Selecting Bug-prone Components to Study the Effectiveness of Reengineering and Unit Testing

Master’s Thesis

Jerry de Swart
Selecting Bug-prone Components to Study the Effectiveness of Reengineering and Unit Testing

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jerry de Swart
born in Dordrecht, 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.
Cloud Solutions
Molengraaffsingel 33
Delft, the Netherlands
www.exact.nl
Selecting Bug-prone Components to Study the Effectiveness of Reengineering and Unit Testing

Author: Jerry de Swart
Student id: 1392050
Email: JerrydeSwart@gmail.com

Abstract

Many software companies see their code grow into legacy code. Making changes to such code is risky, as existing functionality elsewhere can easily be broken. By reengineering the code and covering it with unit tests, the code can be brought back into a maintainable state. In this study we measure the effect of reengineering and unit testing code on the number of fixed bugs. We do so by comparing the predicted and actual number of bugs for a component, after reengineering and unit testing it. We predict the bug-proneness of components by mining the bug repository, and provide and evaluate an approach for selecting the most feasible components by including the estimated test effort. Initial results indicate that the number of bug fixes decreases after a bug-prone component has been reengineered and covered with unit tests. We conclude that our approach is able to predict the bug-proneness of components, and successfully ranks them by feasibility. But in order to formulate a final answer on the effectiveness, more data is needed.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Company supervisor: Ing. Wim Veninga, Exact
Committee Member: Dr. A. Bozzon, Faculty EEMCS, TU Delft
After 8 months of hard work at Exact International Development, I proudly present my Master’s thesis!

I would like to thank a lot of people, in the first place my daily supervisors Andy Zaidman and Wim Veninga for all their support, time, feedback, and for everything I learned during the project from both an academic and business perspective. I would like to thank my parents for their support throughout my studies, and my sister Louise de Swart for her help with the statistics and layout of this thesis in particular. I would like to thank my friends and family for their continued support, in particular Jamie Dorré, Gijs van der Ent, Roland van der Linden and Joyce Waarts.

Many employees from Exact have helped me throughout the process and I would like to thank all of them for the great time. I would like to thank the platform team for their warm welcome, and special thanks go out to Wietze Veld, Ad van der Hoeven, Leo van Houwelingen and Bart Platzbeecker for their support and feedback.

Furthermore I would like to thank Arie van Deursen and Alessandro Bozzon for taking the time to evaluate my work.

Jerry de Swart
Delft, the Netherlands
August 7, 2013
List of Figures

1.1 Research Procedure ........................................... 4
1.2 Research timeline ............................................. 6

2.1 Research Procedure ........................................... 9
2.2 A typical bug report with associated changesets ............... 16
2.3 A bug report with merge tasks attached as child tasks ......... 17
2.4 Example of a changeset with files ................................ 18
2.5 The query used to retrieve the bug reports from TFS .......... 20
2.6 Example of a linked bug report entry in XML format ........... 22
2.7 Correlations of predicted with actual bugs ...................... 23
2.8 Results of the Psla predictor ................................... 23
2.9 Results of the Phugs predictor ................................... 23

3.1 Research Procedure ........................................... 25
3.2 Component selection procedure ................................ 26
3.3 Predicted vs actual feasibility (bugs per required test effort) . 35
3.4 Actual feasibility related to test effort ........................ 36

4.1 Research Procedure ........................................... 39
4.2 Method Invocation on Environment ............................. 43
4.3 Message Chaining: An example of a violation of the Law Of Demeter (LOD). . 43
4.4 Object Passing: An example where the Environment is required as parameter. . 44
4.5 The three responsibilities ...................................... 46
4.6 The original ValidateAccount function ........................ 47
4.7 Added color overlay according to responsibilities ............... 48
4.8 Extracted business logic to new functions ...................... 49
4.9 Injecting the data into the business logic functions .......... 50
4.10 The newly created "dataprovier" object and its interface ...... 51
4.11 Injecting the data provider into the service layer .......... 52
4.12 The first solution: Moving the behavior of the client into the Environment .... 54
4.13 The second solution: Communicate with the required member object instead of the Environment ............................................. 54
4.14 The remaining dependencies on Environment ............................................. 56
4.15 The final fully testable solution ............................................. 57

5.1 Research Procedure ............................................. 59
5.2 The PriceTools and EntryPriceTools classes ............................................. 60
5.3 The ItemPriceTools class ............................................. 61
5.4 The GetVATReferenceDate function in LogisticsTools ............................................. 70

7.1 Research Procedure ............................................. 91
7.2 Component bug scores six months before and five months after the snapshot of Februari 13 2013 ............................................. 92
Chapter 1

Introduction

Many software companies see their codebase grow into a large, untested legacy codebase. Making changes to such code is risky, as existing functionality elsewhere can easily be broken. Early detection of such bugs becomes increasingly difficult as the codebase grows, because a unit test suite is lacking. As a consequence, developers become very cautious of making changes and the risk is mitigated by minimizing the number of changes. This may seem like working with care (a professional thing to do), but in practise it only makes the problem grow even further [1]. The code has taken control over the developers, as Feathers puts it:

“In poorly structured code, the move from figuring things out to making changes feels like jumping off a cliff to avoid a tiger. You hesitate and hesitate. Am I ready to do it? Well, I guess I have to.” [1]

The situation can be turned around by reengineering the code and covering it with unit tests. Such reengineering techniques involve code refactorings and restructurings, to achieve testability (e.g. by breaking dependencies) and to increase maintainability (e.g. by decomposing large functions and classes). Gaining a better understanding of reengineering and unit testing legacy code forms the topic of our study.

We use the term component to indicate all code inside a single file, so each component consists of at least one class. In our study, we measure the impact of reengineering and unit testing a component on the number of bugs that need to be fixed for that component. We do so by first predicting the bug-proneness of components. We use the term bug score to indicate the relative level of bug-proneness. Then, we select components for which most bugs can be prevented per invested time (called predicted feasibility). After reengineering and unit testing the selected component into an improved replacement, it is put into the live production environment. After a period of time, we compare the number of bugs that the new component has caused to the number of expected bugs. The number of expected bugs is estimated afterwards, using the bug scores and actual bugs caused by similar components that form a control group.
1. Introduction

1.1 Problem Context

The research project is conducted at Exact\textsuperscript{1}, an international company employing over 1,800 people worldwide that strives to serve the entrepreneurial world with information technology. Their latest product, Exact Online\textsuperscript{2} (abbreviated EOL), is online business software running \textit{in the cloud} and is used by over 100,000 companies worldwide.

EOL is developed using Microsoft’s \textit{Visual Basic .NET} language. Instead of using the traditional software release life cycle (e.g. with alpha and beta versions), the live environment (or production environment) is being patched on a daily basis. The software is subject to non-stop (small) changes: for example the \textit{Environment.vb} file (an important component) has changed 16 times in the first 4 months of 2013. For our study, we use \textit{snapshots} of the code currently deployed on the live environment.

As is the case with many software companies with a history similar to Exact, large parts of the codebase have grown into legacy code. Very large classes and complex functions form no exception, for which understanding the code and making changes without breaking existing functionality is difficult. The reengineering and unit testing approaches discussed in this study are able to help the developers gain back full control over large parts of the code again.

\textsuperscript{1}http://www.exact.com
\textsuperscript{2}http://www.exactonline.com/
1.2 Terminology

Some of the concepts that are key to this research are the following:

**Legacy Code** Code without unit tests, in particular code that is not testable and requires refactorings.

**Component** A part of the source code: we define a component as the combination of classes inside a single source file. The component is named by the filename, for example the `PriceTools.vb` file contains the `PriceTools` component consisting of the three classes `PriceTools`, `EntryPriceTools`, and `ItemPriceTools`.

**Function** Synonym for `method`.

**Bug** An error in the source code. We use the term *latent bug* to emphasize that the bug does not have to result in an observable error. Our use of the terms `error`, `defect`, `fault` and `failure` is consistent with Nagappan et al. 2006 [2].

**Bug-proneness** The likelihood of bugs showing up. We measure the level of bug-proneness as the number of bugs that are predicted to show up in a period.

**Bug Score** The relative level of bug-proneness. Chapter 2 discusses algorithms that predict component bug-proneness. Each algorithm uses the number of bugs to calculate the bug score, but they differ in how they score the severity of the bugs.

**Bug Predictor** An algorithm that predicts the bug-proneness of a piece of code. In this study, we predict the bug-proneness of components.

**Test Effort** The amount of effort required to understand, reengineer, and fully cover a piece of code with unit tests.

**Reengineering** The term *reengineering* as used throughout this report is the overall process of restructuring and refactoring legacy code into a more modular form. The main goal of reengineering code for this study is to achieve testable code. Besides achieving testability, increasing the maintainability is a subgoal of reengineering code. A survey by Mens et al. provides more information about reengineering and refactorings [3].

**Mockist Testing** Unit testing code with the help of a mocking framework to fake out dependencies.

**Classical Testing** Unit testing code without a mocking framework, by first refactoring untestable code into testable code. Testability of the code is thus achieved by refactorings instead of a mocking framework.
1.3 Research Questions

Studying the effect of reengineering and unit testing code on the number of fixed bugs forms the center of this study. We ultimately seek an answer to the following question:

**Main Research Question** Does the reengineering and unit testing of a bug-prone component result in a measurable difference in reported bugs for that component?

In order to answer this research question we need answers for a few other questions. Figure 1.1 shows the procedure that we use to answer the main research question.

First, bug-prone components that *will cause bugs* need to be selected. The selection procedure uses the *bug scores* of components to identify the most bug-prone components. The first research question focuses on how to extract the required information from the bug repository used for the development of Exact Online:

**RQ1** How can the bug repository (TFS) be used to extract information about the most bug-prone components?

Knowing which components are most bug-prone is a great first step, but time is a limiting factor for software development. The components should therefore preferably be ranked by estimated *feasibility*, so we know what components should be tested and reengineered to prevent the highest number of bugs per test effort. This leads to the following research question:

**RQ2** How can bug-prone components be selected for which invested reengineering and unit testing effort is spent most efficiently?
After identifying the most feasible components, we can reengineer and cover them with unit tests. As is the case with many legacy systems, major parts of the codebase are hard to test due to dependencies. To achieve testability in these cases, developers are used to working with a mocking framework, so the dependencies can be mocked out. But the use of such a framework for achieving testability in legacy code comes with important disadvantages. We propose to use the classical testing approach, for which the code first needs to be made testable. This leads us to the next research question:

RQ3. What are the obstacles for testability of the Exact Online codebase and how can these testability issues be resolved?

Backed by the answer of research question 4, we are able to build replacements of the components that were selected previously (see RQ2). These are created by unit testing and reengineering the existing bug-prone code, to form a replacement that keeps the existing interface for backwards compatibility. The creation of a replacement forms the motivation for the following research question:

RQ4. What issues can be identified for the selected components and how can these be resolved?

The developers are used to working with legacy code and sometimes write unit tests using a mocking framework. This way of working (called “edit and pray” by Feathers [1]) has an alternative called “cover and modify”, in which unit tests are not only used as a safety net when making changes, but also comes with a number of other advantages. The focus of unit testing shifts from writing unit tests after development (we call mockist testing named after the requirement of a mocking framework to achieve testability) to writing testable code and using unit tests during development (called classical testing).

