of the class underwent a lot of changes. In other words, it did not share a reason to change with the rest of the class; it was clearly a separate responsibility. Moving the validation

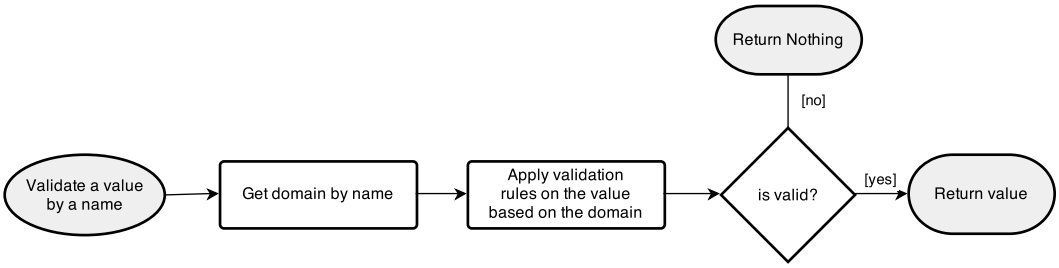


Figure 3.7: Flowchart of validation functionality in `ERDomain`.

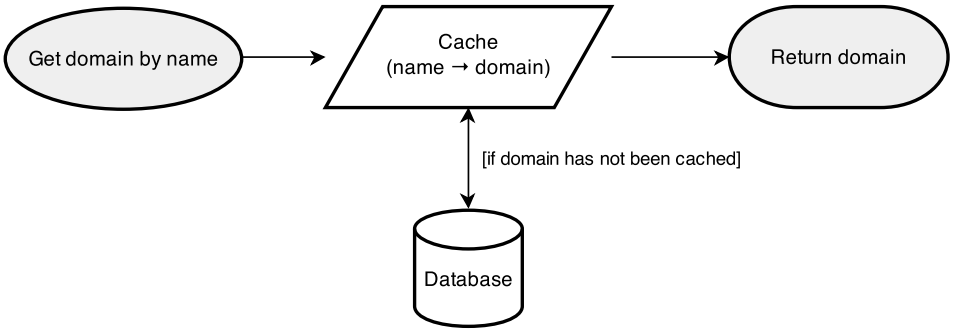


Figure 3.8: Functionality in 2011: getting domains by name.

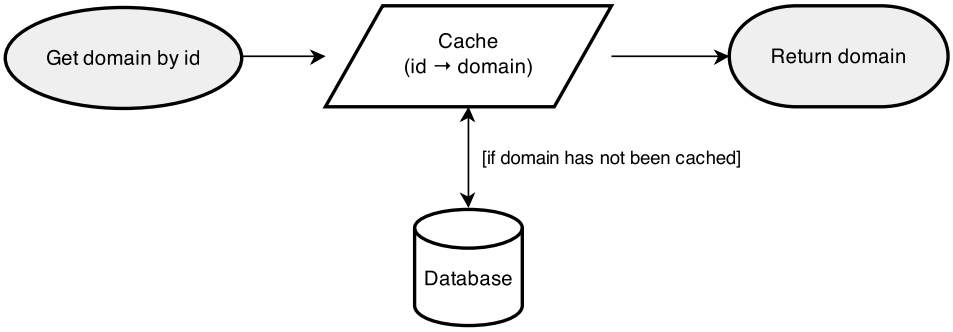


Figure 3.9: Extension: getting domains by ID.

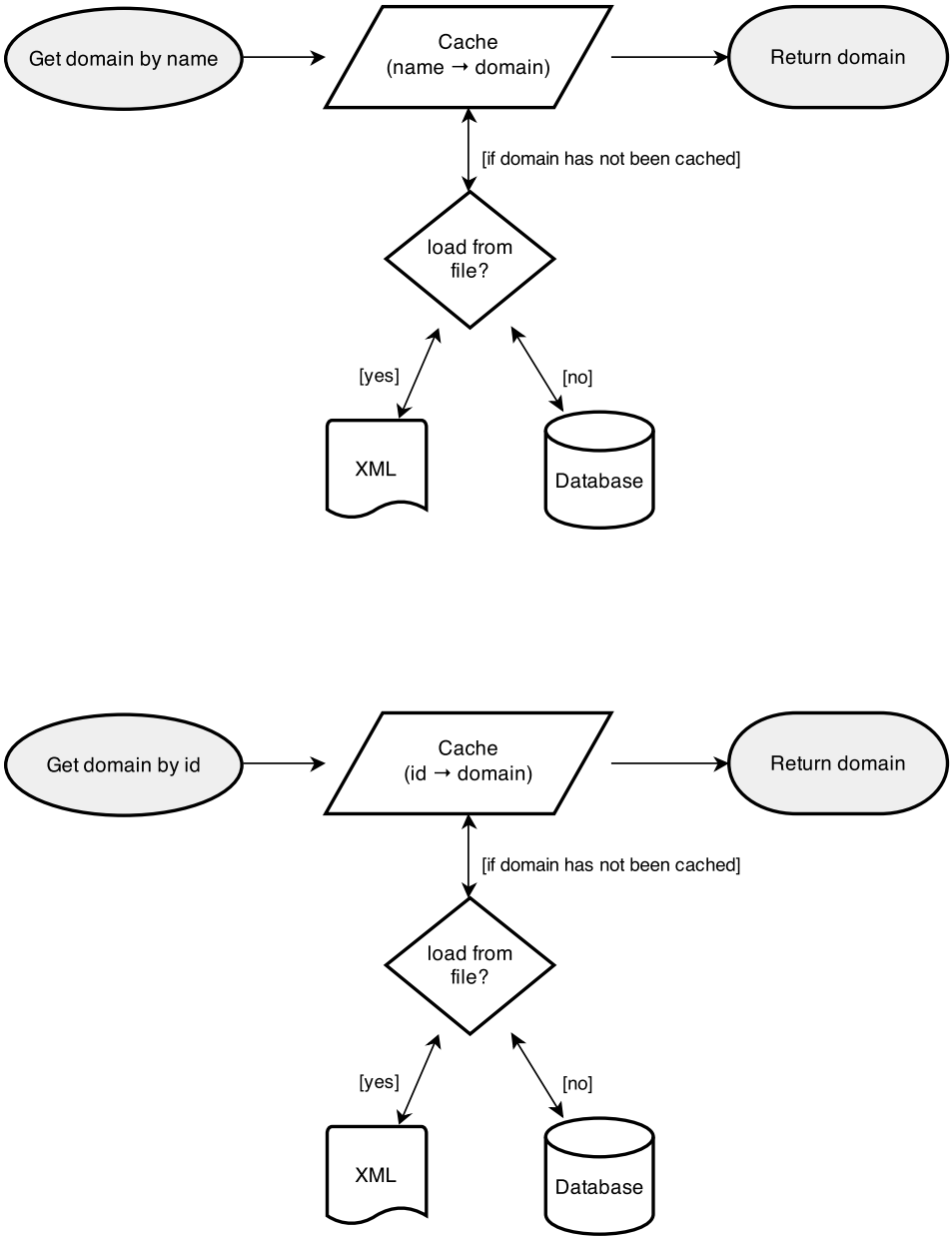


Figure 3.10: Extension: optionally reading domains from files instead of from the database.

logic to its own class was also a matter of *organizing for change* [26]. By moving it out as a first step of refactoring, we could be fairly certain that the correctness of the validation logic would not be harmed accidentally while the rest of `ERDomain` was being refactored.

The reason why a lot of duplication was introduced, was that domains had to be retrieved by two properties: the name (of type `String`) and the ID (of type `Guid`). We solved this situation by *polymorphism*, see Figure 3.11. We created classes `DomainLookupKeyName` and `DomainLookupKeyId` that respectively wrapped the name and ID values whose interfaces were unified in the abstract base class `DomainLookupKey`. The classes implemented the *Value Object* design pattern [14]: two separate `DomainLookupKey` objects containing the same value will be treated as equal. To hide the details of the subclass implementations to the rest of the system, we created two factory methods in `DomainLookupKey`, so that other classes only had to depend on the abstract base class. We changed `ERDomain` so that it used a single dictionary with (`DomainLookupKey`, `RepositoryDomain`) pairs to replace the two previous dictionaries. After that, we could remove the duplicated behaviour.

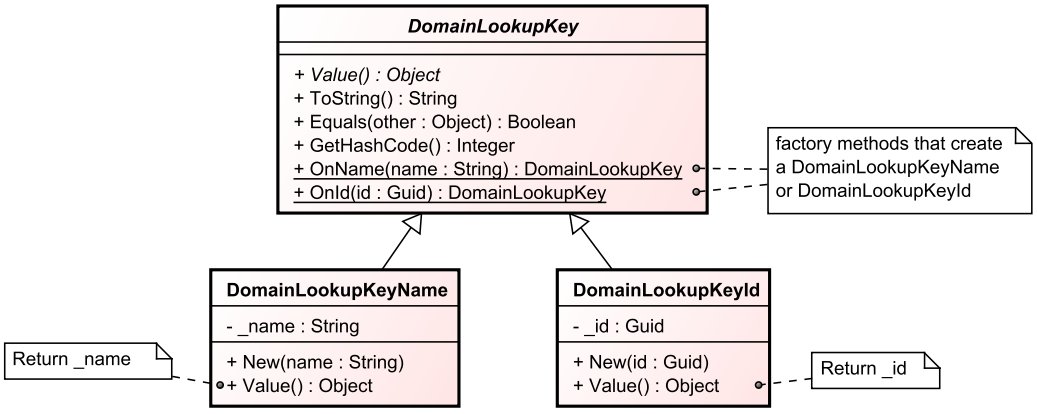


Figure 3.11: Class diagram showing `DomainLookupKey` as a polymorphic Value Object.

The *Large Class* code smell was targeted by introducing the following classes, which are shown in Figure 3.12:

- `ERDomainCollection` represents the cache contents as a dictionary of (`DomainLookupKey`, `RepositoryDomain`) pairs.
- `ERDomainManager` retrieves domains from the cache and handles concurrency. If a domain is not found in the cache, then it will load it from an external data source via an implementation of `IERDomainsLoader`. Because of the loose coupling provided by the interface, `ERDomainManager` does not depend on the data source being used. This follows the *Strategy* design pattern [25]. Finally, `ERDomainManager` was implemented as a *Singleton* [25].
- `ERDomainsFromDbLoader` uses the database as a data source.
- `ERDomainsFromFileLoader` uses an XML file as a data source.

- `ERDomainValidator` contains validation logic.
- `ERDomain` only provides backwards compatibility so that the public interface of `ERDomain` remained intact. It delegates to the new classes.

It is important to note that the list of classes above roughly corresponds to the list of responsibilities that we identified in our analysis. The end result was a clear separation of concerns. The classes (except the unchanged `ERDomainValidator`) were all small (< 100 lines of code excluding blank lines and comments) and therefore their units were easy to understand.

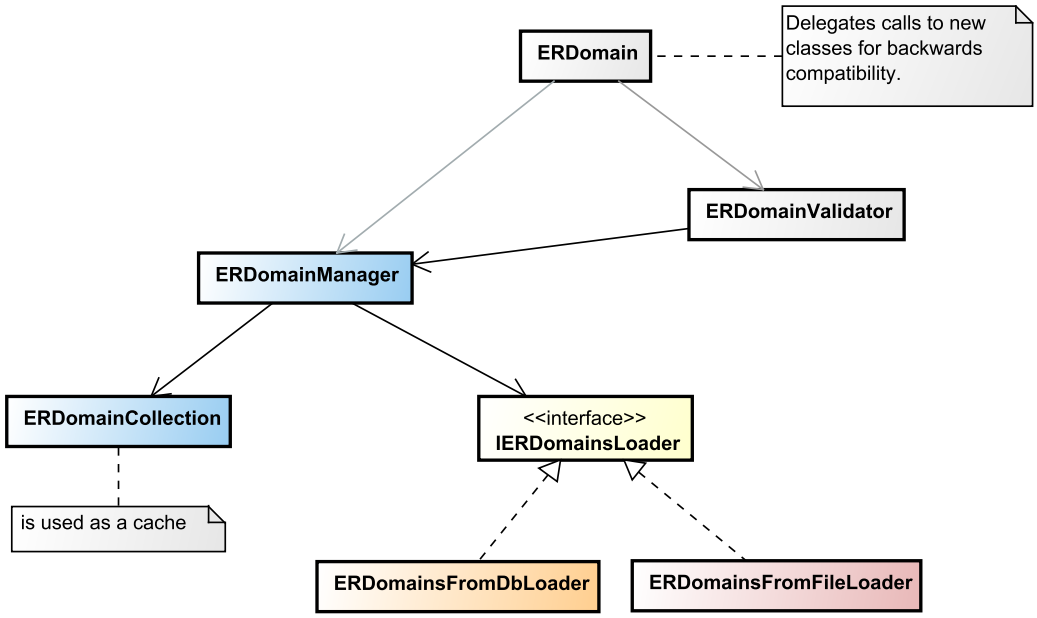


Figure 3.12: A clear separation of concerns with multiple classes.

To be able to verify the correctness of the refactoring, `ERDomain` was thoroughly covered by both unit and integration tests. If we ignore the presence of a private constructor that was never hit, then both the unit and integration tests each achieved a 100% block coverage of `ERDomain`, where a block is defined as "a sequence of consecutive statements that has exactly one entry and exit point" [27]. Because the original code was very tightly coupled to other dependencies, the original unit tests relied heavily on a mocking framework to mock the dependencies (such as loading from database and from XML). The downside of that was that the unit tests took 4 seconds to run. This may not sound like a big deal, but consider that Exact Online consists of a lot of components. If all components have unit tests that require a few seconds to run, a run of the full unit test suite might take very long. Another downside is that the unit tests did not treat `ERDomain` as a black box and they were therefore coupled to the implementation, which could make it difficult to change the implementation without breaking the tests.

After our refactoring, we could refactor the unit tests as well. Because of the abstraction introduced by the interface `IERDomainsLoader`, it was possible to use a *Fake* object [29] in the unit tests that implemented this interface. Instead of loading domains from a real data source, the Fake object would load some domains that could be specified in the test setup. This led to the coverage shown in Figure 3.13. As would be expected, the classes that do the loading from file and from the XML are largely untouched by the unit tests, which use a Fake instead, but they are tested in the integration tests. The unit tests for `ERDomain`, which were also extended to cover the new `DomainLookupKey` component, now took less than a second to run in total.