The research questions and techniques that have been mentioned so far focus on the technical aspects. Part of the teams are currently being trained in the classical testing approach, which provides us the opportunity to evaluate how these concepts are being perceived by the developers themselves using a survey:

RQ5. Do the developers understand that applying the classical unit testing approach is reducing the legacy problem?
1.4 Research Timeline

We use the information inside the bug repository to find answers to the research questions. By matching bug fixes to source locations, so called bug data sets are created: in our case showing what bugs have been fixed for which components. We use such bug sets for two purposes: (1) predicting component bug-proneness and (2) evaluating whether the predictions and previously made decisions were correct. Because EOL is being developed using a continuous deployment model we take a snapshot of the bug repository at a certain moment in time, whereas others use pre- and post-release bugs of a certain version.

![Figure 1.2: Research timeline](image)

Figure 1.2 shows the timeline of our research project, including two snapshots:

**First Snapshot (Prediction)** The first bug repository snapshot is taken on the 13th of February. The predictions of component bug-proneness (further explained in Chapter 2) are based on this bug set. We use a bug history of 6 months for the predictions, indicated as the “Prediction Period”.

**Second Snapshot (Evaluation)** The second snapshot is made 5 months later, on the 13th of July. This bug set is used to evaluate the predictions and decisions that were made on the 13th of February. The time between the predictions and evaluation is called the “Evaluation Period”.

Our replacement of a bug-prone component has been in use for about 2 months from the 13th of May to the 13th of July (see Section 7.1 in Chapter 7 for more details). This period is indicated in Figure 1.2 as “Replacement Active”.
1.5 Thesis Outline

The remainder of this thesis is organized as following: Chapter 2 discusses the use of the bug repository to extract information about bug-prone components (RQ1). In Chapter 3, we describe and evaluate the procedure for selecting the most feasible components (RQ2). Chapter 4 discusses the two obstacles that form the main reasons for the untestability of the code, and presents a step-by-step guide to resolve them (RQ3). In Chapter 5 we discuss the unit testing and reengineering procedure that is being applied to one of the selected components (RQ4), and evaluate the maintainability and testability improvements. Chapter 6 presents the survey that is conducted to evaluate the developers’ opinion about using classical testing to work with and reduce the amount of legacy (RQ5). Chapter 7 provides a discussion about the migration and overall results, and Chapter 8 provides an overview of related work. The final conclusions and directions for future work are given in Chapter 9.
Chapter 2

Bug Repository Mining

Figure 2.1: Research Procedure

The goal of this Chapter is to explain our procedure of predicting the bug-proneness of components by mining the bug repository. It is the first step to answering our main research question (see Figure 2.1). In particular, we focus on answering the first research question:

**RQ1** How can the bug repository (TFS) be used to extract information about the most bug-prone components?

In this Chapter we also evaluate the performance of different predictors. Section 2.1 provides an introduction and background information on bug repository mining and using bug sets for defect prediction. In Section 2.2 the software and process used for the development of EOL are discussed and Section 2.3 explains how the repository mining approach is applied to EOL to produce a ranked list of components. Section 2.4 evaluates and discusses the performance of different predictors and in Section 2.5 we formulate an answer to the first research question.
2. **Bug Repository Mining**

2.1 **Introduction to Bug Sets**

2.1.1 **Terminology**

Some of the key concepts of this Chapter are:

**Bug Score** The relative level of bug-proneness. This Chapter discusses algorithms that predict component bug-proneness. Each algorithm uses the number of bugs to calculate the bug score, but they differ in how they score the severity of the bugs.

**Team Foundation Server (TFS)** Exact Online is developed using Microsoft *Team Foundation Server*¹, abbreviated *TFS*. It supports collaborative software development that provides a variety of services such as version control and work item tracking (bug tracking). In contrast to the products used for open source software development it removes the need for separate systems for version control (e.g. CVS) and bug tracking (e.g. bugzilla), which allows for tighter coupling of the source repository with the bug tracking system.

**Changeset** The set of changes to source files that are being checked in on the source repository. A changeset is similar to a *commit* or *transaction*.

2.1.2 **Bug Data Sets**

Nowadays, it is common practise to use a bug tracker to administer what bugs have been found and fixed. The information inside these bug trackers proved to be a valuable source for many research papers. By linking source locations to bug tickets, so called *bug data sets* are created. These bug sets can be used for a variety of purposes, we use them for the following:

**Bug Prediction** Using historical information to determine where bugs are most likely to show up next (determining the *bug-proneness* of source locations). The first *bug set snapshot* shown in figure 1.2 is used for predicting the bug-proneness of components.

**Evaluation** Looking back to evaluate whether predictions and choices were correct, and finding optimal configurations. The second *bug set snapshot* shown in figure 1.2 is used in our study for evaluation.

The information from the bug repository can be used to predict the bug-proneness of components. The main idea is that a *component that has been causing problems in the past, is more likely to do so in the future as well* [4]. We use these insights as basis for our bug predictors, meaning that we expect a component to be more likely to be *bug-prone* (containing latent bugs [5]) when other bugs have been fixed for it.

2.1.3 Bug Predictors

The simplest predictor that has proven itself to be able to predict bug-proneness simply uses the number of previously fixed bugs to predict the bug-proneness of a component. We propose additional predictors that include the severity of the bugs, and evaluate whether these predictors outperform the simple predictor. These predictors weigh bugs of higher severity heavier than bugs of lower severity, and they differ from each other in how heavy they weigh each bug. The predictors cannot be compared: for example it cannot be said that both the P2 and Psla predictors weigh a bug of high severity equally. The ratios between the bug predictors is what matters.

The predictors use the set of previously fixed bugs to calculate a bug score of each component. Table 2.1 shows the mappings of the bug severity to bug scores for different predictors.

<table>
<thead>
<tr>
<th>Bug Severity</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>Psla</th>
<th>Pbugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>1000</td>
<td>1000</td>
<td>3375</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>High</td>
<td>100</td>
<td>500</td>
<td>2250</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>Medium</td>
<td>10</td>
<td>250</td>
<td>1500</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>Low</td>
<td>1</td>
<td>125</td>
<td>1000</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.1: Mapping of bug severities to bug scores

A predictor calculates the total bug score for a component by adding the scores for the individual bugs. When for example a bug of critical and a bug of high severity have been fixed for a component, the Psla predictor gives this component a total bug score of 1500.

The Psla predictor is used in our research to determine component bug-proneness and is further explained in Section 2.1.4. The Pbugs predictor is the traditional predictor that weighs each bug equally.

Predictors P1, P2 and P3 are added for comparison to the Psla and Pbugs predictors. Table 2.2 shows the ratios between the bug severities for the predictors, so it can easily be seen how the predictors differ.

<table>
<thead>
<tr>
<th>Ratio</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>Psla</th>
<th>Pbugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical - High</td>
<td>10</td>
<td>2</td>
<td>1.5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>High - Medium</td>
<td>10</td>
<td>2</td>
<td>1.5</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Medium - Low</td>
<td>10</td>
<td>2</td>
<td>1.5</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.2: Ratios for different predictors

2.1.4 The Psla Predictor

Each bug report contains the severity of the bug, which is either low, medium, high or critical. This information is included when scoring a component (when that component is included in the changeset that fixes the bug) by mapping the severity to a bug score. For our
research, we chose to pick the SLA (Service License Agreement) solution time for fixing a bug for scoring the severities (Table 2.3). These service goals are set by Exact management and the developers agree to meet the solution times based on the bug severity. The decision to use the $\text{Psla}$ predictor is made on the 13th of February (see the timeline in Figure 1.2 in Section 1.4). By including the severity of the fixed bugs, this kind of predictor might be able to outperform the traditional $\text{Pbugs}$ predictor if a component for which a bug of high severity has been fixed turns out to be more likely to cause bugs in the future. We provide an evaluation in Section 2.4.1.

The reasoning behind the mapping of SLA times to bug scores is the following. Let $x$ be the SLA time in days. The more severe a bug is, the lower the SLA time is, and the higher the bug score should be. The bug score is therefore the multiplicative inverse of the SLA time: $\frac{1}{x}$. For better human readability, the “critical” severity is used as reference and given a bug score of 1000, so the result is multiplied by 3000 to achieve this. The resulting formula for calculating the corresponding bug score is $3000 \times \frac{1}{x}$. Table 2.3 shows the result of the mapping. There is no SLA time defined for bugs of low severity as they are not considered to be dangerous or important. For these bugs, the ratio between bugs of high and medium severities is used as ratio between bugs of medium and low severities.

<table>
<thead>
<tr>
<th>Severity</th>
<th>SLA Time</th>
<th>Bug Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>3 days</td>
<td>1000</td>
</tr>
<tr>
<td>High</td>
<td>6 days</td>
<td>500</td>
</tr>
<tr>
<td>Medium</td>
<td>60 days</td>
<td>50</td>
</tr>
<tr>
<td>Low</td>
<td>n/a</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2.3: Mapping of SLA times to bug scores

2.1.5 Success Factors

The success of using bug data sets for prediction and evaluation depends on multiple factors, including:

**Bug Administration** Performed by developers and mergers: missing links may cause bias [6] and bugs go incognito [7].

**Code Stability/Traceability** Renaming a source file should not make it forget its bug history, and code movements to different places (e.g. by redistributing responsibilities) should be taken into account.

**Component Age** A new component may be very bug-prone, but obviously lacks a bug history. BugCache is an example where new components are per definition marked as bug-prone [8]. This factor is related to code churn.

**Granularity** Measuring on different levels (e.g. module, package, file, class, function) may yield different results (e.g. an approach may work well on file-level but not so
well on function-level). Giger et al. present “bug prediction models at the level of individual methods rather than at file-level” in [9].

**Bug Severity**  The usefulness of weighing bugs of higher severity heavier.

**Bug Score Interpretation**  The ability to map bug scores to the field of interest (e.g. mapping to bug-proneness for defect prediction).

By evaluating these factors a level of confidence can be gained in the correctness and usefulness of the results. There are still many research questions to be answered before bug sets can be used as oracles for defect prediction and performance evaluation.

### 2.1.6 Dependency on Bug Administration

The success of a bug mining approach depends on the administrative tasks performed by developers and mergers. In the end, they are the ones that have to leave a reference to the correct bug report for which the changes they check in provide a fix. Also, they are the ones that decide whether to fix a bug using the aforementioned standard procedure or to commit the fix in combination with changes for another task they were already working on. Since fixing a bug in combination with another task saves time and effort (e.g. no need to create a separate sub branch for it) this option is sometimes preferred, causing inconsistencies such as incorrect or missing links. When the severity of a bug is included in scoring the bugs, the correctness of the severity of a bug has to be correct as well.

When creating a bug set, the correctness of the bug reports and links with changesets are assumed. However there are signals that indicate incorrect administration such as linking a bug fix with an incorrect work item. Understanding the size and impact of missing and incorrect links is important, but very hard to measure (if possible at all). Besides missing and incorrect links, it is also possible that bug reports are missing (called incognito bugs [7]). In this case, a bug has for example been fixed while a developer was actually performing some other (often non-bug related) task. The bug fix is then committed together with the changes for the other task, and gets linked to the work item of this other task instead of a separate bug report. This causes a bug to be fixed but the bug report to be missing, detecting these cases is very difficult.

The bug score for components is solely based on the bug reports that have been linked to source locations. The percentage of bug reports that have successfully been linked to source locations (link-success rate) should therefore be as high as possible and can be measured: others have reported typical success rates of 40% to 60%. Bird et al. conclude about such data sets that they are a sample of the actual set of bug fixes and predictions made from such samples can be wrong, especially “if the samples are not representative of the population” [6]. It is important to minimize this effect by maximizing the link-success rate.

### 2.1.7 Caution in using Bug Sets

In order to understand when the bug-proneness of a component becomes visible in the bug score, we need to understand all the aspects that contribute to the bug score. The bug score
of a component is not only based on the amount of bugs it contains. Before a bug in a component adds bug score to that component:

- **the bug has to show up and be identified as a bug** When a piece of code contains a latent bug, the bug might never show up (e.g. the bug only shows up on specific input). So the usage of code containing the bug is relevant. This is an essential difference with defect predictors that use static metrics: a piece of code may be indicated as very bug-prone according to such a predictor, but when that code is only used in a limited way in which those bugs do not show, the bug-proneness never becomes an actual problem. Although metrics such as Fan-In do incorporate this from a theoretical viewpoint, the bug score is based on real input that caused the bug to show up. Whenever a bug does show up, a user or developer has to identify the behavior as being wrong and caused by a bug. This requires understanding of how the system or code is expected to work, which is not always clear.

- **the found bug has to be reported** Whenever a bug shows up and is identified as being one, it has to be reported. But users are not hunting bugs, they are performing a task. Whenever something unexpected happens they are likely to either retry their action, live with the bug, or try performing their task in a different way (using a workaround): see for example the article called “Everything’s broken and nobody’s upset”\(^2\). Unless the issue is really blocking them from performing their task, the bug might never be reported so the developers do not become aware of the issue.

- **the reported bug has to be fixed** When a component contains many bugs that have been found and reported, the bugs need to be fixed in order to add bug score to that component. For example when the component is scheduled for removal it might be a wise decision not to spend any further development effort on it, resulting in a lower than actual bug score.

- **the fix has to be correctly linked to the bug report** In order for a bug to be associated with source locations, the bug report has to be linked to the correct fix. Such links exist in multiple forms: for example studies that use well known open source repositories make use of the fact that developers leave (textual) references to bug reports in the commit messages of the fixes [10]. The lack of such a hint in a commit message means that a bug is not included for determining component bug scores. The correct linking is an administrative task performed by the developers that commit bug fixes, making the link-success rate depend upon the process and compliance of the process that was agreed upon. Since it is a human activity, this process is also prone to human mistakes.

Because of these factors, it cannot be concluded that a component with a relatively high bug score is always more bug-prone than a component that has achieved a lower bug score. We cannot just map the bug score to the bug-proneness of that component without taking these factors into account: something that many researchers do not (e.g. using bug sets of

---

\(^2\)www.hanselman.com/blog/EverythingsBrokenAndNobodysUpset.aspx
low link-success rates in [2, 11, 12, 13, 14, 15]). And even when a link-success rate of 100% is achieved, the other factors should not be forgotten (e.g. incognito bugs and referencing to incorrect bug reports). Further evaluation of the influence of these particular factors and how to measure them is left as future work.
2.2 Approach at Exact Online

This Section discusses the bug tracker mining approach in the context of Exact Online development.

2.2.1 Team Foundation Server

Exact Online is developed using Microsoft Team Foundation Server\(^3\) (abbreviated TFS). It supports collaborative software development that provides a variety of services such as version control and work item tracking (bug tracking). In contrast to the products used for open source software development it removes the need for separate systems for version control (e.g. CVS) and bug tracking (e.g. bugzilla), which allows for tighter coupling of the source repository with the bug tracking system.

2.2.2 Bug Fixing Process

In order to fix a bug, a developer creates a new subbranch on which the bug can be reproduced and fixed in isolation. Before changes to this branch can be checked in, an associated work item needs to be selected. Example work item types are “Task”, “Test Case” and “Bug Report”. As defined by standard procedures, a developer has to select the bug report when checking in changes that fix the bug. When the check-in succeeds, the resulting changeset (containing all the changes in source files) is linked to the selected work item. After a code review, these changes can then be merged to the “Development” branch and later to the “Release” branch by an authorized person (e.g. merge manager). Again, the

changes are packed into a changeset which gets associated to a selected work item.

As shown in figure 2.2, a typical bug report is therefore associated with the following changesets:

- development changeset(s) containing changes made to the subbranch
- a changeset containing the merge to the “Development” branch
- a changeset containing the merge to the “Release” branch

A bug report is also often attached to a changeset in which the subbranch that was used for fixing the bug is deleted. Since these changesets do not contain a possible fix for the bug, they are left aside.

Sometimes separate work items are created for the development, merge to development, and merge to release tasks. These so called “child work items” are often named “Dev”, “Merge Dev” or “Merge Release” and are attached to the bug report (figure 2.3). The changesets that belong to these child tasks are attached to them instead of the bug report. When searching for a changeset that contains the bug fix, these subtasks should be taken into consideration as well.

The explicit association of bugs with changesets performed by developers and mergers differs fundamentally from traditional heuristics that “rely on the premise that developers leave hints or links about bug fixes in the change logs” [10]. When committing any code, an EOL developer or merger is forced to associate a work item by TFS. Chances that the changeset which fixes the bug is properly associated to the bug report have therefore increased heavily, and makes it likely that bug sets with very high link-success rates are achievable. However, since the selection requires human interaction, attaching a changeset to the incorrect bug report is still possible. Some links are thus still expected to be missing.
2. Bug Repository Mining

Figure 2.4: Example of a changeset with files

A changeset contains one or more changes to source files (see figure 2.4). By selecting the changeset that contains the fix for the bug, a bug can be associated with these source locations. This allows bug scores to be calculated for the source locations based on information from the bug report.

2.2.3 Recognizing a Changeset

In order to recognize a changeset to represent a merge to a certain branch, two possibilities are:

1. inspect the comment and use a heuristic to detect keywords such as “merge”, “development”, “release”, and possibly a work item number “WI12345”. An example of a comment from a changeset that represents a merge to the development branch: “Merge from Wholesale/Team1/12345 to Development for WI12345 ***NO_CI***”.

2. scan the path in the changed source files to detect the targeted branch. An example of a change made to a file in the release branch: “$/ExactOnline/Online/Releases/.../Exact.CashFlow.NL/MT940.vb”.

Adding the proper comment to a changeset (when committing changes) is a human task and thus prone to mistakes. The second option uses information that is added by TFS, and is therefore preferred due to its higher reliability.

2.2.4 Information Correctness at EOL Development

In Section 2.1.5 we provided an overview of some factors that influence the success of bug tracker mining. Based on our experiences and conversations with employees, we provide a reflection of these factors at EOL development to gain an overall level of confidence.
Bug Administration  The administrative procedures at Exact Online development are strict and taken serious: developers and mergers are expected to properly perform these administrative tasks. However as discovered by conversations with some developers, bugs do sometimes go incognito. The link-success rate can easily be measured (and are expected to be relatively high), but any numbers about incognito bugs are lacking.

Code Stability/Traceability  Existing components do almost never seem to be renamed or moved, and when they are, only in a very limited way (e.g. moved to a subfolder). Refactorings where parts of the code are being moved do not seem to take place often.

Component Age  Many components (especially the more important components) exist for a long time. New features get added to them, rather than being added by using new classes and refactorings.

Granularity  Measurements on all levels are possible.

Bug Severity  Bug severities are often determined by management, and are based on guidelines. It therefore seems reasonable to assume the correctness of the bug severities.

Bug Score Interpretation  Evaluation of the bug score for defect prediction forms part of our research.

Overall, the configuration at EOL development looks promising (low bias, correct results) and thus well suited for our study.
2. Bug Repository Mining

2.3 Implementation

This Section explains the implementation of the mining program that extracts information from the Team Foundation Server used for the development of Exact Online. For this purpose, the TFS API\(^4\) is used to communicate with the TFS server and extract all required information. Depending on the configuration, it performs the following actions:

Creating a bug set:

1. Gather bug reports by performing a query (Section 2.3.1)
2. Search and attach changesets (Section 2.3.2)
3. Export the bug set to XML (Section 2.3.3)

Analyzing the bug set:

4. Import the bug set from XML (Section 2.3.4)
5. Perform analysis on the bug set (Section 2.3.5)

2.3.1 Querying for Bug Reports

The first step in creating a bug set is to request all existing bug reports from TFS. This is done by executing a query on TFS using work item query language WIQL\(^5\). Figure 2.5 shows the query that is used to retrieve all bug reports with the status “Released”.

```
Select [Id], [Severity]
From WorkItems
Where [Work Item Type] = 'Bug' AND ([State] = 'Released')
Order By [Id] Desc
```

Figure 2.5: The query used to retrieve the bug reports from TFS.

The query returns a list of bug reports along with their identifier and severity. Because the child links to any subtasks of the reports cannot be requested through a query, they are retrieved afterwards using the TFS API.

2.3.2 Selecting Changesets

The next step is to search for the changesets that are most likely to contain the fix for the bug reports. The algorithm that we implemented searches for the best of the linked changesets by first looping through the directly attached changesets and then through the changesets attached to any child tasks. At any time when a changeset representing a merge to release


\(^{5}\)http://msdn.microsoft.com/en-us/library/bb130306%28v=vs.110%29.aspx
is found, the search algorithm stops and returns the changeset. When no changeset with a merge to release exists the algorithm returns the best alternative (changeset with the highest) it could find, such as a changeset representing a merge to development.

The references to bug reports that are added to the comments by developers and mergers is not used at all. But when no changeset can be found for a bug report (e.g. due to selecting the incorrect work item during the commit), the comments of all changesets could be scanned for a reference to the bug report. This way, missing links might be recovered, but currently we have not implemented this.

Changesets are given a so called certainty factor, which represents the certainty that (1) a changeset contains a fix for the bug and (2) the code changes have been put into production (see table 2.4). When a changeset is found for a bug report that represents a merge to release, it is given a certainty factor of 100. When a merge to the development branch is the best changeset that could be found, a certainty factor of 50 is granted. Sometimes, only changesets that represent merges to the separate subbranch (which is created to fix the bug) can be found. These changesets are given a certainty factor of 0, since it is very doubtful that these changesets do indeed contain a fix for the bug that has eventually been merged to the production environment.

<table>
<thead>
<tr>
<th>Branch</th>
<th>Certainty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td>100</td>
</tr>
<tr>
<td>Development</td>
<td>50</td>
</tr>
<tr>
<td>Other</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.4: Certainty scores related to a merge to a branch

The string pattern matching approach (described in Section 2.2.3) is used to identify to what branch is being merged.

2.3.3 Exporting the Bug Set to XML

The bug reports are then exported to an XML file, along with the best changeset that could be found for them. The advantage of exporting to XML is its human readability, but comes at the price of large files. An example of a bug report is shown in figure 2.6.