To conclude, we went from one monolithic, static class (`ERDomain`) to a design with multiple small classes with their own responsibilities working together to achieve the desired functionality. Furthermore, code duplication was removed. The testability was increased so that the unit tests could give instant feedback.

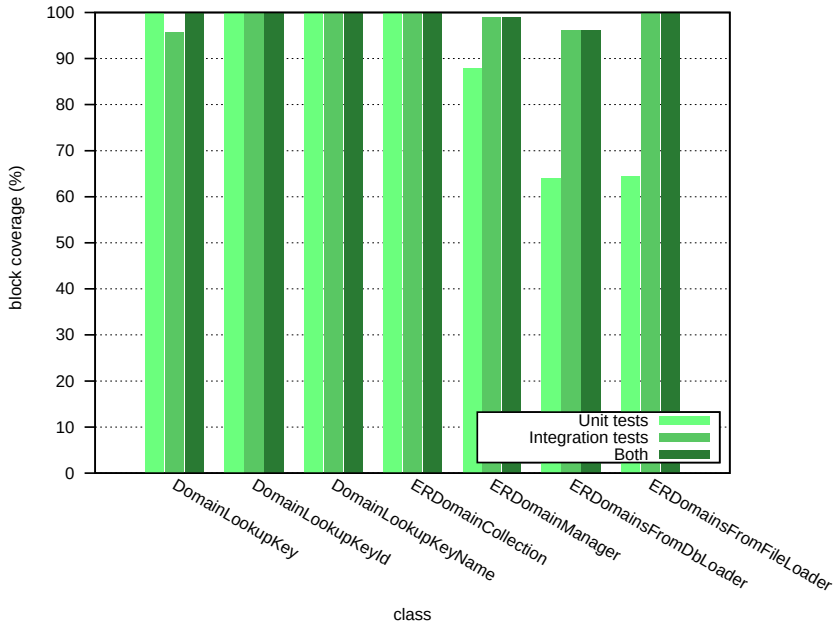


Figure 3.13: Block coverage of `ERDomain` after refactoring.

3.5.3 Refactoring FinTools

The class `FinTools` is one of those large utility classes described in Section 3.5.1, which contains a lot of static helper methods to process data relating to the financial domain (bookkeeping). One of those methods is the following:

```
FinTools.ValidateEntryNumber(env : Environment, entryNumber : Long,
         increment : Boolean, division : Integer, journal : String,
         reportingYear : Integer, entryID : String) : Integer
```

It is a long method of 35 lines of code[4] and it has a cyclomatic complexity of 10. It contains a very complex algorithm that is difficult to understand.

The functionality of the function is as follows. When, during bookkeeping, one saves an entry in a journal, then the number assigned to that entry should be unique within that journal. The function `ValidateEntryNumber()` checks if the proposed entry number is not in use. If yes, table 'Freenumbers' in the database is updated to mark that number as taken. If not, then the function will automatically try other numbers from 'Freenumbers' until it finds a valid one. The exception is when one updates an existing entry instead of creating a new one, then the entry number is already in use by itself, which then also counts as valid.

The function breaks a lot of rules from Clean Code. It is not small, it does a lot of things that even have side-effects (such as updating the database while validating), and it requires more than three parameters (7!). Furthermore, the naming is quite confusing because of the distinction between entryNumber and entryID. The latter one is an internal ID that guarantees uniqueness across the system, whereas the entry number is visible to users and is only unique within a certain journal. The entry number is what is actually validated, whereas the entry ID is a given.

The goal of the refactoring was to increase the understandability, and to increase testability by decoupling the algorithm from the database logic. Because there were no tests for this component yet, the first step was to write integration tests that covered the validation algorithm. The integration tests achieved a block coverage of 100%. Unfortunately, the testing tools in Visual Studio could not calculate branch, path or complete coverage. We made sure by manual inspection that we had covered all paths through the algorithm.

The refactored design is shown in Figure 3.14. The algorithm was moved to a separate class `EntryNumberValidator`. One of the advantages of that was that instance variables could be used to keep track of the state of the algorithm. That made it possible to extract certain steps of the algorithm to private methods. Considering the arguments of the original function, we could identify a smell called *Primitive Obsession* [13]. A lot of data was passed around the system as primitive types such as integers and strings, while they were related data that could be together in a class. Quite some functions accepted the combination $(division, journal, reportingYear)$ as a context for some operation. It represents a certain financial year in a journal of a certain administration (division). We decided to group it together in a reusable class `JournalYear` to solve the code smell and thereby reducing the number of function arguments. To make it clearer what the relations between the other data were, we created a class `EntryPosition` that contains the entry ID (as identification of the entry whose number needs to be validated), the entry number that is validated and the journal details.

The `EntryPositionRepository` made it possible to get `EntryPosition` objects with data from the database. By giving the entry ID, one gets an `EntryPosition` in return, if the entry exists, with the entry number filled in. The `IEntryPositionRepository` interface made it possible to use a Fake object to be able to unit test `EntryNumberValidator`. For the same reason, `IFreenumberEnvironment` was introduced. The real implementation

[4] excluding blank lines and comments

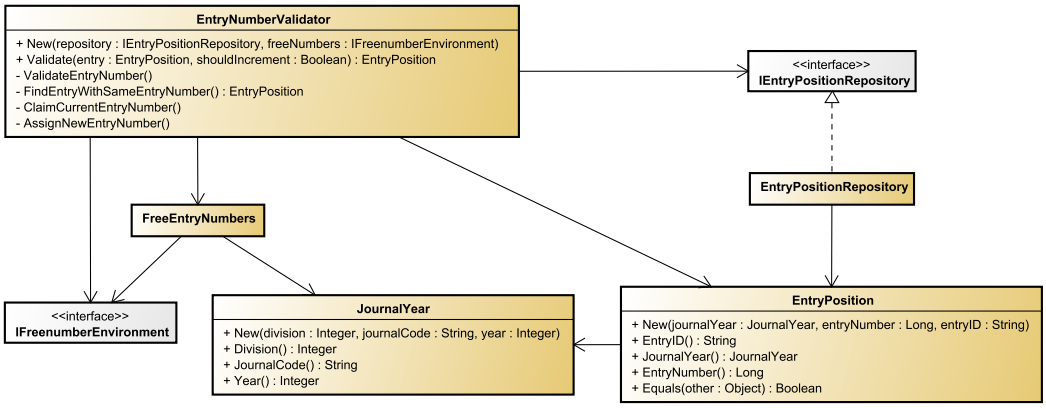


Figure 3.14: The new `ERDomainValidator` and its collaborating classes.

`FreenumberEnvironment` interacted with the database. Separating all database logic from the logic of the algorithm was consistent with the *Single Reponsibility Principle*. Because the `EntryNumberValidator` depends on interfaces for all classes except the data classes (`JournalYear` and `EntryPosition`), the class follows the *Dependency Inversion Principle*.

Because the old public interface had to remain intact, a new responsibility was given to the function `FinTools.ValidateEntryNumber(...)`. It constructed the `EntryPosition Repository` and constructed the `EntryNumberValidator`, while injecting the dependencies in its constructor. Then it simply called the `Validate()` function in `EntryNumber Validator`.

For the new design we could write fast unit tests that covered all paths in the algorithm, which was not possible before. Furthermore, all introduced classes and methods were small. All classes were within 100 lines of code and the largest method contained only 12 lines. Finally, the heart of the algorithm was captured in a high-level story of which the code is shown in Listing 3.5. The details of the algorithm were moved to private helper functions to keep function statements at the same level of abstraction. By repeatedly using the refactoring *Introduce Explaining Variable* [13] we intended to further increase understandability.

Listing 3.5: The algorithm in `EntryNumberValidator` was refactored to a series of high-level steps.

```
1     Private Sub ValidateEntryNumber()
2       Dim existingEntry As EntryPosition = FindEntryWithSameEntryNumber()
3       Dim entryNumberIsNotTaken As Boolean = existingEntry Is Nothing
4       If entryNumberIsNotTaken Then
5         If _shouldIncrement Then
6           ClaimCurrentEntryNumber()
7         End If
8       Else
9         Dim editingExistingEntry As Boolean = existingEntry.Equals(_entry)
10        If editingExistingEntry Then
11          AssignNewEntryNumber()
```

```
12            End If
13          End If
14      End Sub
```

3.6 Summary

Multiple refactorings were discussed in this chapter, ranging from small refactorings such as simple method extractions to large redesigns. A common characteristic of all refactorings was that methods were sized down, even though it increased the number of lines of code in the system as a whole. We strived towards self-explanatory functions to increase understandability. In the large refactorings, we divided responsibilities to increase changeability. Finally, the large refactorings allowed us to write fast unit tests that increased our confidence in the correctness of the code and that would make it easier to change the implementation in the future.

Chapter 4

Experiments

In the previous chapter, we discussed how we refactored various components of Exact Online to increase their code quality. In this chapter, we present the experiments that we performed to establish if our refactorings lead to a difference in the time it takes to understand and change the targeted code. After we have clarified our approach, we will present and discuss our experimental results.

4.1 Approach

In order to measure the difference in productivity between developers working with original and those working with refactored code, developers were given a set of tasks to complete. For each experiment, half of the developers performed a task on the original code, while the other half performed the same task on the refactored code. The developers only performed a single task for each component. After all, if they would perform multiple tasks on the same component, then knowledge acquired during the first task would affect the second task.

Developers were split into two groups, A and B. We made sure that each group contained a roughly equal number of senior and non-senior software engineers, based on their job description. Both groups alternated between performing a task on original or refactored code, as shown in Table 4.1, so that all persons worked with original and refactored code. This effectively means that for each experiment, persons unknowingly switched from the

		Group A		**Group B**	
	Experiment	Refactored code Experimental group	Original code Control group	Refactored code Experimental group	Original code Control group
1.	RejectionNotifier	✓			✓
2.	PurchaseOrderTools		✓	✓	
3.	ContractProlongation	✓			✓
4.	PurchaseOrderTools		✓	✓	
5.	ERDomain	✓			✓
6.	PurchaseOrderTools		✓	✓	

Table 4.1: Roles of the test persons for each experiment.

experimental group to the controlled group, so that the overall outcomes would depend less on the group compositions.

4.1.1 Date and Location

Exact is an international company that applies globally distributed software development. This means that not all software engineers are located in Delft. To have a larger and more representative sample set, the experiments were not only held in the Netherlands, but also in Malaysia.

The experiments were first held in Delft in the first week of March, 2014. Two groups of 5 developers took part in these experiments. Since most software engineers are located at Exact Asia Development Centre in Kuala Lumpur, Malaysia, experiments were also held there with two groups of 10 developers (see Figure 4.1) during the first week of April, 2014.

In total, *30 developers* took part in the experiments for a full day, so that both the experimental group and the control group each contained 15 developers.

(a) Group A (b) Group B

Figure 4.1: Participating software engineers in Kuala Lumpur, Malaysia.

4.1.2 Refactoring Training

To attract more participants, we made the experiments in Kuala Lumpur part of a one-week *refactoring training* that would teach them how to refactor to be able to control the technical debt of the legacy code and to increase testability. The experiments were sold to them as an opportunity to experience the differences between refactored and unrefactored code. By using this approach we could attract a large number of participants. Secondly, developers would be less aware of the fact that they were observed as test subjects.

On the first two training days, the two groups of developers took part in the experiments. On the following days, *after the experiments*, they received the actual refactoring training, which we constructed in collaboration with a trainer from Exact's training department. This refactoring training was given to all 20 software engineers in Kuala Lumpur who took part in the experiments. In the experiments they would see examples of refactored code; in the refactoring training they would then learn how to perform such refactorings themselves and how to take more advantage of object-orientation.

The training contents were as follows:

- Theory on technical debt and code smells, illustrated with examples of legacy code from Exact Online.
- A video talk by Robert C. Martin, author of Clean Code [26], followed by a discussion on clean code principles.
- Theory about using unit tests as a safety net for refactoring.
- Workshop: refactoring ERDomain.

The last part, the workshop, let the participants refactor ERDomain under our guidance over the course of 8 hours. We first let them analyse the responsibilities and the code smells of the class. We let them get rid of code duplication and split up responsibilities. Refactorings steps were alternated with theory on, i.a., the *Single Responsibility Principle* and the *Dependency Inversion Principle*. Furthermore, they applied the *Singleton* and *Strategy* design patterns. After each small refactoring step, we let participants run their unit and integration tests, so that they would get used to running the tests to receive quick feedback. Furthermore, the participants had to write some additional unit tests for a component that was added during refactoring.

The training, especially the workshop, was very much appreciated by all participants. Appendix B lists the feedback they gave after the training. It is obvious from the given answers that most participants learnt a lot. All participants said that they would recommend this training to their colleagues and some had very good motivations. For example, many people had recognized the problems of the legacy code, but until this training they had not known how to refactor to make it better. Also, many people were familiar with or had heard of the term 'technical debt', but taking part in the experiments and the training really made them feel the impact of it.

A second survey was held by the management in Kuala Lumpur three weeks later. A total of 61% was able to put into practice what they had been taught. They applied the boy scout rule, two teams worked on improving existing targets, and one team even applied test-driven development. Exact will strive to remove the obstacles that prevented the other 39% from refactoring and writing clean code.

4.1.3 Design of Tasks

We designed two kinds of tasks for the experiments: fixing a bug, given a bug description, or adding a small feature. The bugs were not actual bugs in Exact Online; they were wittingly inserted in the code for the purpose of the experiments.

How do developers of Exact Online solve bugs in their daily work? Bugs are usually observed at a high-level in the web interface. Developers then use a debugger to walk through the code to discover where an error originates from. After a bug has been found and a change is made, the developer needs to verify the fix. For many bugs, verification can be done by executing the scenario that triggered the bug in the web interface. Developers can even write a scripted UI test (as an end-to-end test) to make the verification steps repeatable.

However, this process is time-consuming. If we would have mimicked this process in the experiments, then we could have done much less experiments within the given time. Secondly, we thought that it might add noise to the experimental results, as there would have been more factors that could have an impact. For instance, people's experience with debugging tools, or the configuration of the local development server might have led to differences in time that were unrelated to whether or not the code had been refactored. An incorrectly configured environment might have led to invalid results; a risk that we did not want to take. Thirdly, there was also a technical limitation. For some components that we had refactored, it turned out to be very difficult to make a change directly visible in the user interface. If we inserted a bug that would lead to inconsistent data, sometimes the inaccuracy did not propagate to the user interface, because of additional validation logic in the presentation layer. Sometimes validation logic was duplicated in multiple layers, so that if we broke one instance, another instance would eventually correct the data. So, to make a bug visible, it would practically mean that additional bugs would have to be introduced in the presentation layer.

It meant that the developers could not verify their changes and therefore they could not know when they had finished their task. We chose a simple solution: developers had to raise their hand if they were confident about a change they had made, and then their change would be verified by the one leading the experiments. If their solution was incorrect, they could simply try again. For each experiment, the developers had to write down their starting and finishing times on a timesheet that was given to them, so that we could determine their durations afterwards. While the developers felt a disadvantage because their workflow was different, the understandability of the code was actually measured more soundly this way. The fact is that the test persons had nothing but the code to go on, so they were required to read and actually understand the code.

For each experiment and its corresponding task, we formulated the following hypotheses:

Hypothesis 1. *If given the refactored code, then one finishes the coding task for* RejectionNotifier *earlier than if the original code were given.*

Hypothesis 2. *If given the refactored code, then one finishes the coding task for* PurchaseOrderTools *earlier than if the original code were given.*

Hypothesis 3. *If given the refactored code, then one finishes the coding task for* ContractProlongation *earlier than if the original code were given.*

Hypothesis 4. *If given the refactored code and accompanying unit tests, then one finishes the coding task for* FinancialEntryTools *earlier than if the original code were given.*

Hypothesis 5. *If given the refactored code and accompanying unit tests, then one finishes the coding task for* ERDomain *earlier than if the original code were given.*

Hypothesis 6. *If given the refactored code and accompanying unit tests, then one finishes the coding task for* FinTools *earlier than if the original code were given.*

4.2 Background of Test Persons

We expected that the background of the participating developers would have an influence on the outcomes of the experience. Therefore, it was important to determine their background to be able to put experimental results into perspective. Many people who had worked for Exact for a long time, originally came from a procedural background. As was demonstrated in the previous chapter, large portions of the code base were programmed in a rather procedural instead of an object-oriented manner, despite the use of an object-oriented language. We took into account that the large refactorings might be too different from what developers were used to, so that perhaps they would solve their tasks faster on the original code than on the refactored code.

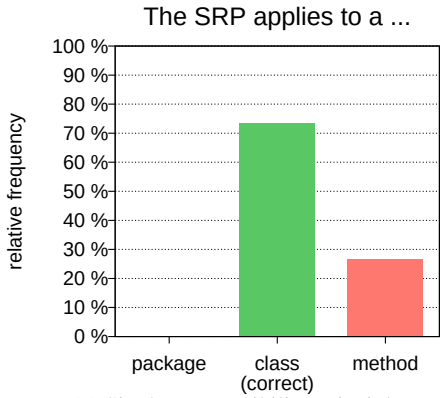
To determine people's experience with object-oriented design, each participant had to fill in a survey with questions on inheritance, overloading, and design patterns. Answers to those questions were either correct or incorrect. Others questions asked about people's practices, such as their preferences regarding method and class size. There is not one correct answer to such a question, but we presumed that people who already strived for small methods would be able to work faster with refactored code than people who were accustomed to large methods.

The complete questionnaire has been printed in Appendix A. The questionnaire was filled in just before the experiments were started and before the participants had seen anything of the refactored code. We made the following observations from the results:

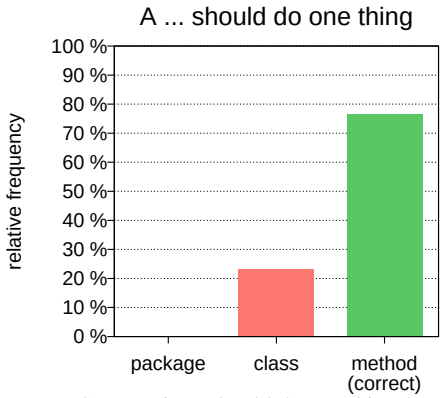
- According to Robert C. Martin, the Single Responsibility Principle (see Section 2.3.3) is one of the simplest of the principles, and one of the hardest to get right [25]. However, as Figure 4.2a and 4.2b show, the seemingly self-explanatory principle is often misunderstood and confused with similar rules such as "Functions should do one thing [26]". The SRP applies to classes, as part of their responsibilities is to govern the internal state that classes encapsulate. Functions, on the other hand, do not have such responsibilities, because they do not own any data; their only responsibility is to uphold a certain contract. If the SRP is misunderstood, then its full implications (e.g., high cohesion) might not be considered.

- Many errors were made in Questions 3 and 4 that focused on important concepts of object-orientation: inheritance and overloading. In total, 8 errors could be made with those questions. Figure 4.2c shows that only a third of the participants answered the questions correctly; two-third lacked basic knowledge of object-orientation. We can conclude that many participants did not know how to take advantage of the abstraction provided by object-orientation. It follows that they probably thought in a more procedural way.

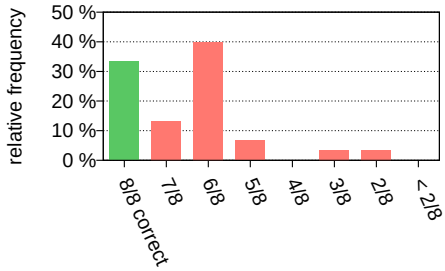
- Question 5 asked what kind of method call would be preferred in an object-oriented language such as VB.NET: `DocTools.GetAttachments(documentId)` or `document.GetAttachments()`, with the latter one being the correct answer. This question was motivated by the abundance of *Inappropriate Static* methods in the code base as discussed in Section 3.5.1, to see if people realized that this was a code smell in the



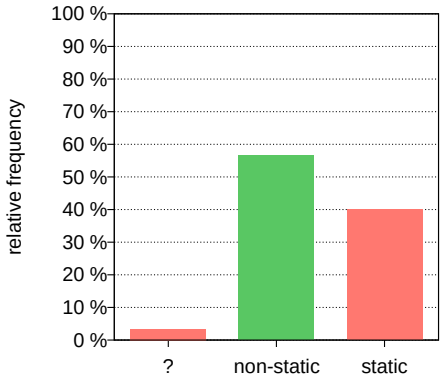
(a) Single Responsibility Principle



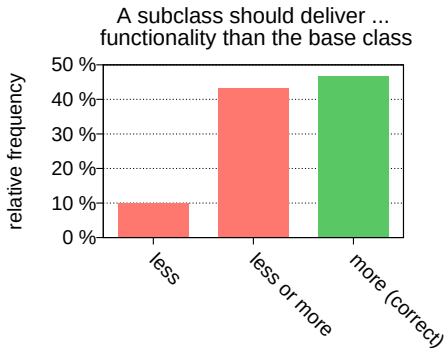
(b) Functions should do one thing



(c) Questions on inheritance and overloading



(d) Awareness of *Inappropriate Static* methods



(e) Liskov Substitution Principle

Figure 4.2: Answers to Q1-6.

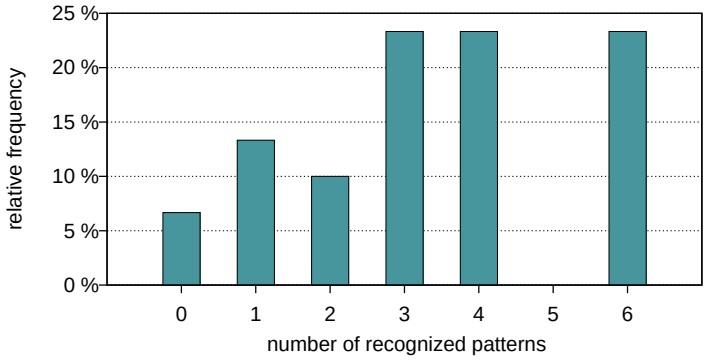


Figure 4.3: Q7: Correctly recognized design patterns.

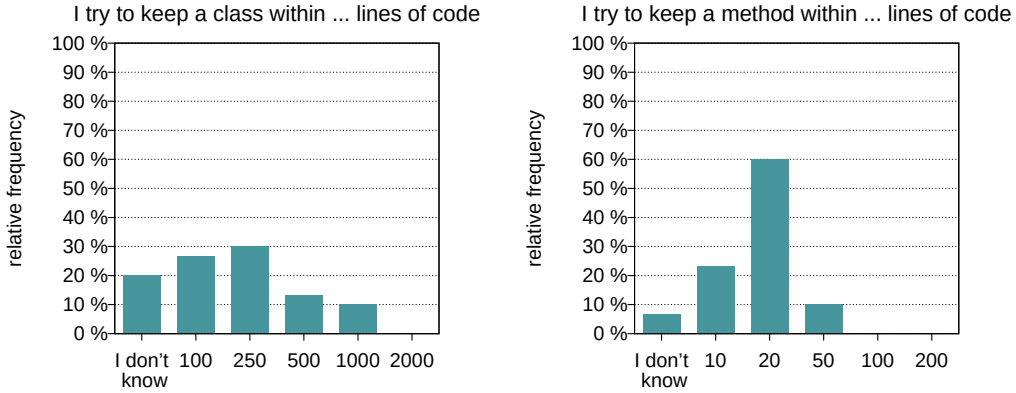


Figure 4.4: Q8-9: Preferred maximum class and method size.

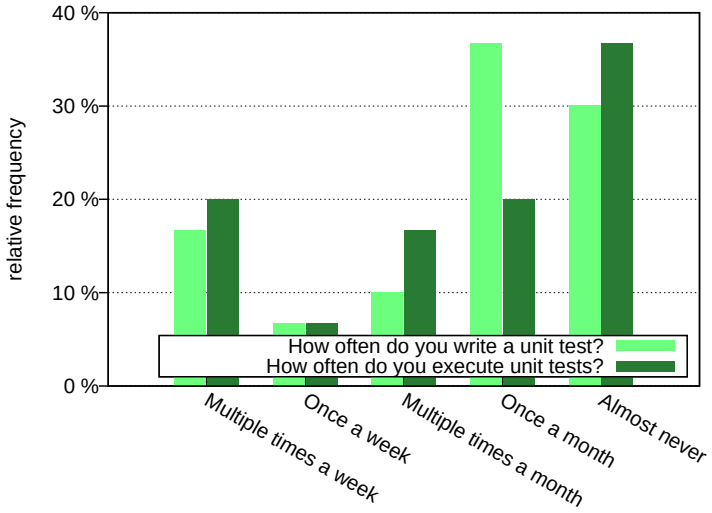


Figure 4.5: Q11-12: Frequency of unit testing.

legacy code, or if they thought that this was still good practice. The given answers are shown in Figure 4.2d. A majority of 57% (of which all participants in Delft accounted for 50%) gave the correct answer, but still a large part (almost everyone in Kuala Lumpur) chose the static method call.

- In Exact Online code, we saw examples where subclasses overrode methods from the parent class with empty implementations, a smell called degenerate functions [25] and a clear violation of the *Liskov Substitution Principle* (see Section 2.3.3). Figure 4.2e shows that slightly more than half of the participants wrongly thought that there was no objection to having a subclass deliver less functionality than its base class. About 47% knew the meaning of the LSP.

- Figure 4.3 shows how many design patterns were recognized by the participants. This picture gives a slightly distorted view, because people could choose patterns from a list, linking each UML diagram with one of the given patterns, which meant that it was possible for them to guess. Most persons who had all patterns correct, often only knew 3 or 4 patterns and guessed the other ones based on what was left. The Singleton and Factory patterns were recognized most often. All people in Delft recognized Singleton, a simple pattern that was already in use within Exact Online. However, 9 out of 20 persons in Kuala Lumpur did not recognize this pattern, which makes clear that there is a difference in knowledge between developers from the Netherlands and Malaysia.

- Figure 4.4 shows the distribution of preferred maximum size of classes and methods. There are no generally accepted rules for class and method length. In *Clean Code* it is simply advised to keep functions as small as possible and that a function should hardly ever be longer than 20 lines and that it can be much smaller most of the time [26]. Fortunately, most participants chose 10 or 20 lines as their guideline. People had more difficulty to think of a guideline for class size, as 20% did not give an answer. Also, a significant percentage of 23% thought classes of 500 or even 1000 lines were still acceptable.

- People's experience with unit testing varied a lot, as shown by Figure 4.5. The majority wrote or executed unit tests only once a month or even less frequent.

From the observations we can conclude that many participants thought in a more procedural way and that some of the concepts on which the refactorings were founded would be new to them.

4.3 Small Refactorings (#1-#3)

In this section, we cover the experiments that were based on the small refactorings.

4.3.1 Refactoring #1: RejectionNotifier

The code of the RejectionNotifier component is listed in Section 3.3.1. The assignment was to change the code so that e-mails would be sent in the language of the recipient. The strings containing the body and subject of the e-mail were formatted with `term.Format()`. The `term` object is used throughout Exact Online for translation services and it contains a method `setLanguage()`. The solution to the assignment, for both the original and refactored code, was to add the line `term.SetLanguage(person.Language)`.

Since the change was the same in both the original and refactored code, the only difference was the location where the line was to be added. In the refactored code, it had to be placed in the small, extracted method `SendEmail(person : Person, mailer : ExactSmtpMail)`. In the original code, people had to look for the part where e-mails were sent within the large function.

The results are shown in Figure 4.6a. Contrary to our expectations, most developers required more time with the refactored code than with the original code. Another observation is that there is a lot of variance in the data. One reason for that is that some developers did not know how to change the language in the framework and found out late. They could have found this by simply inspecting the interface of `Term` and seeing the method `setLanguage()`, but not everyone thought of that. Similarly some found that the `Person` object contained a property `Language`, but did not know what to do with it. For those reasons, 4 out of 15 developers that had the original code and 3 out of 15 developers that had the refactored code had to give up, because they could not find the solution.