For each bug report its identifier, severity and best changeset are exported. For each changeset that is attached to a bug report its identifier, certainty score and list of files is saved. When no changeset could be found for a bug report, the bug report is still saved but simply lacks a “BestChangeset” in XML.

2.3.4 Importing a Bug Set From XML

A bug set is loaded from an XML file for analysis. Querying TFS and searching for the best changeset may take quite some time, so skipping the creation of a bug set allows quick analysis with different settings and removes the need for a connection to TFS. Whenever a bug set has just been created, there is no need to import the data from the XML file as it is still in memory.
2. Bug Repository Mining

```
<anyType xsi:type="BugReport">
  <ID>99613</ID>
  <Severity>2</Severity>
  <BestChangeset>
    <ChangesetID>77438</ChangesetID>
    <Certainty>100</Certainty>
    <Files>
      <anyType xsi:type="xsd:string">
        $/ExactOnline/Online/Releases/.../Exact.CashFlow.NL/MT940.vb
      </anyType>
    </Files>
    <ChangesetDate>2013-02-12T12:25:14.47+01:00</ChangesetDate>
  </BestChangeset>
</anyType>
```

Figure 2.6: Example of a linked bug report entry in XML format

2.3.5 Analyzing the Bug Set

In the analysis phase the component bug scores are calculated. To do so, all bug reports in the bug data set are being inspected: for each bug report that has a changeset attached, the date in the changeset is checked first to see whether the bug was fixed within the configured date range. Then, the changeset is checked to see if it satisfies the configured minimal certainty. When it does satisfy these conditions, the files to which changes are made are added to a list if they were not already registered. For these files, a link to the bug report is reported. When this has been done for all bug reports, a list of files along with the bugs that have been fixed for these files (the components) is the result. Based on the severities of these bugs, the bug score is calculated for each of these components. The list of components is then sorted by bug score and printed to a file.
2.4 Discussion

2.4.1 Predictor Evaluation

The use of a second bug set snapshot (see Figure 1.2 in Section 1.4) allows us to verify whether the predictors as described in Section 2.1.3 are able to successfully predict the bug-proneness of components. Figure 2.7 shows the Pearson correlations between predicted and actual number of bugs.

\[
\begin{array}{cccccc}
\text{Actual Bugs} & \text{Pearson Correlation} & P1 & P2 & P3 & \text{Psla} & \text{Pbugs} \\
\text{Sig. (2-tailed)} & .264^* & .337^* & .359^* & .275^* & .368^* \\
N & .000 & .000 & .000 & .000 & .000 \\
\end{array}
\]

Figure 2.7: Correlations of predicted with actual bugs

The results indicate that the predictions made by all predictors show weak to moderate positive correlation with the number of actual bugs that were fixed within 5 months after the predictions were made. For this study we made the decision on the 13th of February 2013 to use the \textit{Psla} predictor to predict the bug-proneness of components. Looking back 5 months later, we conclude that this predictor is indeed able to predict component bug-proneness, but the na"ıve \textit{Pbugs} predictor outperforms the \textit{Psla} predictor. Figures 2.8 and 2.9 show plots of the \textit{Psla} and the \textit{Pbugs} predictors.

Figure 2.8: Results of the \textit{Psla} predictor.  
Figure 2.9: Results of the \textit{Pbugs} predictor.

By tweaking the settings we can find an optimal configuration that would have resulted in the best predictor. Finding such an optimal configuration is feasible as it could also lead to better predictions in the future. We have found evidence that other combinations of settings are indeed able to outperform the \textit{Pbugs} predictor, but leave this as future work.
2.5 Conclusions (RQ1)

The research question that is at the center of this Chapter is the following:

**RQ1** How can the bug repository (TFS) be used to extract information about the most bug-prone components?

By selecting the correct changeset, we can link bug fixes to source locations. This information (stored in bug data sets) can be used by predictors to give each component a bug score, and rank them accordingly. The bug score indicates the predicted bug-proneness of each component. The predictions of all evaluated predictors show a correlation with the actual bugs that have been fixed within 5 months after prediction. The predictor used in this study (Psla) shows a Pearson correlation of 0.275 and can thus be used to predict component bug-proneness, but it does not outperform the Pbugs predictor (0.368 Pearson correlation).

The success of bug repository mining for prediction and evaluation depends on a number of factors, including the correctness of the administrative tasks. We conclude that these factors at EOL development look promising for our study (e.g., low chance of bias in the results), and the information inside the TFS bug repository is of such quality that it can be used for development decisions as well.
Chapter 3

Component Selection Procedure

Figure 3.1: Research Procedure

The goal of this Chapter is to explain and evaluate the selection procedure that we use for selecting components for which reengineering and unit testing effort is spent most efficiently. We answer the second research question:

RQ2 How can bug-prone components be selected for which invested reengineering and unit testing effort is spent most efficiently?

In Section 3.1 an overview of the selection procedure is given. Section 3.2 describes what configuration is used to determine the bug scores of components. Next, as described in Section 3.3, the components that are too data related are filtered out. Section 3.4 describes how an estimation of the required test effort is made and combined with the bug scores of the components to form the ranked list. In Section 3.5 the final selection for this study by manually inspecting the candidates is explained. In Section 3.6 we present the results of the selection procedure. Section 3.7 provides a discussion and evaluates the predicted versus actual feasibility, and in Section 3.8 we answer the second research question.
3. COMPONENT SELECTION PROCEDURE

3.1 Overview of the Selection Procedure

To answer the research questions (see Section 1.3) we are looking for components that have caused many bugs. But because these components need to be fully covered with unit tests and reengineered where needed, these components should preferably also require a low testing effort. As such, we are trying to balance components that are relatively easy to put into a unit test harness and components with a high bug score. The components are therefore ranked by a combination of the two: bug score per test effort.

To select the components that are most feasible for answering the research questions, the following procedure consisting of four steps is used:

1. Determine component bug scores (Section 3.2)
2. Filtering out data related components (Section 3.3)
3. Combining an estimation of the component test effort (Section 3.4)
4. Determine project feasibility by manual inspection (Section 3.5)

The output of each step forms the input for the next step (figure 3.2). The first step produces a list of components that is ranked on highest bug score. In the second step the data related components are filtered out of this list and in the third step an estimation of the test effort is included to determine the project feasibility of the components. In the last step, the final selection of components is made by manual inspection.

Figure 3.2: Component selection procedure
3.2 Step 1: Determining Bug Scores

The bug repository mining procedure as described in Chapter 2 is used to produce a ranked list of components. In this Section we present and discuss the configuration that is used to attach bug scores to components.

3.2.1 Configuration

Table 3.1 shows an overview of the configuration that is used to determine the bug scores for components.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Definition</td>
<td>Based on filename</td>
</tr>
<tr>
<td>Mining Period</td>
<td>Past 6 months</td>
</tr>
<tr>
<td>Severity Scores</td>
<td>Based on SLA time</td>
</tr>
<tr>
<td>Allowed File Extensions</td>
<td>Visual Basic source files (&quot;.vb&quot;) only</td>
</tr>
<tr>
<td>Allowed Changsets</td>
<td>Only merges to the Release branch</td>
</tr>
<tr>
<td>Moved File Matching</td>
<td>Simple pattern matching on filename and package</td>
</tr>
</tbody>
</table>

Table 3.1: Bugtracker settings

Component Definition

A component is defined by the collection of classes inside a file. The component is named after the filename, so the Environment.vb file contains the component called Environment.

Bug Selection Period

The bug scores of the components are based on bug fixes of the past 6 months, starting from the 13th of August 2012 up to and including the 12th of February 2013. The decision to pick a period of 6 months is based on the idea that “the fixed bugs of the past 6 months seem reasonable to determine the bug-proneness of a component”. The bug fixes that took place longer than 6 months ago do not seem relevant any longer, because the components are likely to have changed too much in the meantime for the bug to still be relevant for defect prediction. All bug fixes are treated as equally important, evaluation of approaches that weigh bug fixes to age (e.g. see [16]) is left as future work.

Severity Scores

We use the Psla predictor, that is explained in Section 2.1.4.

File Extensions

The filenames of the components are filtered on the "vb" extension, which are Visual Basic source files. Large parts of the source code has been put into "aspx" files as well (ASP.NET
source files), but these are excluded from selection because of the limited tool support for collecting static metrics from ASPX files. Section 3.7.3 contains more info about a separate ASPX selection.

Allowed Changesets

Only changesets that contain a merge to the Release branch are included in the results. When a changeset does not represent a merge to release, something is likely to be wrong (either with the bug fix or bug report). Bug reports for which no changeset can be found at all are per definition not included, because a changeset is required to determine what components have been changed to fix the bug. Also, for these bug reports it cannot be determined whether their fix falls within the bug selection period. Bug reports for which only a changeset can be found that does not represent a merge to the release branch are left out as well, because there is often a reason why the fix did not make it to the release branch in the end. Manual inspection shows that bugs sometimes will not be fixed even though the fix has already been developed (e.g. the code will soon be removed anyway) or the changes of multiple related bugs are merged to release at once with the changeset being attached to only one of the bug reports. Including these changesets would result in false positives and counting a bug twice. The problems with these bug fixes are caused by human error: the status of the bug report has been incorrectly set to “Released”, or the bug fixes should have not been merged at once but separate.

Moved File Matching

Using a simple pattern, file moves within a package are detected. Before a bug score is added to a component, the algorithm checks whether a component that matches the pattern already exists. When it finds such a component, the bug score is added to this component.

The algorithm matches filename and package, and ignores subfolders. For example the “Environment.vb” file has been moved to a subfolder called “environment”. The algorithm matches “Exact.Core/Environment.vb” with “Exact.Core/Environment/Environment.vb”, by checking whether a component with the same filename (“Environment.vb”) and package (“Exact.Core”) is already present. Files that have been renamed or moved to a different package are therefore not detected.

The decision to not put additional effort into more complex approaches (that can detect file renaming and moves to other packages) has been made after it became clear that not much files are or have been renamed or moved in the EOL codebase. And by applying the simple pattern matching algorithm that is described above, most file moves are detected.

3.2.2 Results

The query returned a total of 4851 bug reports with the status “Released”, of which 1341 fall within the selected period. Of these 1341 bug reports, 877 bug reports (65%) are linked to a changeset that represent a merge to Release. This is an improvement compared to the link-success rates reported by others (typically 40% to 60% in [17, 10, 6], see Section 8.1.1) who use the textual reference technique.
Step 1: Determining Bug Scores

However, after investigating why 464 of these bug reports (34%) are not linked, the actual correct linking percentage turned out to be much higher. The problem with existing approaches is that large parts of the links do exist but cannot be found [6]. But in this case, large parts of the links cannot be found because they do not exist.

Some of the reasons (why no changeset with a merge to Release exists) are:

- The bug is fixed in the database without making changes to the source code (fixing data corruption).
- The bug is not reproducible or will not be fixed, but the status of the bug report is incorrectly set to “Released”.
- The bug report is used as overview of multiple bugs, where each bug is fixed in a separate bug report. The bug report that contains the overview is actually a task, not a bug.

After manually inspecting 50 of the 464 bug reports for which no correct changeset could be found, it turned out that 30 bug reports (60%) belong to the group for which no changeset can be found because it does not exist. So an estimated 60% of 464 bug reports should not be included in the total set of 1341 bug reports. By excluding them, an estimated 83% of the bug reports have been successfully linked to a changeset: a significant improvement compared to the results of others that use the textual reference technique. This confirms the suggestion of Section 2.2.2 that high link-success rates can be achieved due to the used process for EOL development (forced work item association) and our approach being able to successfully exploit this.

The resulting list of components ranked by bug score can be found in appendix A.
3. COMPONENT SELECTION PROCEDURE

3.3 Step 2: Filtering out Data Related Components

Many of the components that score high in the first step seem to be well feasible for our research at first sight. Unfortunately after discussing the targets with one of the developers, it was concluded that many of these component contain specific database access or storage code (SQL statements and querybuilders) or data transformation code (BusinessComponent transformations), also called data related code. This means that for unit tests these parts would have to be either mocked out using a mocking framework, or refactored out of the functions in such a way that they can be replaced by fakes. The high amount of data related code is one of main reasons why the code cannot be tested by unit tests. The problem is being analyzed into further detail in Chapter 4.

Although it is possible to factor out the data related code in unit tests, it is likely that these are exactly the parts that have been causing the bugs in the first place. These components should therefore not be tested by unit tests, but for instance by integration tests using a real database. This falls out of the scope of this study, and thus the components that contain a certain amount of data related code are being filtered out by manually inspecting all components in this step. The components that are less likely to contain data related code are the components that belong to the “core” and “utils”.

About all components contain database access code, the most obvious examples being the creation and execution of so called QueryBuilder objects, after which the data is immediately used. However some components contain functions that merely build QueryBuilder objects without actually executing them. These cases are less problematic, and telling the difference using an automated approach is difficult as many factors influence this decision. Therefore it was decided to filter out components that contain too much data related code by manual inspection and not based on an automated approach using predefined rules.

Many of the components in the top 100 bug score list are so called functional components. Without going into details, these components are responsible for performing data related actions (often including QueryBuilder objects). For example the VATGLTransactionEntryFC component (located inside the Exact.VAT package) contains an entire region called “Queries” of 933 LOC on a total of 1301 LOC. The 23 functional components (recognizable by “FC” in the filename) are examples of components that are filtered out in this step.

Another example of a component that contains data related code is PriceTools (PriceTools.vb, located in the Exact.Logistics.Core package). It contains two functions that build and return an AndBuilder, which is similar to the QueryBuilder. Based on this observation it could be decided to filter out this component, but a closer look shows that the remainder of the component is very feasible as research target because of the cohesion amongst these other parts. Also, two such functions on a total of 500 LOC is not so much. Ideally, the components should not contain any data related code but the limited amount in this example is no reason for filtering it out.

Appendix B shows the list of remaining components ranked by bug score, after the data related components have been filtered out. Unfortunately, 80% of the components have been filtered out in this step, meaning that only 20 out of the original list of 100 components are left.
3.4 Step 3: Estimating and Combining Test Effort

In this step, each component is given a *feasibility score* by combining the bug-proneness of a component with the estimated test effort of that component.

### 3.4.1 Testing Effort Estimation

The test effort that is required to cover the components with unit tests is estimated using static metrics. For predicting unit testing effort of classes Badri et al. conclude that "*the metrics that were found significant predictors of the testing effort (in the univariate analysis and in the multivariate analysis as having a relatively high effect) are related to the important OO attributes: size, complexity, cohesion and coupling*" [18]. More specific, they conclude that "*overall, the metrics WMC, RFC, LCOM and LOC have the highest effect on the testing effort*" [18].

Based on these insights, our estimation of the required testing effort for components consists of a combination of:

1. **an estimation of the amount of required tests** by the *combined cyclomatic complexity of all functions* of a component (similar to WMC), based on the idea that for each path through a function one unit test needs to be written.

2. **an estimation of the complexity per test** by the *average cyclomatic complexity per function* of a component.

The estimation of the testing effort incorporates the two OO attributes *size* and *complexity* as identified by Badri et al. in the following way: the sum of the cyclomatic complexities of the functions of a component is an estimation of the amount of required tests and in that sense a measure for *size*, and the estimation of the complexity per test is a measure for *complexity*. The component test effort is calculated by multiplying the amount of tests by the average complexity per test.

#### Estimating the amount of unit tests

The estimation of the amount of required tests is made by the sum of the cyclomatic complexities of all functions. This is equal to the *Weighted Methods per Class* (WMC) metric where each method is weighted by its cyclomatic complexity, but since a component may contain multiple classes the WMC metrics are summed.

The tool used for determining the cyclomatic complexity is Microsoft Visual Studio 2012\(^1\). It is able to calculate the cyclomatic complexity per package, per class and per function. Because we are interested in the cyclomatic complexity of a component (possibly containing multiple classes per file), the cyclomatic complexities of multiple classes have to be summed manually.

For example the PriceTools component contains the classes PriceTools (CC of 41), EntryPriceTools (CC of 16) and ItemPriceTools (CC of 12). Therefore the PriceTools component has a total CC of 69, which forms its estimated amount of required tests.

3. Component Selection Procedure

Estimating the unit test complexity

To estimate how much effort is required to write a unit test, the cyclomatic complexity is divided by the amount of functions of a component, so the average cyclomatic complexity per test is calculated. The higher this cyclomatic complexity per test is, the more effort is required to write a single unit test for such a function.

Unfortunately, Visual Studio 2012 is not able to quickly display how much functions a class or component contains. Therefore, the NDepend tool is used, and the results for individual classes are again summed to determine the amount of functions for a component. As an example, the cyclomatic complexity for the PriceTools component is 69, and it contains a total of 32 functions according to the NDepend tool. Therefore, the estimated average unit test complexity becomes 2.16.

3.4.2 Determining Research Feasibility

The feasibility score of a component is calculated by the bug-proneness of that component divided by the required test effort for that component. The bug-proneness of a component is a direct mapping from the bug score, and the test effort is estimated using the process described before. Table 5.3 shows a small overview.

<table>
<thead>
<tr>
<th>Calculated by / Mapped from</th>
<th>Bug-Proneness / Test Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feasibility Score</td>
<td>Bug-Proneness / Test Effort</td>
</tr>
<tr>
<td>Bug-proneness</td>
<td>Bug score</td>
</tr>
<tr>
<td>Test Effort</td>
<td>Amount of Tests * Test Complexity</td>
</tr>
<tr>
<td>Amount of Tests</td>
<td>Cyclomatic Complexity</td>
</tr>
<tr>
<td>Test Complexity</td>
<td>Cyclomatic Complexity / Number of Functions</td>
</tr>
</tbody>
</table>

Table 3.2: Calculating component feasibility score

Because both the bug tracker and version control system are continuously changing (no strict versioning), the static metrics are based on a snapshot of the Release branch taken at the same moment in time. The bug reports are included up to the 12th of Februari 2013, so the snapshot used for the static analysis has been made in the morning on the 13th of Februari.

The results for the top 20 components as identified in the previous step can be found in appendix C. The final feasibility by manual inspection (color overlay) is explained in the next Section.

---

2http://www.ndepend.com/
3.5 Step 4: Final Selection

3.5.1 Manual Inspection

In the final step, the components are manually inspected for project feasibility. It might for instance be possible that a large component caused so many bugs, that its bug-proneness per test effort ratio is very high compared to other components. Putting reengineering and testing efforts into this component would thus not be a waste of time. However, because the entire component needs to be reengineered and tested thoroughly, time is a limiting factor for this study. Another reason for performing manual inspection is that the used method for predicting testability may fail. For example some classes inside a component inherit logic from a base class, so understanding and testing the component requires knowledge about the inherited base class as well. A component is considered to be either very feasible as research target, hard to test and refactor within time limits, or impossible to use as research target.
3. COMPONENT SELECTION PROCEDURE

3.6 Results

3.6.1 Final Selection

Appendix C shows the top 20 components, with their research feasibility highlighted. Only 4 components turned out to be well suitable for this study:

1. **VATPercentageComboBox**: UI-related component to build up combo boxes, very low cyclomatic complexity.

2. **QuotationDetailTools**: Loosely coupled “tools functions” requiring quite some knowledge of the domain in order to understand.

3. **PriceTools**: Tools functions for calculating value added tax (VAT) that contains a class with high cohesion but also some data related functions.

4. **LogisticsTools**: Loosely coupled “tools functions” in support of other classes.

The VATPercentageComboBox component contains logic for building up a combobox. This UI-related code does not seem to be really important or contain complex (bug-prone) logic. It can be used for this study, but is not really an interesting target. The QuotationDetailTools component on the other hand does seem to be quite important, but requires lots of domain knowledge. The PriceTools and LogisticsTools components seem to contain more complex business logic functions and both come from the same package (*Exact.Core.Logistics*). The PriceTools component also contains a class with high cohesion, which makes it an interesting research target. Therefore, the PriceTools and LogisticsTools components have been chosen as the targets that will be reengineered and tested.
3.7 Discussion

3.7.1 Verification of Research Feasibility

Figure 3.3 shows the predicted versus the actual reengineering and testing feasibility. We can calculate the actual feasibility only for those components that required at least one bug to be fixed in the evaluation period (after prediction). The results show a very strong positive correlation, indicating that the calculated feasibility seem to be well capable of predicting the actual feasibility. It has to be kept in mind that only a limited set of 21 components could be included.

Figure 3.3: Predicted vs actual feasibility (bugs per required test effort)
Figure 3.4 shows the test effort versus actual feasibility. Again, the actual feasibility can only be calculated for a limited number of components. According to these results, the more test effort is required the less bugs per invested test effort can be prevented. A possible explanation could be that components which require more test effort, are more likely to be older and therefore more stable. In addition, developers seem to be afraid to touch those big components as they might be easier to break.

### 3.7.2 Automating the Process

The selection procedure described in this study still requires human interaction because some of the steps require human judgement. Since the output of each step is used as input for the next, the entire process could be automated by combining the tools used in the selection steps. By automating the process, the bug tracker can for example be used to extract information on a daily basis. The information may be used in support of decisions such as where to focus test and refactoring efforts.

Mining the bug tracker in the first step to produce the list of bug scores for the components is already fully automated. Determining what components of this list contain “too much data related code” is much harder as this requires insight into the problem (as described in detail in 3.3). An automated approach might be able to make the same decisions a human would, but due to the problem complexity, this is left as future work. Gathering and combining the static metrics (in the third step) is again a matter of combining the used tools. The resulting list of components can be considered the output of the bug tracker mining process, as the final selection depends on varying project requirements.
3.7.3 ASPX Selection

The selection procedure discussed in Sections 3.2 to 3.5 can easily be adapted for similar selection purposes. To demonstrate this, a ranked list of ASPX files is created as well. These files are used for presentation in a web site. Many source code has been put into these files and therefore it would be feasible to produce a list of ranked ASPX components.