There was a statistically significant difference in completion times between the groups with original and refactored code (Mann-Whitney-Wilcoxon test[1], $W = 28$, $p = 0.0179 < 0.05$). However, the completion times were higher rather than lower for the developers with refactored code. We did not find any evidence for our first hypothesis that the coding task would be finished earlier with refactored code than with the original code, so we had to *reject* it.

4.3.2 Refactoring #2: PurchaseOrderTools

A similar task was designed for the PurchaseOrderTools class. The results in Figure 4.6b show that the task was finished significantly faster with the refactored code. About 75% of the developers with refactored code finished the task before 25% of the developers with the original code did.

There was a difference between the tasks for RejectionNotifier and PurchaseOrderTools that might explain why PurchaseOrderTools performed so well. The assignment for PurchaseOrderTools already contained a partial snippet of the code that the developers had to insert. One had to throw an exception under certain conditions. The code for checking those conditions was given to the participants. Because the code was almost given and the assignment was very straightforward, most developers spent their time only at understanding the code to see where the code had to be inserted. This caused much less noise in the results. So

[1] A non-parametric test was used because the sample sizes were not large enough to assume that the completion times were normally distributed.

the difference in time between original and refactored code for PurchaseOrderTools is really the difference between understanding original and refactored code. In this experiment, only one person with original code gave up, because the person did not know how to throw an exception. This low number also demonstrates that the task was straightforward for most people.

There was a statistically significant difference in completion times between the groups with original and refactored code (Mann-Whitney-Wilcoxon test, $W = 176.5$, $p = 0.0012 < 0.05$). Therefore, we have enough evidence to *accept* our second hypothesis that the coding task for PurchaseOrderTools would be finished earlier with refactored code.

4.3.3 Refactoring #3: ContractProlongation

The task designed for ContractProlongation was to find and fix a bug. The bug description said that a certain property (`ProRata`) was not always set properly. This value was assigned in a few places as `contractLine.ProRata = proRataCalculator.CalculateProrata(..)`. However, we removed the assignment at one place, so that the statement read `proRataCalculator.CalculateProrata(..)` without the assignment, so that the return value of the function would not be stored. One could fix this by writing the actual assignment. While the bug that we inserted could be considered silly, the participants agreed that forgotten assignments do happen. It is something that can be easily overlooked. Thus it represented a plausible scenario.

The results for the experiment with the ContractProlongation component, shown in Figure 4.6c, are similar to the results for RejectionNotifier. Both experiments showed roughly the same absolute difference between the original and refactored code. In both the control group as well as the experimental group, 3 out of 15 people had to give up, because they could not finish the assignment.

There was a statistically significant difference in completion times between the groups with original and refactored code (Mann-Whitney-Wilcoxon test, $W = 34$, $p = 0.0273 < 0.05$), but the completion times were higher rather than lower for the developers with refactored code. Again, just as in the case of RejectionNotifier, we did not find any evidence for our third hypothesis that the coding task for ContractProlongation would be finished earlier with refactored code than with the original code, so we had to *reject* it.

A possible explanation for this result is that we could only extract pieces of code to a separate method by requiring that variables from the calling methods were passed to the extracted methods. The flow of method arguments and return values might have been more difficult to understand for most developers than a linear flow in a large method.

4.4 Medium Refactoring (#4: FinancialEntryTools)

For FinancialEntryTools, we had introduced a bug by simply negating three similar boolean conditions. In the original code, we flipped the boolean condition, shown below, by replacing <> with =.

```
CInt(.Prop("AllowDeleteSourceGreaterThan100").Value) <> 0
```

The equivalent condition in the refactored code was simply the boolean `AllowedToDeleteSourceGreaterThan100` in the class that was added. We simply wrote a `Not` in front of it, at the three places where the boolean was used. We presumed that the bug would be overlooked much more easily in the original code, where the cast to an integer and the comparison with zero introduced a lot of cognitive load. In contrast, the boolean in the refactored version was much more explicit and did not need any conversion.

The developers in the experimental group also received a set of unit tests. We made sure that we did not include a failing test case that exposed the bug, for that would have made finding the bug trivial. The developers were instructed that they could use the unit tests how they wanted, for example, to gain a better understanding of the functionality and to be able to verify what *did* work. Also, when they changed the code, they could use the tests to verify that they did not break anything that was working correctly.

The results are shown in Figure 4.6d. Four persons with the original code and three with refactored code gave up, because they could not locate the bug. During the experiment, we observed that the provided unit tests had a positive impact on the results.

The two persons that finished earliest found the bug by writing a failing test case based on the bug description. The given unit tests, which were small and easy to understand, had encouraged them do add a test. When they had a failing test case, they used the debugger in the IDE to run the test case, so that they found the actual bug by stepping though the code. It was remarkable that these persons used the exact same process independently of each other and thereby found the bug very quickly. Both persons indicated on the questionnaire that they executed unit tests on a daily basis and that they wrote tests 'multiple times a week', so they were more experienced with unit testing than most participants (cf. Figure 4.5).

Indeed, we found weak statistical differences between the results of those with refactored code who wrote tests multiple times a week and those with refactored code who wrote tests less often (Mann-Whitney-Wilcoxon test, $W = 18$, $0.05 < p = 0.1212 < 0.15$). To eliminate the fact that this difference was caused by another cause (e.g., those that often write unit tests might also be more experienced software engineers or have better problem-solving skills), we applied the same statistical test to the control group where we found no significant differences in the distribution (Mann-Whitney-Wilcoxon test, $W = 14$, $p = 0.7758 > 0.15$). Therefore, it is not far-fetched to state that if all participants had more experience with unit testing, the difference in time between those working with original and refactored code would likely have been bigger.

While the developers with the original code had only one method to focus on, the developers with the refactored had to jump from the original method to the extracted class that implemented an interface. While one person specifically said that he could not quickly locate the class that implemented the interface, the other developers found the extracted class quite soon, so the addition of an extra class was not a disadvantage.

Both with the original and refactored code, some participants required almost 30 minutes to finish the assignment. In both cases it was possibly to overlook the bug. However, with the refactored code, results started to arrive much earlier. Unit tests proved to be very beneficial to experienced developers.

The Mann-Whitney-Wilcoxon test ($W = 72$, $p = 0.7399 > 0.05$) indicated that it was unlikely that the experimental and control group had different distributions with respect to

completion times. This can be explained by the fact that there is not a major shift of the data between the two groups. Figure 4.6d shows that the medians are roughly equal. However, the figure also shows that the interquartile range is much smaller for the refactored group. This means that the task was completed earlier by most people, when they had refactored code. Also, the test only considers the ranking of the data and therefore does not add any significance to the fact that two persons finished *much* earlier than most others due to their use of the unit tests.

Therefore, despite the outcome of the Mann-Whitney-Wilcoxon test, we argue that we produced enough supplementary evidence, to *accept* our fourth hypothesis that the task can be finished earlier with refactored and unit tests. Especially the fact that the use of unit tests led to such advantages should not be disregarded.

4.5 Large Refactorings (#5-#6)

In this section, the experimental results for the large refactorings are presented.

4.5.1 Refactoring #5: ERDomain

For the task designed for ERDomain, we re-introduced a bug that had actually occurred on the production server of EOL. The bug description that was given to the participants was that an `ArgumentException` was thrown with the following message: "An item with the same key has already been added". In other words, it was attempted to put a domain in a dictionary for the second time. Recall from Section 3.5.2 that ERDomain interacted either with XML files or with the database. We gave the participants the hint that the bug was *only* triggered when ERDomain loaded domains from XML files.

The results in Figure 4.6e show a drastic difference in time between original and refactored code in favour of the original code (Mann-Whitney-Wilcoxon test, $W = 36$, $p = 0.0374 < 0.05$). The developers solved the bug much faster when they had the original code. However, looking beyond the mere data, we noticed an important difference in quality between the implemented bug fixes.

With the original code, the developers made a 'quick fix' at the place where all objects were added to the dictionary, to check if the object was already there. This check was placed right before the addition to the dictionary, which could throw the exception. This means that any inconsistency, that was triggered by loading from XML files, was caught at the very last moment before it caused an exception. The change was made near the *symptom* of the bug and the fact that the bug was only triggered when loading from XML files was disregarded.

However, the solutions made for the refactored code were much better. While an equivalent quick fix would have been possible in the refactored code, the developers targeted the `ERDomainsFromFileLoader` that took care of loading domains from XML files and made the bug fix at the highest possible level. Here, the bug was fixed near the *root cause*. Ergo, while it took time for developers to understand the relations between classes, *the refactored code led to better solutions*.

In total, 2 persons with original and 2 persons with refactored code gave up. Furthermore, one person with original code had to be disqualified, because he recalled the original

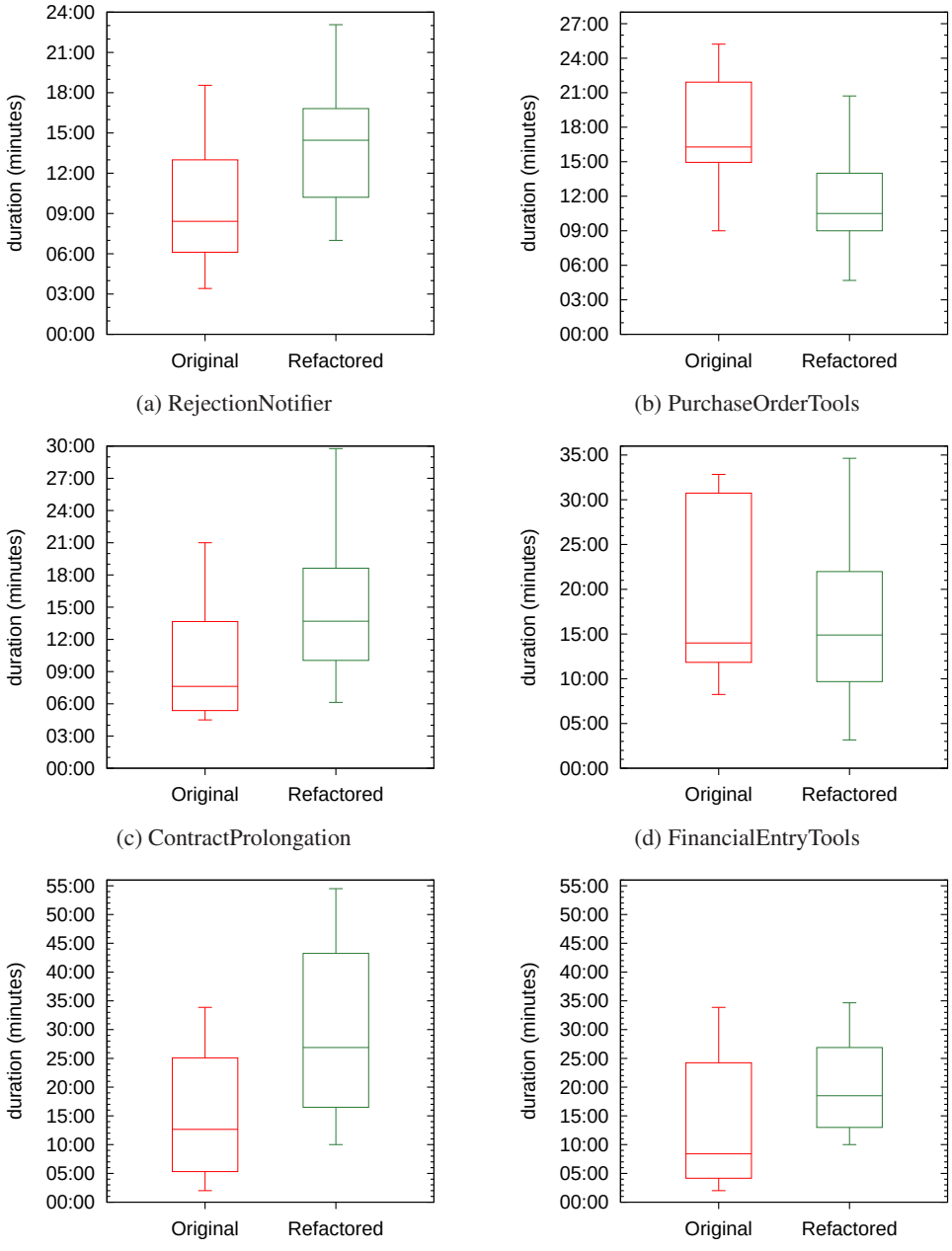
bug. One person with refactored code also was ignored, because of process errors while the experiment was carried out. So there were 12 data points for the experimental group as well as the control group.

An important observation is that the data lead to a much more nuanced view, if only the data gathered in Kuala Lumpur are considered. See Figure 4.6f. In fact, in that case the difference is statistically insignificant (Mann-Whitney-Wilcoxon test, $W = 18$, $p = 0.1520 > 0.05$). The experiment for ERDomain was the only experiment where there was such a difference between the data collected in Delft and Kuala Lumpur, even though the same procedure was followed on both locations. Note that the results for the original code were roughly the same on both locations; the difference only applies to the refactored code. We can think of a few possible reasons.

First of all, some developers in Delft had worked on ERDomain before, even though that was more than a year ago. It turned out that one participant recalled the actual bug that had occurred on the production server. He was disqualified and did not participate in the experiment. The other participants did not know about the bug, but because ERDomain fell under their responsibility, they possibly spent more time on understanding the complete refactoring, to see the bigger picture. LaToza et al. found that developers have a very strong notion of *team code ownership*: 92% of developers they interviewed, made a clear distinction between the code of their own team and the code owned by other teams [23]. The participants in Kuala Lumpur knew nothing about ERDomain and would not have spent more time on understanding the classes than necessary.

Secondly, the average age of participants was higher in Delft than in Kuala Lumpur. While this means that the participants in Delft were more experienced, it also means that most of them had originally programmed in a procedural paradigm in their early professional career. The younger developers in Kuala Lumpur might have had a more receptive mind. On the other hand, the participants in Delft answered more questions in the questionnaire about object-oriented programming correctly than the participants in Kuala Lumpur did.

However, even when only considering the results from Kuala Lumpur, the results do not support our fifth hypothesis that the coding task for ERDomain can be finished faster with refactored code. It might take more time to understand the relations between classes that emerge from a large refactoring, especially if a developer is used to large classes. We do note that the quality of the solution was better when refactored code was used, but our hypothesis focused only on productivity. The change that was required in the code was very small, so that we mostly measured the required effort to understand the code. However, if we had designed larger tasks, then we might have measured maintainability. We presume that the refactoring of ERDomain more clearly pays off when making large changes. For instance, the applied Strategy pattern makes it easy to change the underlying data source, while for the original code the most natural solution would be to duplicate code in quite some places.



(a) RejectionNotifier

(b) PurchaseOrderTools

(c) ContractProlongation

(d) FinancialEntryTools

(e) ERDomain (results from Delft and KL)

(f) ERDomain (results from KL only)

Figure 4.6: Experimental results.

4.5.2 Refactoring #6: FinTools

For FinTools we had designed a task that was quite similar to the one for FinancialEntryTools. The developers had to fix a bug that we had inserted by negating a boolean condition.

In the refactored code, we had changed `If editingExistingEntry Then` to `If Not editingExistingEntry Then`. Equivalently, we changed `ElseIf Tools.IsGuidEqual (vEntryID, EntryID) Then` to `ElseIf Not Tools.IsGuidEqual(vEntryID, EntryID) Then` in the original code. The boolean condition was much more expressive in the refactored code compared to the original code, because `editingExistingEntry` acted as an explanation for the fact that two ID's were compared. This explanation was lacking in the original code.

However, it turned out that this task was simply too difficult to solve, as 9 persons with original code and 10 persons with refactored code gave up. The original method contained a very complex algorithm. While the algorithm was made much more readable in the refactored code, we did not change the actual functionality, the external behaviour that was expected by the rest of the system. The functionality was inherently difficult to understand, something that even a refactoring could not solve.

However, it is worth mentioning that, in the case of the refactored code where we had added multiple classes, most participants focused on the small method of 12 lines that contained the high-level description of the algorithm. The refactoring had improved understandability to such an extent that at least the search scope was reduced. Still, most participants did not get any further than that and simply overlooked the bug. To conclude, we did not get any evidence to support our sixth hypothesis.

4.6 Summary

This chapter has described the set-up of the six experiments with refactored code that were performed in Delft and Kuala Lumpur. Each experiment dealt with another component from Exact Online and with another coding task. Before the experiments started, all test subjects had filled in a survey with questions on object-oriented design and unit testing, which demonstrated that quite some participants did not understand important concepts of object-orientation. For each experiment we formulated a hypothesis that we expected that the participants would solve a change task faster on refactored code than on original code.

In the remainder of this chapter, we presented our experimental results. It followed that only two of our six hypotheses were confirmed. The next chapter describes what those outcomes mean for our research questions.

Chapter 5

Discussion

In the previous chapter we presented our experimental results. Three experiments involved small refactorings, one involved a medium refactoring, and two experiments involved large refactorings. The results were very mixed and therefore only two of our hypotheses were supported by the results. Therefore, we are not in the position to state that having refactored code directly leads to an increase in understandability; the matter is not that straightforward. In this chapter, we first do some further observations, and then relate the results to our research questions. The results from experiment FinTools will be disregarded, as no meaningful data came out of that experiment.

5.1 Influence of Tasks

We observed that the specifics of each coding task caused a lot of variance in the results. Some participants had more knowledge and experience than others. Some missed specific knowledge of the Exact Online framework, which we had assumed they knew, so that they did not know how to add the desired behaviour. This also caused quite some participants to give up, as shown in Table 5.1. The difference in participants who did not finish between original and refactored code is negligible, indicating that there were difficulties inherent to the tasks, which did not depend on whether the code had been refactored or not.

	Participants who did not finish	
Experiment	Original code	Refactored code
RejectionNotifier	4/15	3/15
PurchaseOrderTools	1/15	0/15
ContractProlongation	3/15	3/15
FinancialEntryTools	4/15	3/15
ERDomain	2/14	2/14
FinTools (discarded)	9/15	10/15

Table 5.1: Participants who did not finish.

It is noteworthy that the experiment where only one person did not finish, was also the experiment of which the outcome most strongly favoured refactored code. Arguably, the experiment of which the task difficulty added the least noise to the results, measured the understandability of the source code in a sounder way than the other experiments.

The influence of tasks on the experiments is also acknowledged by Ng et al. [37] who performed similar experiments and stated that the successful implementation of change tasks could be heavily influenced by the nature of the task. This can make it difficult to generalize experimental results. On the other hand, a change task needs to be challenging enough so that developers have to spend some time in order to comprehend the program. These requirements are conflicting [37], hence we conclude that the influence of tasks on experimental results is difficult to avoid.

5.2 Habits

In Kuala Lumpur, when the original and refactored code were shown side-by-side after each experiment, most developers appreciated the refactored code and preferred it over the original code. The fact that not all experiments supported the intuitive feeling, which was also felt by the participants, that tasks would be solved faster with refactored code, can also be explained by the habits of the developers. The participants were used to working with long, procedural methods and thus were also trained by experience to skim through long methods to understand its structure. When working with small methods, on the other hand, one might have to jump around between different methods to understand a certain feature. Of course, the advantages are that small functions are reusable, better testable and that their names act as higher-level descriptions of their functionality. But if one is used to long methods, then one might need more time to adjust to this different way of working. Our experiments presumed that having refactored code would lead to instant benefits. There was no time for participants to adjust to working with clean code. Experiments that would let developers work with refactored code for a longer period might lead to different results.

5.3 Similar Results in Delft and Kuala Lumpur

There was not much difference between the groups in Delft and Kuala Lumpur if we consider the interquartile range (25-75%) for each experiment, even though we found that the participants in Delft were generally more knowledgeable.

Both in Delft and Kuala Lumpur there were individual differences in experience between participants. However, as a whole, the groups in Delft had more experience than those in Kuala Lumpur. In Section 4.2 we saw that the participants in Kuala Lumpur answered many questions in our questionnaire incorrectly. Furthermore, in Delft participants only gave up on the FinTools experiment, whereas in Kuala Lumpur more assignments were left unfinished. However, we made clear that this related to the difficulty of the tasks, instead of whether the code had been refactored. What matters is that the differences between Delft and Kuala Lumpur did not lead to differences in favouring refactored or original code.

The experiments were first performed in Delft and then repeated in Kuala Lumpur. While the developers in Kuala Lumpur in general required more time to complete each assignment, we did not observe a significant shift of the interquartile range (25-75%) for each of the first four experiments. Similarly, Ng et al. [37] found that change tasks on refactored code were performed faster than on original code, regardless of work experience of the subjects. While we did not observe an unambiguous favour of refactored code as they did, at least we share the observation that the results were robust.

However, though we established differences in knowledge between the developers in Delft and Kuala Lumpur, they still work on the same code base. So it is not ruled out that the results could have been different if they had been exposed to more clean code in their daily work, as discussed under *Habits*.

5.4 Small Refactorings

We repeat our second research question here.

> **RQ2** *What is the difference in completion time between people that finish a change task on code that has been refactored with small refactorings, such as Extract Method, and those that finish a task on the original code?*

We set up three experiments with small refactorings. All three experiments showed statistically significant differences between the groups with refactored and originally code, though only one experiment indicated a decrease in completion times for refactored code. Therefore, we have to conclude that the difference can be positive or negative, depending on the targeted code and the specific change task. We can think of a few reasons why some experiments led to unexpected results.

First of all, not all method extractions are straightforward. Variables that are both accessed in an extracted method and used in the calling method, need to be passed into the extracted method as input variables and possibly returned as output variables [52]. We presume that this might have impacted understandability in the ContractProlongation experiment.

Secondly, the rationale for our hypothesis that extracting methods would lead to increased completion times, was based on the assumption that long methods negatively impact understandability. While this may be true in general, we should not forget that the software engineers at Exact were quite experienced with working with long methods. Through experience, they had possibly optimized their way of working under these circumstances.

Thirdly, by extracting details to helper methods, one can make the calling method read more like a story. Again, the developers at Exact were not used to this concept and hence may not have taken full advantage of such aids.

To conclude, extracting methods can sometimes lead to an increase in understandability, and to a decrease at other times. We gave a few possible explanations for this result. For methods of which their extraction could lead to a decreased understandability, there may still be other benefits, such as reusability and testability, to which our experiments did not

extend. Hence, if, by extracting a method, one can prevent code duplication or increase testability, then those benefits might outweigh a possibly decreased understandability, whereas an increased understandability would be a bonus.

5.5 Medium Refactorings

> **RQ3** *What is the difference in completion time between people that finish a change task on code that has unit tests and that has been refactored so that an extra class was added, and those that finish a task on the original code?*

We only had one experiment (FinancialEntryTools) to answer this question. We noted a significant advantage of unit tests in this experiment. Because of the quick feedback given by the unit tests, one participant required no more than 3 minutes to finish the task, whereas the first person to finish with original code required 8 minutes. Furthermore, 75% with refactored code finished within 22 minutes, while 75% with original code required up to 31 minutes. As noted in Section 4.4, if more developers had been experienced with unit tests, then it is likely that those with refactored code would all have finished much faster than those with the original code.

Considering the mixed results that we got for the small refactorings, we should be cautious to generalize this result based on just one refactoring. However, the positive effect that unit testing can have on productivity is evident. If we disregard unit testing, then there is no significant statistical difference between completion times of those with refactored and those with original code. Hence, the refactoring itself did not increase understandability as much as we had thought.

5.6 Large Refactorings

We now answer our fourth research question.

> **RQ4** *What is the difference in completion time between people that finish a change task on code that has unit tests and that has been refactored, so that responsibilities were divided over multiple classes, and those that finish a change task on the original code?*

Unfortunately, the task that we designed for the FinTools experiment proved to be too difficult. Hence, we should base our conclusions solely on the ERDomain experiment. We found that a large refactoring can increase the time required to complete a small change task. More time is required to understand how classes collaborate. We noticed significant differences between the participants in Delft, where quite some developers came from the team that was responsible for ERDomain, and those in Kuala Lumpur. We think that those participants in Delft who had worked with the original code of ERDomain in the past, took more time to understand the big picture when they received the refactored code, more than would have been necessary for completing the assignment in the fastest way. Their sense of ownership of the original ERDomain target, and the fact that they knew for certain that the

code of ERDomain was refactored because they vaguely recalled the original form, might have led them to behave differently. Possibly, the refactoring simply confused them. Still, if we only consider the more reliable results from Kuala Lumpur, we also see that developers required slightly more time with refactored code (though the difference is not significant).

Börstler et al. said that with clean code, the complexity is captured inside the design, while the units of code are simple [5]. In contrast, with 'procedural object-oriented' code, the units are complex. The term 'procedural object-oriented code' was coined by Bhatti et al. [4] to describe code that lacks an object-oriented design but is still written in an object-oriented language. While it may be true that units are simpler to understand in clean code, to understand a certain feature, one needs to understand the connections between different units, which is not always easy because of the complex design. One participant noted afterwards that he would have finished the task faster if we had provided a class diagram. A class diagram helps to increase the understandability of a complex design. While we had made a class diagram, we deliberately left it out of the experiment to avoid having yet another variable, i.e., documentation, that would affect the results, as that would have made it harder to draw strong conclusions.

While we saw a decrease in productivity with refactored code, we observed an *increase in quality* of the implemented solutions, as explained in Section 4.5.1. In other words, good quality code prevents developers from writing sloppy code. This is in line with the *broken windows theory*, a criminological theory that can also be applied to refactoring [19]. Consider a building with broken windows. If the windows are not repaired, it may encourage vandals to break some more windows, as it seems like the owner does not care. Similarly, a dirty code base makes developers feel that they can get away with 'quick and dirty' code changes, while developers might be less inclined to do so in a clean code base.

This brings us to an important point: *maintainability*. A quick and dirty fix might suffice for the moment, but it complicates maintainability. On the other hand, the implemented solutions on the refactored code were clean solutions of which the intent was clear. Not only does refactored code promote writing more maintainable code, its abstraction makes it easier to extend the functionality. For example, in the ERDomain refactoring, it is easy to add another data source, with minimal impact on the existing code, because of the implemented Strategy pattern. Ng et al. [37] performed experiments where code was refactored in such a way that it specifically supported a given change task. These tasks were larger maintenance tasks that took several days to finish. In contrast, our tasks were quite small and took no more than one hour to finish. We did not study the effect of refactored code on a large maintenance task.

To conclude, a large refactoring can increase the time it takes to finish a small coding task, though the quality of the implementation might be better. Participants have suggested that the use of design documents might compensate for the complexity of a design. A large refactoring might pay off in the longer term, where changing requirements can lead to maintenance. Also, as discussed under *Habits*, the large refactorings were quite different from the 'procedural object-oriented' code that the participants were used to. Studying large refactorings over a longer period of time could reduce the influence of programmer habits on experimental results.

5.7 Threats to Validity

This section describes threats to the validity of our experiments and how we mitigated some of those threats.

5.7.1 Internal Validity

For each experiment, the participants were not told if they had received original or refactored code. In other words, they did not know if they were part of the control group or the experimental group. Because Exact Online has such a large code base, most participants did not recall the code, when they were given original code. The only exception is that some developers in Delft recalled the original ERDomain, as discussed before. While participants could guess if code was refactored or not, it also happened that people guessed incorrectly or were unsure. Because targets came from different projects and because the refactorings ranged in size, learning to recognize which experiments were refactored and which not, was made difficult. Moreover, for small refactorings, often a class was only partially refactored.

By this approach, we aimed at avoiding the Hawthorne effect and the John Henry effect [38]. The Hawthorne effect applies when people behave differently because they know they are in the experimental group. The John Henry effect applies when people who know they are in the control group, work harder to make up for that fact, so that they exhibit the behaviour that one expects from the experimental group.

Secondly, there was no researcher bias. Both the participants in the control group and in the experimental group received the same task descriptions. No hints were given during the experiments. The researcher could only clarify the assignment if it was unclear, and approve solutions.

The presence of unit tests with the refactored code in some experiments could have triggered the novelty effect [38], so that participants spent more time with the unit tests than they would have done otherwise. However, we established that those who took the most advantage of unit tests, where those who were already experienced in unit testing. Because the experiments were timed, those who did not know how to take full advantage of unit tests, did not spend much time on looking at the tests, but quickly returned to the code under test.