The ASPX files consist of two parts: Visual Basic source code (business logic) and markup code (e.g. HTML and JavaScript). The reasons for not using these files in this study are (1) the inability to extract the same static metrics for ASPX files and Visual Basic source files, and (2) because it cannot be detected whether a bug was fixed on the business logic or markup code. The bug scores are extracted from the bug tracker using the same configuration in Section 3.2, but then with “.aspx” as file extension. The only static metric for ASPX files that can easily be determined is LOC (Lines of Code). A simple measure for determining what ASPX files require attention first could be the bug score per LOC. Appendix D shows the ASPX files ranked by bug score per LOC.
3. COMPONENT SELECTION PROCEDURE

3.8 Conclusions (RQ2)

**RQ2** How can bug-prone components be selected for which invested reengineering and unit testing effort is spent most efficiently?

Such components can be selected by including the *required test effort*: an estimation of the effort that is required to reengineer and write unit tests for a component. By calculating the *bug score per test effort* (called *feasibility*), components with the combination of a high bug score and low test effort rank highest.

Unfortunately, 80% of the components have been filtered out as they contain too much data related code which cannot be unit tested. Eventually the procedure results in a list of components, ranked by feasibility. Of this list, we have selected the **PriceTools** and **LogisticsTools** components as targets to be reengineered and unit tested.

The results of the validation of the actual feasibility show that in general it is better to spend reengineering and unit test effort on the components that require less test effort, as more bugs can be prevented per invested time for these components.
Chapter 4

Reengineering EOL for Testability

Figure 4.1: Research Procedure

In this chapter we analyse the main reasons for untestable code and present a step-by-step guide to remove the identified testability issues. The process of reengineering existing legacy code is focused on achieving unit-testable code:

RQ3 What are the obstacles for testability of the Exact Online codebase and how can these testability issues be resolved?

The goal of this chapter is not only to demonstrate how to reengineer existing code into unit testable code, but the developers can also use the process to gain insights into avoiding such testability problems as currently found in the system.

Section 4.1 defines and explains testability as used in this study. In Section 4.2 the testability problems for EOL are analyzed, showing the main reasons for the code being untestable. Section 4.3 discusses how the code can be made testable without the use of a mocking framework and demonstrates the process with an example. In Section 4.4 we answer the third research question.
4. REENGINEERING EOL FOR TESTABILITY

4.1 Introduction to Testability

4.1.1 Testability Defined

Testability can be interpreted in different ways, and researchers often define testability in a way that matches their particular application area. ISO defines testability in a quite general way as “attributes of software that bear on the effort needed to validate the software product” [19]. In our work testability is specifically defined as “the ability to write a unit test harness for a piece of source code without the use of a mocking framework”. We choose this definition of testability because our main interest is to cover legacy code with unit tests, and are less interested in how easy it is to do so.

4.2 Analysis

In this section we provide an analysis and point out the main reasons why it is impossible to write unit tests for the majority of the classes of the EOL codebase in its current state.

4.2.1 Switching from Mockist to Classical Testing

The starting point is the codebase of EOL which is for the biggest part legacy code that is not covered by an automated unit test harness. Driven by the decision to increase unit test coverage, it is decided to only write unit tests for newly developed code. As is the case for many systems that have not been developed with testability (or even maintainability) in mind, existing code is not testable. Because newly written code is developed in the same way as existing code, also this newly developed code turns out to suffer from the same testability issues.

In an initial attempt to make the code testable, the developers deployed a mocking framework. All function calls that cannot be executed (e.g. database access calls) or objects that cannot be instantiated in a normal way (e.g. due to dependencies in the constructor), are being mocked out using the framework. This is called mockist testing, and differs from classical testing. In the classical testing approach a developer uses either a real object or a “test double” such as a stub. Fowler compares the two testing approaches (with a focus on Test Driven Development (TDD)) and explains the differences into detail in [20].

Although mocking may seem to be a good solution to achieving testability at first sight, in practice at EOL this way of mocking has learned that it comes with important disadvantages:

1. Code Quality: the code under test is not required to be of “decent quality”. The codebase of EOL contains many legacy methods and classes that cannot be unit tested without the use of a mocking framework. In order to test this code using the classical testing approach it has to be made testable first. In the process of achieving testability (e.g. by removing dependencies and splitting the code up) the quality of the source code is very likely to improve. By using a mocking framework, a developer is not forced to write testable code of decent quality.
2. **Missed Testing Goals**: Developers are more likely to apply an *old-fashioned* approach to testing. Writing unit tests is considered to be an “*after development task*” which according to Feathers is “*testing to attempt to show correctness*” [1]. This is a good goal, but testing can be applied for other reasons as well (e.g. writing tests in support of code decisions: “*understanding how your newly developed code would be used*”). Not using a mocking framework is more likely to make the developers think about testability of their code and writing tests early, which promotes the use of unit testing for other goals as well.

3. **Performance Issues**: it is often required to mock big classes which causes the unit tests to take (too) long: unit tests taking seconds to run are not uncommon.

4. **Required Expertise**: the developer requires knowledge of the mocking framework.

5. **Whitebox Testing**: the developer requires a significant amount of knowledge of the internals of the code under test.

6. **Vendor Lock-in**: it creates a dependency on a specific mocking framework.

   Some functions contain many lines of code that retrieve data from the database, followed by relatively few lines of business logic that act on the retrieved data. In order to unit test the business logic of this function, a significant amount of mocking needs to take place due to the dependency on the database connection. This happens so often, that the term “*we feel like we are testing the mocking framework*” is commonly heard.

   A solution is to split the function into a data retrieval and business logic function, so the business logic can be tested by injecting the data into the function. When no mocking framework is available, a developer is forced to make changes to the source under test before any test can be written. Since many developers are unfamiliar with writing unit tests (and unfamiliar with writing testable code), forcing to make changes to the source code can be considered an advantage as it is very likely to improve code quality (maintainability) and break dependencies.

   Using the classical (or *regular*) testing approach, the problems with a mocking framework can be avoided. Unfortunately in order to switch to the *classical approach*, changes to the source code under test are required and developers need to stop writing legacy code and start writing *testable code* first. As a result, the quality of the source code improves, unit tests run fast, developers are more likely to write unit tests and the dependency on a mocking framework becomes history.

### 4.2.2 Reasons for Untestable Code

When one tries to create a unit test harness for a random class or method, the same two problems that are the main reason for untestable source code keep showing up:

1. **Database access**: in the same function where data transformations take place, the required data is being retrieved or stored as well. In other words a function or class has the responsibilities of retrieving (or storing) data and the responsibility of performing calculations on the data.
2. **Dependencies on Environment**: an instance of the *Environment* class is required in order for the class or function to be able to perform its actions.

**Database Access**

Database accesses occur in multiple ways, including:

- calls to (static) caching functions
- instantiation of *database-objects* (loading themselves from the database)
- direct SQL query execution (using a *QueryBuilder* object)

These calls require a working database connection, either directly (as parameter) or indirectly (hidden inside an object that is passed as parameter). The database connection cannot be faked so the calls to *database access* functions need to be mocked out using a mocking framework in order for the code to become testable. Another option to testing code that contains database access calls is to provide a real database connection and use integration tests. However for classical testing, it is impossible to work around the database connection in the current state.

**Dependencies on Environment**

The other problem that forms the main reason for untestable code is the dependency on the *Environment class* (simply called “Environment” from now on).

Based on manual inspection we conclude that the Environment class is a combination of (1) a *godclass* of about 3.5 KLOC that provides a lot of (mostly non-related) functionalities, and (2) a *service-container* that is similar to a data-container but instead of containing data it contains many service objects (such as a logger). It also contains a lot of info about the customer that is currently logged in (comparable to session info) and can be used for many purposes. The functions of Environment itself do not seem that complex (in terms of code complexity and size), but the amount of provided functionalities (number of methods and public *properties*) is what introduces the complexity. Some of the many responsibilities are:

- Licensing
- Database connection
- Date/Time string formatting
- User settings
- Data caching
- Logging
- String localization
Not all of these functionalities are implemented in the Environment itself but many of them are implemented in a separate class (called service objects). These classes can be seen as subparts of the Environment and often require an instance of Environment in order to function. They are therefore strongly related to the Environment class. Access to these member objects is provided by the Environment using public properties, causing the Environment to be passed on whenever a member object (such as the database connection) is required in a function. The instantiated Environment object is therefore being passed on as function parameter or in constructors in many cases (about 3700 times), because these functions and classes use at least one of the member objects that the Environment contains.

The Environment class cannot be instantiated for unit testing purposes, and even if it could it would probably take a relatively large amount of time to do so due to its size. There are three ways in which the Environment is being used inside a function:

1. **Method Invocation**: a function is directly called (figure 4.2)

2. **Message Chaining**: a function on one of the member objects is called (figure 4.3)

3. **Object Passing**: the Environment is being passed on (figure 4.4)

Figure 4.2 shows the first use case, a direct function call on Environment. Note that the variable holding an instance of the Environment class is often called “env”, as shown in the example.

```vbnet
Private Sub Page_Init()
    ...
    Dim perm As Permission = env.GetFunctionRights(...)
    ...
End Sub
```

**Figure 4.2**: *Method Invocation* on Environment.

The second case (calling a method on one of the member objects) is a violation of a rule called the “Law of Demeter”. By adhering to the rule software becomes better maintainable and less coupled, and a number of potential problems can be avoided [21]. In section 4.3.8, we show that by not violating the rule, a particular testability issue can be avoided. Figure 4.3 shows an example of message chaining.

```vbnet
Public Shared Function ValidateAccount (env As Environment, ...) As String
    ...
    env.Term.ConstructTerm(...)
    ...
End Function
```

**Figure 4.3**: *Message Chaining*: An example of a violation of the Law Of Demeter (LOD).
The third usecase is demonstrated in figure 4.4 where a class is instantiated (in this case “BusinessComponent” using a static factory) but the Environment is being passed on as a parameter. This piece of code cannot be unit tested, because we cannot control the creation of the BusinessComponent from the outside of the CopyBillOfMaterial function. So it is impossible to use a stub of the BusinessComponent class here.

```vbnet
Public Shared Sub CopyBillOfMaterial(env As Environment, ...)  
  Dim im As BusinessComponent = BusinessComponent.Create(env, ...)  
End Sub
```

Figure 4.4: Object Passing: An example where the Environment is required as parameter.
4.3 Solution: Achieving Testability

The purpose of this section is to show how to make existing code fully unit testable without having to make changes that are too fundamental (e.g. no changes involving many other classes). The reason for setting this boundary is that the fundamental refactorings would not be changes at a low level of abstraction but rather at the architectural level (high level of abstraction). This implicates that these changes would involve too many existing classes while these refactorings are not strictly necessary to achieve testability. The lack of a decent test harness is what causes the danger, so when the refactorings concern only a few classes a test harness for only these classes would have to be written. Although the plans for a different architecture (that supports better testability of the code in general) are ready, these changes fall out of the scope of this project.

In this section we introduce a step-by-step plan to refactor the untestable code, and demonstrate the approach using an example from the EOL codebase.

4.3.1 Selecting Examples

The developers confirmed that many code has been written as so called “tools” classes that contain mostly static (also called shared) functions. For example in one of the “core functional packages” 13 out of 21 filenames contain the word “Tools”. The static functions inside these tools-classes are used to demonstrate the refactorings because of their feasibility (good examples) and size of the problem (there are many tools-classes). These functions are good examples to demonstrate the reengineering process, but nevertheless the same principles can be applied to smaller and non-static functions with minor modifications as well.