Some participants could not finish all assignments. However, we demonstrated in Section 5.1 that this was not related to the dependent variable (refactored/original code) being studied, but rather to the given tasks, so this was not a threat to the internal validity.

A potential threat was that developers had to write down the starting and finishing times for each experiment themselves on a distributed timesheet. However, because we had to approve the solutions for each experiment, we could verify that times were entered correctly. We are confident that any minor differences in how participants noted their durations did not impact the eventual results.

5.7.2 External Validity

First of all, the results from our experiments should not be generalized to all software engineers in general. The circumstances at Exact, i.e., the large amount of legacy code, are

very specific to Exact. However, in this section we claim that our sample population was representative of the total population of software engineers at Exact.

First of all, there were roughly equal numbers of senior and non-senior software engineers in the experimental and control groups. Secondly, since Exact applies global software development, our sample populations were also international. The majority of participants were abroad, which is also true for the total population of software engineers. Thirdly, the 30 participants came from 14 different development teams, taking into account that different development teams work on different parts of the code base. Fourthly, there was no matching bias [38], as all participants alternately worked with refactored and original code. Finally, we established the background knowledge of participants through a survey. The variation in the answers that were given makes clear that our sample population accounts for the diversity in the total population. Though, a threat imposed by the variation in our sample is that the sample size was relatively small (two sample populations of 15 persons).

Another concern is whether or not the provided refactorings and the designed tasks can be generalized to refactoring in general. We chose targets to refactor from different namespaces to cover multiple subsystems of Exact Online. Design decisions made during refactoring can be very subjective. Therefore, the large refactorings were also discussed with a software engineer. A threat is that not all software engineers have the skills to apply similar refactorings as we performed in our research, as can be deduced from the results of our background survey (see Section 4.2).

A final threat is that the tasks that had to be performed were not very realistic. The bugs that we inserted were realistic, but when finding the cause of a bug, developers usually walk through the application top-down, starting from the web interface, using their debugging tools. In the experiments, however, developers could not use their debugging tools or verify code changes. While this was not realistic, it shortened the duration of tasks, so that we could perform more experiments given the resources that we had at our disposal. Also, we did not want to introduce too many factors in our experiments. To conclude, our sample population was representative of the software engineers at Exact, though when designing realistic change tasks, we had to make trade-offs to get the most out of the resources that we had at our disposal.

Chapter 6

Related Work

In the previous chapter we discussed the results of our experiments. Before we will present our final conclusions in the next chapter, we will first discuss some of the related work in the field of refactoring, to see how our research contributes to the field.

Refactoring can be considered a tool of the craft in agile software development. To be able to quickly react to changing user requirements, the structure of code often needs to be changed and refactoring is a formal name for that. As such, there is a wide variety of literature on refactoring.

We can distinguish three categories of related work. First of all, there is literature on what refactoring is, and *how to refactor*, mainly aimed at software developers. The most influential works in that area were discussed in Chapter 2 and we used them to prepare the refactorings for our experiments. Secondly, quite some research is devoted to studying how people refactor (e.g., [48, 50]), in order to *support and automate the refactoring process* with tools [28, 34]. It turns out that current refactoring tools in IDE's are seldom used [35, 36]. Studying how people refactor manually might lead to ideas for better refactoring tools. Finally, there is quite some empirical research that studies the *effects of refactoring* code. Since we presented such a study in this work, this chapter specifically focus on related work in the latter category, to be able to compare those studies with our own.

6.1 Effects on Productivity

Moser et al. studied the impact of refactoring on quality and productivity in a close-to industrial environment [32]. The motivation for their research was that, despite claims that refactoring would lead to increased development productivity [13], there was only limited empirical evidence.

They measured productivity as the number of added lines of code over time between development iterations. Each iteration took one or two weeks. The development teams had to implement a system from scratch in eight weeks. There were two specific user stories to refactor the architecture, so that refactoring would be a large development activity in two iterations. In this way, the productivity between the iterations before refactoring and the iterations after refactoring could be compared.

A notable observation they did, was that development did not slow down before refactoring. They had expected a decrease in productivity as the system's complexity and coupling grew. The authors did not give a clarification for their observation. We think it is reasonable that productivity did not slow down. After all, the project was still young. One can expect productivity to increase in agile development when a system is built from scratch as the system's architecture gradually becomes more pronounced and developers get more used to it. Next they observed that the productivity in the iterations directly following an iteration of refactoring was significantly higher than the average productivity of the remaining iterations. Thus they found evidence that refactoring raises development productivity in the short-term (a number of iterations).

The development team consisted of 3 students and 1 professional software engineer. As the authors indicate, their results are therefore not statistically significant. Their experiments differ from ours as they studied a project that was started from scratch. On the other hand, we studied a real system that was developed in the industry. Furthermore, the code that we refactored was legacy code with a high complexity that is unlikely to be found in a project that is started from scratch. Also, they did not consider different kinds of refactorings. Finally, they measured productivity over a longer period of time (development iterations) as the lines of code that were added in maintenance. We, on the other hand, did not measure the number of lines of code, but measured the time it took to complete a task, of which a large part was spent at program comprehension instead of writing code.

6.2 Estimating Return on Investment for Refactoring

Leitch et al. proposed a method for estimating the costs and benefits of refactoring to be able to quantify the trade-off between the up-front cost of refactoring and the expected downstream savings [24]. As part of that method, a detailed re-structuring plan is made that identifies new and modified procedures in the design of the system. This plan provides the basis for estimating the costs of refactoring. The expected difference in maintenance is based on the differences between the original and the revised control and data dependency graphs. Their assumption is that for any single maintenance task, a substantial part of the cost is the cost of the required regression testing after the modification is completed [24]. For each procedure in the system, a list should be made of additional procedures that need to be re-tested, if it were changed. Doing so for the control graphs of the current code and the proposed code, gives an indication of the difference in regression testing for any maintenance activity after refactoring. The estimated refactoring costs and maintenance costs, when combined, provide an estimate for the return on investment (ROI). The authors tested their method with two case studies.

However, since the authors proposed this model in 2003, we see that some assumptions underlying their method are no longer true. Firstly, creating a comprehensive re-structuring plan up-front, which should include all changed and new procedures, conflicts with the values of agile software development. It ignores the fact that additional insights while doing the actual refactoring, might result in the refactoring taking another direction (think of *emergent design* [26]). More importantly, basing all future maintenance activity on just manual

regression testing is superseded by the fact that regression testing is often performed by automated tests for a large part.

In 2006, Schofield et al. [42] applied the method of Leitch et al. on an open-source system to see if the estimated ROI would support refactoring decisions made in an actual software project. Via repository mining they found changesets that involved refactoring. For those changes, they calculated the estimated ROI. The authors found that for 5 of the 6 changesets of which more than 50% of the changes involved refactored classes, the estimated ROI was indeed positive. Of course, this does not say if the estimates were correct and whether or not a return on investment had really occurred.

6.3 Effects on Reported Defects

Quite some studies indicated that it is likely that less defects are found in a component after it has been refactored. Kim et al. [20] studied a very large, system-wide refactoring performed by Microsoft on Windows. They found that refactored modules experienced a significant reduction in the number of defects in Windows 7 compared to Windows Vista.

Ratzinger et al. [40] studied five open-source projects and found that the number of software defects decreased, if the number of refactorings increased in the preceding time period. Geppert et al. [16] also found a decrease in reported defects after refactoring an industrial legacy system. Additionaly, they noted a decrease in the effort required to change the code after refactoring, though this was based on an estimate.

At Exact, De Swart [45] studied if re-engineering and unit testing a bug-prone component would lead to a difference in reported bugs for that component. Though initial results provided a first indication that the number of reported bugs would decrease, there was not enough data to give a conclusive answer.

6.4 Supporting Change by Refactoring

Ng et al. [37] performed a research that was similar to ours in set-up. Like us, they refactored components and performed experiments where test subjects had to finish a change task on the refactored or unrefactored code. Unlike us, they found that all change tasks were finished faster with refactored code. However, there were some important differences with our research with respect to their set-up.

First of all, the targeted system was an open-source project of which the code was already quite clean. In fact, the project even served as a showcase for the use of design patterns. Then, the refactorings had intentionally been done in such a way that they *supported* the change task. Thus, the refactoring could be considered a part of the maintenance task, which the test subjects had to finish. Furthermore, the given change tasks took multiple days to solve, as they required participants to implement major features, so the tasks were quite different from our small tasks. Because the refactorings supported the change tasks, the researchers added the time they had used during refactoring to the durations of the test subjects. So, they measured the time it took to complete a change task, if one performs a refactoring that would help the change task (*refactoring approach*), compared to if a task

was performed by directly revising the program (*direct approach*) without refactoring to make the change easier. Thus, their research had a different angle than ours.

The research from Ng et al. was statistically significant as they had a large number of test subjects. They had 55 inexperienced subjects who where enrolled in an undergraduate-level programming course and had no formal work experience. Also, they had 63 experienced subjects with at least one year's work experience in the industry who where enrolled in a post-graduate software engineering course.

The authors found that the time spent while using the refactoring approach is much shorter than while using the direct approach even after including the duration of the refactoring process in the refactoring approach. Furthermore, they found that there was not much of a difference in the correctness between changes implemented with the direct approach or with the refactoring approach.

Our refactorings were not targeted specifically at the tasks that were given to test subjects. If one refactors a legacy component, then developers should benefit from it each time they need to change the code, assuming that a refactoring improves understandability and changeability. Therefore, from that perspective, we did not add the time we took for refactoring into the equation.

6.5 Effects on Code Metrics

The effects of refactorings on code metrics have also been studied quite extensively (e.g., [1, 33, 32, 44]). While many studies demonstrated that refactoring led to improvements in code metrics, Stroggylos et al. [44] found the opposite when they studied a few open-source projects.

As an example of a study that indicated benefits of refactoring, Usha et al. [47] studied the effects of refactorings on software maintainability, reusability and understandability. They evaluated understandability by using code metrics for complexity and modifiability. From their experimental results they found that refactorings led to improvements in reusability, understandability and maintenance. However, we argued in Chapter 2 that expressing understandability in code metrics might give a too narrow view. In our study we measured understandability by actually measuring how long people took to comprehend the code. Therefore, to limit our scope, we do not further discuss any literature that focuses on the relation between refactoring and code metrics.

6.6 Summary

A recurring theme in the literature is that there is a lack of solid, empirical, and quantitative evidence of the benefits of refactoring on the productivity of developers, despite claims saying so. We found a small number of studies that performed an empirical research where productivity or task completion times were measured. Both studies let test subjects perform large change tasks that took more than a day to complete. Research indicated that change tasks were performed faster when code was refactored in anticipation of the change.

Compared to that, the tasks that we designed were quite small. Because of that, the largest part of the time required to finish change tasks was taken up by trying to understand the code. Thus, while other studies focused on differences in maintainability, we focused more on understandability. We have not found evidence that such a study was done before.

Chapter 7

Conclusions and Future Work

The main goal of the current study was to determine if refactoring would lead to such an increase in code understandability, that developers would be able to complete coding tasks faster than otherwise. This chapter presents the contributions and conclusions of this research. The first section lists our contributions. The second part summarizes the conclusions to our research questions, before the third parts moves on to the conclusion to the main research question. Finally, we present some recommendations for future work.

7.1 Contributions

This work has provided the following contributions.

- The empirical findings in this study provide a better understanding of how refactoring can lead to different results with respect to code understandability in an industrial context.
- The study has provided a short literature survey on quantitative and qualitative notions of code quality.
- Furthermore, this study has given an overview of the different effects of refactoring that have been studied in empirical research.

Additionally, the following contributions are of importance to Exact.

- To increase the knowledge on refactoring and technical debt, we constructed a refactoring training of which we hosted the first session in Malaysia. Exact's training department will extend this training and train all software engineers at Exact worldwide, so that this knowledge will lead to a more maintainable code base.
- During this research we provided several refactorings of Exact Online components. Of these, the re-engineered version of `ERDomain`, which was the largest refactoring, has been merged to production, as the responsible team recognized its increase in maintainability.

7.2 Recap of Research Questions

As part of our research methodology we defined four research questions that would bring us closer to an answer on the main research question. We now review each of these questions.

Research Question 1.
In what ways is code quality described?

Code quality is sometimes described quantitatively by different (aggregations of) code metrics. These can be useful to give a quick overview of the health of a code base. At the same time, they cannot capture every aspect that developers associate with code quality. Code quality is often also described in informal terms such as 'clean' and 'dirty'. Developers have formed principles and guidelines that are associated with high quality code, so that code quality can be described as the level of adherence to these principles.

The term 'clean code' has been made popular by Robert C. Martin who is a strong advocate of code that is self-explanatory, that can be read like a story, and that describes its intent. This can be accomplished by having a decomposed design with small classes with a single responsibility and small functions that 'do one thing'. In this study, the refactorings that we performed and whose effects were evaluated, were largely motivated by Martin's clean code guidelines, as well as his S.O.L.I.D. principles, which are generally accepted principles of agile software design, of which the Single Responsibility Principle is well-known.

Research Question 2.
What is the difference in completion time between people that finish a change task on code that has been refactored with small refactorings, such as Extract Method, and those that finish a task on the original code?

This study has shown that extracting methods can lead to both increases and decreases of completion times. It depends on the targeted code and the specifics of the change task. Furthermore, we should acknowledge the fact that the developers were used to reading long methods in a linear flow. Jumping back and forth between extracted methods might have been uncommon for them, so that the time needed to adapt to this style might have partially undone the advantages of small methods with respect to understandability.

Research Question 3.
What is the difference in completion time between people that finish a change task on code that has unit tests and that has been refactored so that an extra class was added, and those that finish a task on the original code?

Our experiment with one such refactoring suggested that the refactoring alone, which resulted in better readable code, led to just a small decrease in completion times, which was not statistically significant. However, the most obvious finding was that the use of unit tests *did* lead to significant decreases. In our research, developers were given code containing

a bug and a small, incomplete unit test suite for that component, which lacked a test that exposed the bug. A couple of developers were motivated by the presence of tests to add a test case based on the bug description to find the cause.