4.3.2 Step-by-step Guide

The existing code cannot be put into a testing harness without using a mocking framework. Fortunately for major parts of the codebase it has been written in an equal way: developers “take existing code as an example”. The non-testability of the code is therefore also often caused by the same reasons as discussed in section 4.2: (1) database access, and (2) the dependencies on Environment. We provide a step-by-step guide in this section, that demonstrates how to tackle these two particular problems in EOL. By removing them, the majority of untestable code can be made testable for the “classical testing” approach.

The steps to resolve the aforementioned problems and achieve testability of the code are the following:

Step 1 : Identifying Responsibilities

Step 2 : Extracting Business Logic

Step 3 : Injecting Data

Step 4 : Injecting Dataproducers

Step 5 : Removing the Remaining Dependencies on Environment
In steps 1 to 4 the first testability issue is resolved. By moving the database calls to a database access shell (called dataprovider), and by injecting an interface of such a dataprovider, the data is no longer directly retrieved from the database but from a dataprovider. A stub that implements the interface of a dataprovider can then be deployed to provide test data in unit tests. Step 5 focuses on the second issue: removing the remaining dependencies on Environment.

### 4.3.3 Step 1: Identifying Responsibilities

Many functions in the EOL codebase have too many responsibilities, especially the larger and more important functions that should be tested thoroughly. A first step to achieving testability is to split the code according to responsibilities. Figure 4.5 shows the separation of code according to three responsibilities: database access shell, business logic and service layer code.

![Figure 4.5: The three responsibilities](image)

**Database Access Shell** Database accesses belong in a Database Access Shell. The code inside the access shell has the knowledge of how to retrieve the data. According to Sneed: "Rather than having the database accesses scattered throughout the code, it is also better to pull them together into a single access component for each database table. In this way different procedures can use the same accesses." [22]. This step increases testability in the sense of removing the need for duplicated tests ("since every database access should be tested" [22]), but we are mainly interested in this refactoring step due to the ability to fake or stub the database access shell for testing purposes. Integration tests are ideal to test the database access shell, either using a real or fake database connection.
**Business Logic** Low level calculations and actions take place in the functions and classes that belong to the business logic. Data should be injected into this code rather than retrieving it. Business logic is well testable by unit tests, due to the injection of data and low dependency on other code.

**Service Layer** Code that belongs to the service layer takes care of the communication between the database access shell and business logic, and performs checks that do not belong to the business logic. The service layer can be tested by unit tests using fakes for the database access classes, however in practice this will mostly come down to testing the business logic again. It might be better to use integration tests (and thus include the database accesses and business logic in these tests), or not spending effort into testing the service layer at all.

Figure 4.6 shows the original *ValidateAccount* function that is used as example to demonstrate the refactoring steps.

```csharp
Public Class EntryAccount

Public Shared Function ValidateAccount(ByVal env As ExactCore.Environment, ByVal sAccount As String, _
                     ByVal bSales As Boolean, ByVal bCheck As Boolean, ByVal bCheckBlocked As Boolean, _
                     ByVal DescriptionTermID As Integer, ByVal Description As String) As String

    Check kind of account
    If bCheckType Then
        Dim Wrong As Boolean
        Dim idType As String = CRMCache.Account.All.Get(env, sAccount, True).Type
        If bSales Then
            Dim saStatus As String = CRMCache.Account.All.Get(env, sAccount, True).Status
            Wrong = (idType <> "A" OrElse saStatus <> "C")
        Else
            Dim bSupplier As Boolean = CRMCache.Account.All.Get(env, sAccount, True).IsSupplier
            Wrong = (idType <> "A" Or bSupplier)
        End If
    End If

    If Wrong Then
        Return env.Term.ConstructTerm(2008, "Invalid", DescriptionTermID, ":" & Description, 3881, "Type")
    End If
End If

If bCheckBlocked Then
    Dim bBlocked As Boolean = CRMCache.Account.All.Get(env, sAccount, True).Blocked

    If bBlocked Then
        Return env.Term.ConstructTerm(DescriptionTermID, Description, 1267, ":Blocked")
    End If
End If

Return ""
End Function

End Class
```

Figure 4.6: The original ValidateAccount function

In figure 4.7, the ValidateAccount function is colored according to responsibilities. It now becomes visible that the ValidateAccount function itself belongs to the service layer and hosts two business logic functions. Each of the business logic functions perform calls to the database (calls to cache functions in this case).
4. **REENGINEERING EOL FOR TESTABILITY**

Figure 4.7: Added color overlay according to responsibilities

4.3.4 **Step 2: Extracting Business Logic**

After the responsibilities have been identified, the business logic code can be moved out into new functions. These extracted functions can either be static functions (see "expose static method" by Feathers [1], page 345) or regular functions in a new class that can be instantiated for testing purposes. The color overlay of the previous step shows two blocks of business logic, suggesting these should be extracted into two functions. Figure 4.8 shows the result of extracting the static “ValidateAccountType” and “ValidateAccountBlocked” functions from the ValidateAccount function.

```csharp
Public Class EntryAccounts1

Public Shared Function ValidateAccount(ByVal env As Exact.Core.Environment, ByVal sAccount As String, _
    ByVal bDelete As Boolean, ByVal bCheckType As Boolean, ByVal bCheckBlocked As Boolean, _
    ByVal descriptionTermID As Integer, ByVal Description As String) As String
    ' Check kind of account
    If bCheckType Then
        Dim sWrong As String = CKCache.AccountAll.Get(env, sAccount, True).Type
        If sWrong Then
            Dim accStatus As String = CKCache.AccountAll.Get(env, sAccount, True).Status
            sWrong = (sWrong <> "A" OrElse accStatus <> "C")
        Else
            Dim bSupplier As Boolean = CKCache.AccountAll.Get(env, sAccount, True).IsSupplier
            sWrong = (sWrong <> "A" Or Not bSupplier)
        End If
    End If
    If bWrong Then
        Return env.Term.ConstructTerm(20006, "Invalid", DescriptionTermID, ": " & Description, 3881, "type")
    End If
End Function

End Class
```
Solution: Achieving Testability

Figure 4.8: Extracted business logic to new functions

4.3.5 Step 3: Injecting Data

The data is still being retrieved by the business logic functions themselves. Instead of retrieving the data from database, the data should be injected into these functions. The ValidateAccountType and ValidateAccountBlocked functions retrieve an AccountAll object from cache. Figure 4.9 shows how the calls to the database cache are pulled up towards the service layer, and the AccountAll object that is retrieved is being injected into the Vali-
4. REENGINEERING EOL FOR TESTABILITY

dateAccountType and ValidateAccountBlocked functions.

Note that one of the dependencies on Environment is being removed from the business logic functions. After moving the data retrieval calls to the service layer, the business logic functions become testable when there is no dependency on Environment left. In this example, both functions still require an instance of Environment which will be removed later in step 5.

```csharp
Public Class EntryAccounts3

Public Shared Function ValidateAccountType(env As Exact.Core.Environment, envVal As String, _
ByVal bSales As Boolean, ByVal bCheckType As Boolean, ByVal bCheckBlocked As Boolean, _
ByVal DescriptionTermID As Integer, ByVal Description As String) As String
    ' Retrieve
    Dim account As CRMCache.AccountAll = CRMCache.AccountAll.Get(env, sAccount, True)
    ' Check the type
    If bCheckType Then
        Dim typeError As String = ValidateAccountType(env, account, bSales, DescriptionTermID, Description)
        ' Return the error when there was one
        If Not String.IsNullOrEmpty(typeError) Then
            Return typeError
        End If
    End If
    ' Check if the account is blocked
    If bCheckBlocked Then
        Dim blockedError As String = ValidateAccountBlocked(env, account, DescriptionTermID, Description)
        ' Return the error when there was one
        If Not String.IsNullOrEmpty(blockedError) Then
            Return blockedError
        End If
    End If
    Return ""
End Function

Public Shared Function ValidateAccountBlocked(env As Environment, account As AccountAll, _
ByVal bSales As Boolean, ByVal DescriptionTermID As Integer, ByVal Description As String) As String
    Dim barang As Boolean
    If bSales Then
        barang = (account.Type <> "A" OrElse account.Status <> "C")
    Else
        barang = (account.Type <> "A" Or Not account.IsSupplier)
    End If
    If barang Then
        Return env.Term.ConstructTerm(20005, "Invalid", DescriptionTermID, ",: Invalid", Description, 2001, "Type")
    End If
    Return ""
End Function

Public Shared Function ValidateAccountBlocked(env As Environment, account As AccountAll, _
ByVal DescriptionTermID As Integer, ByVal Description As String) As String
    If account.Blocked Then
        Return env.Term.ConstructTerm(DescriptionTermID, Description, 1267, ",:Blocked")
    End If
    Return ""
End Function

End Class
```

Figure 4.9: Injecting the data into the business logic functions
### 4.3.6 Step 4: Injecting Dataproviders