So, when one has to fix a bug, one can find the cause of the bug much faster when one writes a failing unit test that reproduces the scenario in order to expose the bug. Furthermore, this test can verify the correctness of the bug fix, thereby making it possible to fix the bug in a test-driven way. Being able to write a unit test implies that the code must be unit testable. Hence, the largest benefit of the refactoring that this study evaluated for this research question, was not the fact that it improved the understandability of the code, but rather that it resulted in testable code. However, these findings are limited by the fact that we only evaluated one refactoring that fit the scope of this research question.

Research Question 4.
What is the difference in completion time between people that finish a change task on code that has unit tests and that has been refactored, so that responsibilities were divided over multiple classes, and those that finish a change task on the original code?

We evaluated the re-engineering of a large class in order to answer this research question. The first finding from this research was that the completion times did not decrease in the case of refactored code. Our observations suggest that developers might need more time to understand how classes collaborate, especially if they are used to legacy code that lacks an object-oriented design. However, one of the more significant findings was that the quality of the implemented change was higher when developers worked with refactored code. Clean code prevented developers from accepting a 'quick and dirty' code change. The separation of concerns made developers make the change at the right location. This evidence suggests that increases in maintainability through refactoring might lead to more time savings than increases in understandability do.

7.3 Main Conclusions

Returning to the main research question, listed below, which we posed at the beginning of this study, it is now possible to state that, contrary to our expectations, refactoring does not necessarily decrease the time needed to understand a piece of code to complete a small coding task.

Main Research Question.
How much difference in time is there when performing a small coding task on refactored code, compared to if the code had not been refactored?

The results from several experiments with refactored code at Exact have shown that both increases and decreases in completion times are possible, though increases were more

common. Even though the difficulty of some of the designed coding tasks caused quite some variability in the data, for most of our experiments there were statistically significant differences between developers with refactored code and those without.

According to proponents of clean code, clean code (with small methods and small classes) significantly improves the understandability of code by making code almost self-explanatory. We therefore presumed that replacing parts of legacy code with clean code would *instantly* lead to benefits. However, our findings suggest that the matter is not that simple. The developers at Exact were used to long code structures and a minimum of abstraction. Working with smaller components requires a different way of reading code. Furthermore, our results indicated that many developers lacked important knowledge of object-orientation, which could make it time-consuming for developers to understand an abstract design. An implication of this is the possibility that developers will complete tasks faster with refactored code in the longer term, when they have gained more experience with clean code. These findings would seem to suggest that the short-term benefits of refactoring depend on the context, such as the extent to which developers are used to a certain coding style.

Besides understandability, which this study mainly focused on, there are other claimed benefits of refactoring, such as increased reusability, maintainability and testability. Therefore, while our study has shown that refactoring does not necessarily increase understandability in the short term, those other aspects might still make refactoring beneficial. Other studies have given evidence of how productivity increases over multiple development iterations as a result of refactoring. One of the findings of this research supports the idea that clean code spurs on developers to keep the code clean. Another significant finding of this study was that the use of unit tests, which may require refactoring to first make the code testable, could drastically reduce the time needed to find and fix a bug. Therefore, in cases where refactoring decreases understandability in the short term, there may be other benefits. Furthermore, our study has not ruled out the possibility that increases in understandability are more likely to show up in the long term.

A recommendation that follows from the conclusions listed above is that one should not just refactor code with the intention to improve its understandability, but one should have additional motives, such as to ease coming maintenance or to increase the testability of a component. This view is supported by Pizka [39] who claimed that refactoring is of little help for retrospective restructuring of larger code bases for quality reasons without the refactoring being aimed at supporting a concrete change.

This research has extended our knowledge of the effects of refactoring on code comprehension. Though, the findings in this report are subject to a few limitations. The current study has only examined understandability on a very short term and in a very specific context, i.e. an industrial organization dealing with legacy code. Notwithstanding these limitations, this study has demonstrated that an instant increase in understandability is not always obvious.

7.4 Future Work

A small number of studies have evaluated the effects of refactoring with regard to productivity and we added our share by studying those effects in a very specific context. Having concluded our research, we now provide some recommendations for further research work.

- It would be interesting to assess the understandability of refactored code in comparison to unrefactored code, when test subjects come from another context and have no experience with the original code base. This study suggested that the increase in understandability might have been limited by the fact that test subjects were used to working with legacy code and thus would have optimized the way they read code for that specific situation.

  Performing similar experiments as in this study with test subjects that have no connection to the original code base nor even to the style of the original code base, could possibly lead to more apparent decreases in the time required to understand code. If that is the case, it would mean that newly employed developers require less time to become familiar with an existing code base, so that they can start delivering results earlier.

- Secondly, a future study at Exact could investigate the effects of refactoring on productivity when developers have to perform larger tasks that span one or more development iterations. Such research could evaluate if the increases in productivity on a longer term, as reported by other studies, also apply when working with legacy code. A challenge of such research, which might explain why only few have studied this in an industrial context, is that it requires many resources.

- Thirdly, this study has modestly demonstrated that unit testing can have a positive effect on productivity. Quite some case studies have already investigated the effects of test-driven development. However, the effects of unit testing on productivity when specifically working with legacy code could use some more investigation. For Exact, it would be helpful to have this effect quantified.

  For our study, it was too soon to investigate the effects of unit testing in more detail, as quite some developers struggled with making legacy code testable. However, now that all developers at Exact will receive training on refactoring as a result of this study, the time may soon be ripe to examine the effects of unit testing in more detail.

Bibliography

[1] Mohammad Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and software technology*, 51(9):1319–1326, 2009.

[2] 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.

[3] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[4] Muhammad Usman Bhatti, Stéphane Ducasse, and Marianne Huchard. Reconsidering classes in procedural object-oriented code. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 257–266. IEEE, 2008.

[5] Jürgen Börstler, Michael E. Caspersen, and Marie Nordström. *Beauty and the beast–toward a measurement framework for example program quality*, 2007. Department of Computing Science, Umeå University, Sweden.

[6] Andrea Capiluppi, Maurizio Morisio, and Patricia Lago. Evolution of understandability in oss projects. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 58–66. IEEE, 2004.

[7] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, 2009.

[8] Ward Cunningham. The wycash portfolio management system. In *ACM SIGPLAN OOPS Messenger*, volume 4, pages 29–30. ACM, 1992.

[9] David Consulting Group. *How much of a problem is technical debt and what should we do about it?*, 2013. Retrieved in June, 2014 from `http://www.davidconsultinggroup.com/media/137521/how-much-of-a-problem-is-technical-debt-and-what-should-we-do-about-it.pdf`.

[10] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Elsevier, 2002.

[11] Kasper Edwards. When beggers become choosers. *First Monday*, 5(10), 2000.

[12] Michael Feathers. *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[13] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[14] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[15] Gartner, Inc. . *Gartner estimates global 'IT Debt' to be $500 billion this year, with potential to grow to $1 trillion by 2015*, 2010. Retrieved in June, 2014 from `http://www.gartner.com/newsroom/id/1439513`.

[16] Birgit Geppert, Audris Mockus, and F Robler. Refactoring for changeability: A way to go? In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 10–pp. IEEE, 2005.

[17] James D Herbsleb and Deependra Moitra. Global software development. *Software, IEEE*, 18(2):16–20, 2001.

[18] Yiannis Kanellopoulos, Panos Antonellis, Dimitris Antoniou, Christos Makris, Evangelos Theodoridis, Christos Tjortjis, and Nikos Tsirakis. Code quality evaluation methodology using the ISO/IEC 9126 standard. *International Journal of Software Engineering & Applications*, 1(3), 2010.

[19] Joshua Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[20] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[21] Jitender Kumar Chhabra, KK Aggarwal, and Yogesh Singh. Code and data spatial complexity: two important software understandability measures. *Information and software Technology*, 45(8):539–546, 2003.

[22] Gerben Kunst. *Programming language popularity chart*, 2013. `http://langpop.corger.nl/`. Retrieved in May, 2014.

[23] 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.

[24] Robert Leitch and Eleni Stroulia. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Software Metrics Symposium, 2003. Proceedings. Ninth International*, pages 309–322. IEEE, 2003.

[25] Robert C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[26] Robert C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[27] Aditya P Mathur. *Foundations of software testing*. China Machine Press, 2008.

[28] Erica Mealy, David Carrington, Paul Strooper, and Peta Wyeth. Improving usability of software refactoring tools. In *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, pages 307–318. IEEE, 2007.

[29] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[30] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[31] Microsoft. *Maintainability Index Range and Meaning*, 2007. Retrieved in April, 2014 from `http://blogs.msdn.com/b/codeanalysis/archive/2007/11/20/maintainability-index-range-and-meaning.aspx`.

[32] 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.

[33] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. Does refactoring improve reusability? In *Reuse of Off-the-Shelf Components*, pages 287–297. Springer, 2006.

[34] Emerson Murphy-Hill and Andrew P Black. Breaking the barriers to successful refactoring. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 421–430. IEEE, 2008.

[35] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18, 2012.

[36] Emerson R Murphy-Hill and Andrew P Black. Why don't people use refactoring tools? In *WRT*, pages 60–61, 2007.

[37] TH Ng, SC 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.

[38] Anthony J Onwuegbuzie. Expanding the framework of internal and external validity in quantitative research. 2000.

[39] Markus Pizka. Straightening spaghetti-code with refactoring? In *Software Engineering Research and Practice*, pages 846–852. Citeseer, 2004.

[40] Jacek Ratzinger, Thomas Sigmund, and Harald C Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 35–38. ACM, 2008.

[41] Robert D. Ryne. Advanced computers and simulation. In *Particle Accelerator Conference, 1993., Proceedings of the 1993*, pages 3229–3233. IEEE, 1993.

[42] Curtis Schofield, Brendan Tansey, Zhenchang Xing, and Eleni Stroulia. Digging the development dust for refactorings. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 23–34. IEEE, 2006.

[43] Margaret-Anne Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 181–191. IEEE, 2005.

[44] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring–does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*, page 10. IEEE Computer Society, 2007.

[45] Jerry de Swart. *Selecting bug-prone components to study the effectiveness of reengineering and unit testing*. Master's thesis, Delft University of Technology, 2013.

[46] Edith Tom, Aybüke Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[47] K Usha, N Poonguzhali, and E Kavitha. A quantitative approach for evaluating the effectiveness of refactoring in software development process. In *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on*, pages 1–7. IEEE, 2009.

[48] Yi Wang. An exploratory investigation on refactoring in industrial context. In *Product-Focused Software Process Improvement*, pages 185–198. Springer, 2009.

[49] Kurt D. Welker. The software maintainability index revisited. *CrossTalk*, pages 18–21, 2001.

[50] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported-an eclipse case study. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 458–468. IEEE, 2006.

[51] Aiko Yamashita and Leon Moonen. To what extent can maintenance problems be predicted by code smell detection?—An empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013.

[52] Limei Yang, Hui Liu, and Zhendong Niu. Identifying fragments to be extracted from long methods. In *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific*, pages 43–49. IEEE, 2009.

Appendix A

Questionnaire about Experience and Practices

On the following pages, the questionnaire on development experience and practices has been printed. This questionnaire was filled in by all test persons before the experiments were started. The results were discussed in Section 4.2.

Participant No. _____ *Should be filled in for you.*

Questionnaire

= exact TU Delft

Please answer the following questions. Each question has a single answer, unless otherwise stated.
Your answers will be processed anonymously.

Object Oriented Programming

1. **The Single Responsibility Principle entails that each _____ should have a single responsibility.** *Choose the answer that should be filled in.*

   - ☐ I don't know.
   - ☐ method
   - ☐ class
   - ☐ project/assembly

2. **A _____ should do one thing.** *Choose the answer that should be filled in.*

   - ☐ I don't know.
   - ☐ method
   - ☐ class
   - ☐ project/assembly

For answering the next questions, see the following piece of code. The class `SimpleCode` has a method to encode characters by incrementing them (e.g. incrementing `"A"` by one, yields `"B"`).

Class definitions:

```
Public Class SimpleCode

  Public Overridable Function Encode(character As Char) As Char
    Return Chr(Asc(character) + 1)
  End Function

End Class


Public Class SmartCode
  Inherits SimpleCode

  Public Overrides Function Encode(character As Char) As Char
    Return Encode(character, 3)
  End Function

  Public Overloads Function Encode(character As Char, key As Integer) As Char
    Return Chr(Asc(character) + key)
  End Function

End Class
```

Block A:

```
Dim code As SimpleCode = New SimpleCode()
Dim c As Char = code.Encode("B"C)
```

Block B:

```
Dim code As SimpleCode = New SmartCode()
Dim c As Char = code.Encode("C"C)
```

Block C:

```
Dim code As SimpleCode = New SmartCode()
Dim c As Char = code.Encode("D"C, 4)
```

Block D:

```
Dim code As SmartCode = New SmartCode()
Dim c As Char = code.Encode("D"C, 4)
```

3. Will the pieces of code above give errors?

	No error	Compiler error	Run-time error
Block A	☐	☐	☐
Block B	☐	☐	☐
Block C	☐	☐	☐
Block D	☐	☐	☐

4. What is the value of `c` for each piece of code?
Leave the box empty if a block of code has errors.

Block	**Value**
Block A	☐
Block B	☐
Block C	☐
Block D	☐

5. In an object-oriented language such as VB.NET, what kind of method call would be preferred for getting attachments from a document?

☐ `DocTools.GetAttachments(documentId)`
☐ `document.GetAttachments()`

6. In a subclass you can override the behaviour of a base class.
Which of the following statements is true, according to the Liskov Substitution Principle?

☐ I don't know.
☐ A subclass may deliver *less* functionality than the base class in overridden functions.
☐ A subclass may deliver *less or more* functionality than the base class in overridden functions.
☐ A subclass may deliver *more* functionality than the base class in overridden functions.

7. For this question, refer to the supplement *Design Patterns* that includes diagrams of fictional systems.
Each UML diagram matches exactly one of the 6 design patterns given below. Link the diagrams to the corresponding design patterns by drawing lines between them.
Note: if you think a diagram corresponds to multiple design patterns, choose the best match.

Diagram	**Pattern**
A •	• Adapter
B •	• Command
C •	• Decorator
D •	• Factory
E •	• Singleton
F •	• Strategy

There are no strict rules on the maximum size of methods and classes. However, you probably have some personal preferences regarding the length of classes and methods. Please answer the following questions by selecting the answer that comes closest.

8. I try to keep the length of most of my classes within ...

- ☐ I don't know.
- ☐ 100 lines
- ☐ 250 lines
- ☐ 500 lines
- ☐ 1000 lines
- ☐ 2000 lines

9. I try to keep the length of most of my methods within ...

- ☐ I don't know.
- ☐ 10 lines
- ☐ 20 lines
- ☐ 50 lines
- ☐ 100 lines
- ☐ 200 lines

10. There is often a trade-off between the number of classes and the size of classes and their methods. Small classes are preferred over large classes, just as a small number of classes is preferred over a large number of classes. However, if classes are made smaller by splitting up a class into multiple ones, the number of classes increases.
When facing this dilemma, which aspect is more important for you?

- ☐ To have small classes and methods. (Downside: more classes)
- ☐ To keep the number of classes as low as possible. (Downside: larger classes)

Unit Testing

11. How often do you execute (a subset of) the unit test suite for Exact Online?
Please select the answer that comes closest.

- ☐ multiple times a week
- ☐ once a week
- ☐ multiple times a month
- ☐ once a month
- ☐ almost never

12. How often do you write a unit test?
Please select the answer that comes closest.

- ☐ multiple times a week
- ☐ once a week
- ☐ multiple times a month
- ☐ once a month
- ☐ almost never

The End

Supplement: Design Patterns

Diagram A

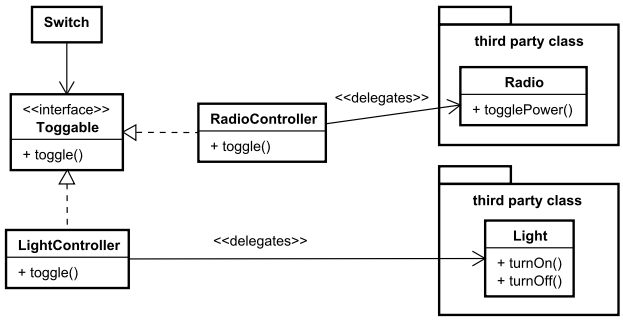


Diagram B

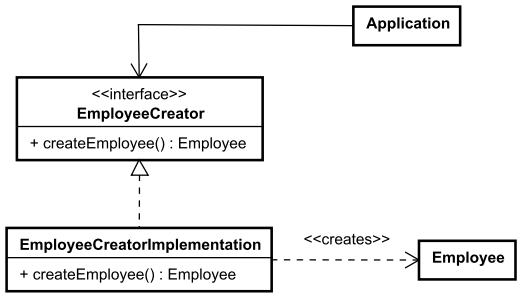


Diagram C

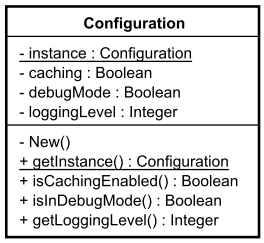
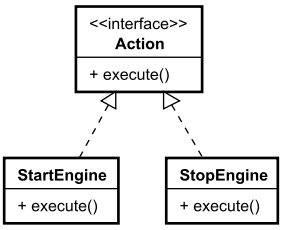
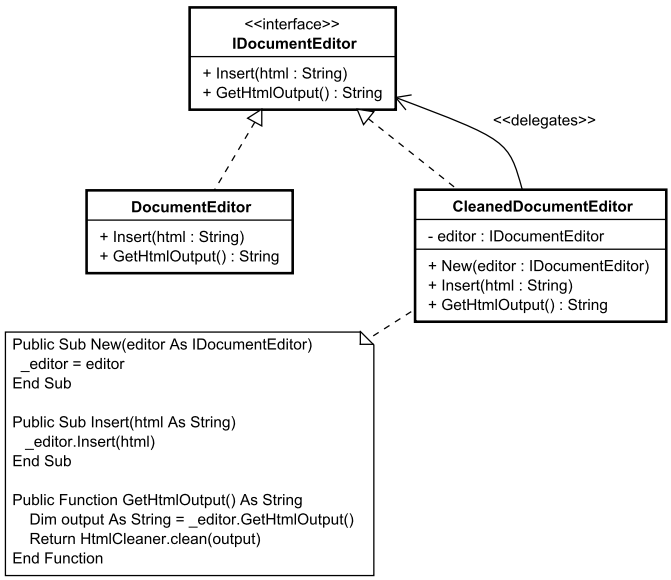


Diagram D



(please turn over)

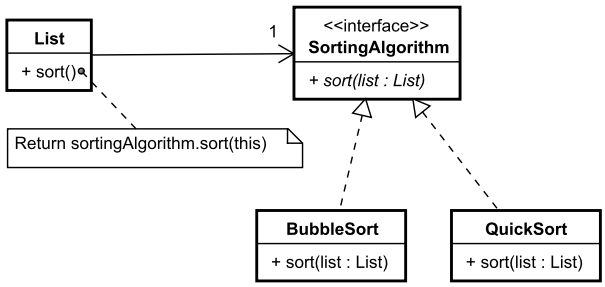
Diagram E



Application code:

```
Dim editor As IDocumentEditor = New DocumentEditor()
Dim safeEditor As IDocumentEditor = New CleanedDocumentEditor(editor)
```

Diagram F



Appendix B

Questionnaire about Refactoring Training

The software engineers who particated in the refactoring training were asked to complete a questionnaire afterwards, of which the raw results are given in this appendix. Some answers have been slightly changed with respect to spelling and grammar, unless there was a risk of changing the meaning of the answer.

1. What are the three most important things (to yourself) that you learnt during the training?

- 1. Code refactoring makes code more readable. 2. Functions with proper naming help understanding better.
- 1. How to refactor bad code. 2. How to reduce dependencies on a concrete class, by depending on an abstract class / interface. 3. Increase confidence that a huge refactoring could actually be done by refactoring small functions step by step.
- 1. Technical debt & code smells in EOL. 2. How to start to do clean code, what can we do. 3. Hands-on refactoring.
- 1. How to do refactoring (especially applying the Singleton design pattern for ERDomain was an eye opener) 2. Awareness of test driven development (write unit tests before coding). 3. Awareness of the code smells. (e.g., I was previously not aware that more than 3 arguments is bad)
- 1. Code smells 2. One class should have one responsibility and one function should have one purpose. 3. Try on using abstract class.
- 1. Always keep the code clean, readable and maintainable. 2. Unit tests as the safeguard for any code changes (e.g., for adding new features or even refactoring!) 3. Spend some time to clean up code (boy scout rule).

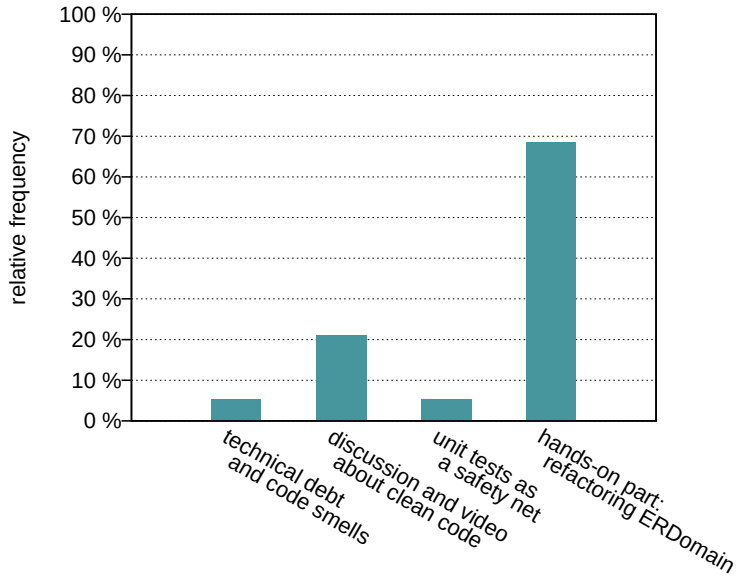
- 1. Keep the functions as short as possible; only do one thing. 2. More functions is better, we can label what code does. 3. Keep renaming functions to be as meaningful as possible and so that they communicate its purpose.
- 1. "Anyone can write code that a computer understands, but not everyone can write code that humans understand" 2. Clean code (short functions, do one thing, descriptive function names) is really important.
- 1. Planning, think how to design before writing code, especially for new functionality. 2. Singleton design pattern. 3. Always keep unit testing in mind.
- 1. What the meaning of 'one thing' is, this is really an important guideline to me. 2. Use unit tests as a safety net for refactoring. 3. How to move logic out of a multiple responsibilities class.
- 1. Technical debt. 2. Code smells and clean code. 3. Refactoring (hands-on).
- 1. Code smells 2. Refactoring 3. Unit testing
- 1. Refactoring 2. Clean code 3. How unit testing benefits.
- Understand refactoring, writing clean code and unit tests.
- 1. Reduce duplication of code. 2. Writing small and understandable methods and classes. 3. Always use meaningful function names.
- 1. Clean code rules 2. Refactoring steps 3. Unit tests
- 1. Refactored code is easier to read. 2. It's okay to take time to think of an understandable name. 3. Classes can be smaller.
- 1. Understandable/readable functions, methods, variables. 2. Smaller classes. 3. Refactoring 4. Unit tests as safety.
- 1. Single responsibility 2. No duplication 3. Separation of concerns.

2. Now that you have had this training, what aspects (if any) in the way that you write code, do you expect to change?

- Write smaller functions. Before copying, think of how to combine methods/functions
- Would consider to create class/interface in a way that we could minimize the dependencies. Will also take unit test into consideration when designing it.
- Yes, I'll change. Single responsibility, unit test, reusability.

- Apply the ‘boy scout’ rules whenever checking-in code and encourage the team to do so as well. Think of design patterns before go for real implementation. Also, to search and read more articles about refactoring & design pattern.
- The length and responsibility of a function or class. Also on how to make them even smaller and group them to be more independent.
- I do expect to change in many aspects including readability, maintainabiliy, separation of concerns and many more for my code.
- Keep functions short and make classes have only a single responsibility.
- Yes. The code will be extremely nice if broken to smaller functions. I will start to practice this immediately in my daily job.
- I will think twice about we should organize our code before we start coding.
- Use descriptive names, small code, single responsibility class.
- To start with: should have smaller methods, classes should have a single responsibility and duplication should be avoided. Naming conventions should be taken into consideration.
- Have more classes and shorter methods.
- Yes.
- All codes are unit testable.
- Reduce duplication, reducing size of methods and classes, make use of interfaces.
- Yes, I would change, apply the rules/principles that I learned to my daily coding work.
- Write smaller methods.
- Better naming of classes, methods and more readable functions.
- Single responsibility

3. Which part of the training contents did you like most?



4. Would you recommend this training to a colleague or not? Please explain why.

- Yes, it creates good code practices and quality to all developers.
- Yes, for someone who is new to clean code where functions should be small, single responsibilities, etc., it could be an eye opener for them. Also, most people know that code is bad but they don't know how to refactor.
- Definately YES! Most of the colleagues understand the technical debt in EOL, but by attending this training, I believe everyone will feel the impact more & the urgency is immediate. And I believe this training will give a better idea how we can start to do better coding because of the hands-on.
- Yes, it's an interesting and fun session and promotes awareness on how to do refactoring. Overall, I think this is an excellent and valuable session. It does trigger me towards more about refactoring, the art of programming.
- Yes, to create more awareness among the developers.
- It feel it's very important for Exact colleagues to know about refactoring skills, especially when we work on the corporation framework.
- Yes, every developer should have the same knowledge so that improvement in the team is possible.

- Yes, this training is really a benefit to everyone, especially software engineers. This training provides a step-by-step guide how to make code look better and [how to write] clean code.
- Yup, to me this is not only for refactoring purposes, I also learnt a lot of coding methods.
- Yes, it is really relevant to software engineering work and I wish everybody could [learn] these skills.
- Yes, because often software engineers code just to get the things done without thinking about technical debt they are going to introduce. It's very important to understand why clean code is needed.
- Yes, it's very beneficial to other software engineers.
- Yes, all the software engineers in EOL should attend this workshop, so everyone can develop quality code.
- Yes, because this training can help us to be better developers.
- Yes, it is really useful to everyone.
- Yes. If everyone writes more structured code / easier readable code, my life would be easier (in Exact).
- Yes, so that more people would write smaller methods.
- Yes, in this training I learnt a lot of new things that I didn't know. In this training also shows the level of my programming skills.
- Yes, because a clean code base is the responsibility of all developers.

5. Do you have any suggestions how this training could be improved?

- More examples. ERDomain is a good example, but it was a little complex and less practical in day to day Exact tasks.
- Well, probably the thinking part should be emphasized more. This is because the learning part is truely dependent on how much one thinks about *how* to refactor bad code.
- Longer time, too short for the session.
- It's fun for the first day session (but maybe for research purpose). Maybe can talk more about how to refactor EOL code to apply other design patterns (besides Singleton).

- Perhaps give more time and it is kind of rush as some people might need more explanation on each step they are doing.
- Perhaps more hands-on session would be great! :)
- 1. It should be 5 days of training. [Likely refers to all 4 days, including experiments, so the suggestion is to have one more day.] 2. Should have an assignment to refactor existing code and also create new code at the end of the training session, maybe on the last day of training. Maybe try to refactor on the first day, just to see how effective the training is.
- If time is not an issue, to complete all of the 'refactoring hands-on' (ERDomain) would be excellent and a plus side.
- Can't think of any so far.
- I think the idea to pick one team's [???] and do the refactoring together is very good.
- It went well overall, but perhaps we can improve by allocating more time for the hands-on. (But that shouldn't reduce time for other stuff. That was quite okay.)
- Perhaps there could be at least two examples of refactoring (different classes).
- I like the training structure now. It manages to give a very good idea/concept how and why refactoring is needed in EOL.
- For me, I want to ask questions, but time is the constraint. Hopefully the training period will be longer or the size of the group reduced.
- Make it a 3 days training. [Likely refers to the two days of training, excluding the experiments, so the suggestion is to have one more day.]
- Maybe can allocate more time (longer session) for the training so that we can ask more questions and also finish the hands-on of refactoring.
- Time was not sufficient. It would be good to be able to finish the hands-on.
- None.
- Refactoring assignment to atendees and the result can be reviewed and feedback to them on what's right & wrong.