In step 3, the retrieval of data has been pulled up from the business logic to the service layer. The next step is to move the data retrieval code from the service layer into the database access shell by creating *dataproviders* that encapsulate the database calls. The data is then requested from these dataproviders. By injecting *an interface of the dataprovider(s)*, fakes such as stubs can be injected for testing purposes. Figure 4.10 shows the *AccountAll-Provider* with its interface, and figure 4.11 demonstrates the injection of the provider into the *ValidateAccount* function.

```csharp
'// (summary)
'// The interface for retrieving AccountAll objects from cache.
'//
Public Interface IAccountAllProvider
  Function RetrieveAccountAll(accountkey As String) As AccountAll
End Interface

'// (summary)
'// The actual implementation for production.
'//
Public Class AccountAllProvider
  Implements IAccountAllProvider

  Private env As Exact.Core.Environment

  Public Sub New(environment As Exact.Core.Environment)
    env = environment
  End Sub

  Public Function RetrieveAccountAll(accountkey As String) As AccountAll Implements IAccountAllProvider.RetrieveAccountAll
    ' Check if the cachedQueriesEnvironment is not null
    If env Is Nothing Then
      Throw New Runtime.InteropServicesException("Failed to retrieve AccountAll " & accountkey & " but the AccountAllProvider is not properly set up")
    End If

    ' Retrieve the AccountAll from the cache
    Return CRMCache.AccountAll.Get(env, accountkey, True)
  End Function

End Class

Figure 4.10: The newly created "dataprovider" object and its interface
4.3.7 Breaking the remaining Dependencies

The second issue that causes major parts of the EOL codebase to be untestable is the dependency on Environment. For each of the three use cases of Environment described in section 4.2.2 (method invocation, message chaining, and object passing) a different solution is required in order to make the code testable. It is important to keep in mind that the possibilities to change source code are limited due to backwards compatibility: prefer-
ably the changes should not break any existing functionality elsewhere or require additional changes on other places.

**Method Invocation**

For the first use case (method invocation on Environment) an interface of the Environment can be deployed to achieve testability. Adding an interface to Environment and using it in a function does not break existing code, and it allows both the use of a full implementation for production and stub implementations for testing. From a maintenance perspective the question arises “whether the function requires the full Environment or just a particular part of Environment”, but no further action is required from a testing perspective.

**Message Chaining**

In the case of message chaining one of the problems of the strong relation between Environment and the member objects occurs. In a sense, member objects of Environment are part of Environment. So when the Environment is replaced by an interface of Environment for a parameter, one would expect an interface for the member objects as well. This is however not the case, and cannot be achieved by replacing the returntype of the getter for an interface of the member object because this would break existing functionality. Existing code that retrieves a member object of Environment using a getter does not expect an interface to be returned but the actual type of the member object.

As a possible solution, the member objects can be adapted to become usable for testing purposes. By adding an empty constructor (without any parameters) and making the member object *overridable*, it is possible to create stubs that inherit the functions and override them with stub implementations. This approach however cannot always be applied, as a constructor without any parameters may already exist in the member object. When for instance a database connection is being set up inside this constructor, it cannot be used for testing purposes without making changes first. Because changes to source code are required in order to use the member objects for testing purposes, this is not considered to be an elegant solution.

Another solution would be to introduce extra getters for the member objects, which coexist with the existing getters. These new getters return an interface of the member objects instead of the actual type. The real member objects are returned in production, while stubs or fakes can be returned for unit tests. Code that requires a member object should thus use these new getters in order to become testable. This approach does achieve code testability, but unfortunately the duplication of getters forms a disadvantage. Since the Environment contains so many member objects, the duplication results in a considerable growth of the Environment (which is already large). Also, the existing getters would become obsolete and should be removed. But due to the heavy use of the member objects, replacing the existing calls for calls to the new getters requires a major effort.

There is however a different solution for this particular testability problem: solving the underlaying problem. Environment is a *service container* that contains a lot of member objects. It is the use of one of the internals (the member objects) of Environment from the
outside of Environment that causes the message chaining (see figure 4.3). To resolve this problem either:

1. **Apply delegation pattern**: the behavior of the client should be *moved closer to the services* so the client can use the Environment itself instead of the internals (figure 4.12).

2. **Inject (an interface of) the member object**: the client code should only communicate with the particular member object it requires instead of the Environment (figure 4.13).

```vbnet
Function ValidateAccount (env As IEnvironment, ...) As String
    ...
    env.ConstructTerm(...)  
    ...
End Function
```

Figure 4.12: The first solution: Moving the behavior of the client into the Environment.

By applying the first option, the Environment would become the *service provider* of the member object, thus adding additional functions to the Environment. Because manual inspection shows that the Environment and member objects are very unrelated, adding the functionality of the member object to Environment does not make any sense. In other words, the Environment containing the member object does not make sense in the first place. Therefore this is not a preferable solution, although it does make the function testable because instead of the Environment type an interface can now be applied.

```vbnet
Function ValidateAccount (termProvider as ITerm, ...) As String
    ...
    termProvider.ConstructTerm(...)  
    ...
End Function
```

Figure 4.13: The second solution: Communicate with the required member object instead of the Environment.

The second option to resolving the message chaining would make the client code communicate directly to the member objects instead of the provider. By applying this option the provider of the services (Environment) would not be needed anymore which is our goal to achieve testability. We are then able to pass an interface of the member object, as this change does not cause any existing code to break. Stubs can implement the interface for testing purposes while the only change that is required to the source code is the extraction and implementation of an interface, and no changes to the internals.
Object Passing

For the third usecase of Environment (section 4.2.2) there are two solutions to achieve testability:

1. **Pass the interface**: make the function or constructor that is being called accept an interface of the Environment.

2. **Create specialized objects**: move the code to a new object that is being set up using the Environment and inject an interface of the newly created object. Replacing database access with a database access shell that knows how to retrieve the data is an example of this solution.

The second option is a more elegant solution since the first solution would result in a situation where the code being called is tested as well. Another issue with the first option is that the changes are made to other objects instead of the code under test, which is not preferable either. Because the second solution uses interfaces of the specialized objects, stubs can easily be set up and used for testing.

4.3.8 Step 5: Removing the Dependencies on Environment

During the process of splitting the functions according to their roles in steps 1 to 4, some dependencies on Environment have been automatically removed as a result (by moving the database access into an access shell for instance). In the final reengineering step the remaining dependencies on Environment are being removed. Figure 4.14 highlights the places where Environment is being passed on, or used.

The two functions “ValidateAccountType” and “ValidateAccountBlocked” contain examples of a violation of the Law of Demeter as discussed before. By injecting an interface of Term (a member object of Environment), the Environment is no longer required. The result is shown in figure 4.15.
Figure 4.14: The remaining dependencies on Environment
Solution: Achieving Testability

```
Public Shared Function ValidateAccount(account As AccountAll, termProvider As ITerm, ByVal sAccount As String, ByVal Sales As Boolean, ByVal bCheckType As Boolean, ByVal bCheckBlocked As Boolean, ByVal DescriptionTermID As Integer, ByVal Description As String) As String
    ' Retrieve
    Dim account As CRPCache.AccountAll = accountAllProvider.RetrieveAccountAll(sAccount)
    ' Check the type
    If bCheckType Then
        Dim typeError As String = ValidateAccountType(account, termProvider, Sales, DescriptionTermID, Description)
        ' Return the error when there was one
        If Not String.IsNullOrEmpty(typeError) Then
            Return typeError
        End If
    End If
    ' Check if the account is blocked
    If bCheckBlocked Then
        Dim blockedError As String = ValidateAccountBlocked(account, termProvider, DescriptionTermID, Description)
        ' Return the error when there was one
        If Not String.IsNullOrEmpty(blockedError) Then
            Return blockedError
        End If
    End If
    Return ""
End Function
```

```
Public Shared Function ValidateAccountType(account As AccountAll, termProvider As ITerm, ByVal Sales As Boolean, ByVal DescriptionTermID As Integer, ByVal Description As String) As String
    Dim bDlgーン As Boolean
    If Sales Then
        bDlgーン = (account.Type <> "A" OrElse account.Status <> "C")
    Else
        bDlgーン = (account.Type <> "A" Or Not account.IsSupplier)
    End If
    If bDlgーン Then
        Return termProvider.ConstructTerm(20006, "Invalid", DescriptionTermID, "" & Description, 3801, "Type")
    End If
    Return ""
End Function
```

```
Public Shared Function ValidateAccountBlocked(account As AccountAll, termProvider As ITerm, ByVal DescriptionTermID As Integer, ByVal Description As String) As String
    If accountBlocked Then
        Return termProvider.ConstructTerm(DescriptionTermID, Description, 1267, "Blocked")
    End If
    Return ""
End Function
```

Figure 4.15: The final fully testable solution
4.4 Conclusions (RQ3)

**RQ3** What are the obstacles for testability of the Exact Online codebase and how can these testability issues be resolved?

The two main reasons for untestable code are:

1. **Database Access**
2. **Dependencies on Environment**

   To resolve the first obstacle, we propose to move the database calls to a database access shell called `datapropvider`, and injecting a an interface of such a dataprovider into the code under test.

   The second testability obstacle, the dependencies on Environment, occurs in three ways: *method invocation* on Environment, *message chaining* member objects of Environment and *passing* the Environment on. To remove the method invocation dependency, an interface of Environment instead of the Environment itself could be used. The message chaining problem can be resolved by injecting (an interface of) the required member object instead of the Environment. The dependency where the Environment is being passed on can be removed by moving the untestable code to a new (specialized) object, and injecting an interface of the newly created object.
Chapter 5

Testing and Reengineering the Selected Components

This Chapter discusses the unit testing and reengineering procedure being applied to the PriceTools component to form its replacement called the PriceCalculator component, and evaluates the improvements. The main goals are to find and fix bugs, and to make the component more resilient to bug-introducing changes. To do so, we create a unit test harness with a high percentage of code coverage, and improve the quality of the code in general. The identified issues and improvements are not being discussed into details (code level), but at a rather high level. We answer the following research question:

**RQ4** What issues can be identified for the selected components and how can these be resolved?

First in Section 5.1 the creation of a unit test harness is discussed. Then, Section 5.2 discussed the reengineering of the existing into a new component, and in Section 5.3 the process of verification of existing functionality by unit tests is discussed. In Section 5.4 discusses further possible improvements and the decision to skip other components, and final Section 5.5 provides an answer for research question 4.
5.1 Step 1: Creating a Unit Test Harness

The unit testing and reengineering procedure consists of 3 steps. In the first step a unit test harness is created with the following three purposes:

1. Understanding the code (using characterization tests)
2. Creating a unit test harness used later to make sure existing behaviour is retained
3. Identifying issues, ambiguities and bugs in the code

5.1.1 A First Look

The PriceTools component contains three classes: **PriceTools**, **EntryPriceTools**, and **ItemPriceTools**. Figure 5.2 shows a simplified UML diagram of the PriceTools and EntryPriceTools classes.

![UML Diagram](Image)

**Figure 5.2: The PriceTools and EntryPriceTools classes**

The **PriceTools class** contains the core logic that is used to calculate prices including or excluding *Value Added Tax (VAT)*. It needs to be instantiated and configured using the
constructor and setters. The calculate function then needs to be invoked after which the result can be retrieved by a getter. Another calculate function has been added that performs the setup, calculation, and retrieval of the result at once.

The **EntryPriceTools class** only contains a couple of static (also called shared) functions and cannot be instantiated. These functions make use of the PriceTools class to perform calculations, and provide a layer of extra functionality on top of it (e.g. performing the instantiation of PriceTools, calling the calculation function, and retrieving the result at once). The functions of EntryPriceTools depend on the PriceCalculator class to function, and are being invoked in many different places in the codebase.

<table>
<thead>
<tr>
<th>ItemPriceTools</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Constructors</td>
</tr>
<tr>
<td>No Properties (Getter/Setters)</td>
</tr>
<tr>
<td>SQL-Functions</td>
</tr>
<tr>
<td>+GetWhereClauseDefaultSalesPrice()</td>
</tr>
<tr>
<td>+GetWhereClauseDefaultPurchasePrice()</td>
</tr>
<tr>
<td>Simple Functions</td>
</tr>
<tr>
<td>+GetWhereClauseDefaultSalesPrice()</td>
</tr>
<tr>
<td>+GetWhereClauseDefaultPurchasePrice()</td>
</tr>
<tr>
<td>isStandardSalesPrice()</td>
</tr>
<tr>
<td>ItemUnitPricesLeading(...)</td>
</tr>
<tr>
<td>+UnitPricesLeading(...)</td>
</tr>
</tbody>
</table>

Figure 5.3: The ItemPriceTools class

The **ItemPriceTools class** (see Figure 5.3) contains two constants, five static functions that only perform a single function call to another class, and two functions that return SQL-code.

The PriceTools and EntryPriceTools classes contain code that is hard to understand at first sight and seem bug-prone because it looks hard to not break existing functionality when making changes. There is no documentation, there are no unit tests and no integration tests, and therefore this code meets our definition of legacy code (see Section sec:introterminology). The ItemPriceTools class, however, differs from the other two. It does not contain much logic, and thus there is not much to be unit tested. The two functions that return SQL-code could best be tested by integration tests using a real database to verify their correct functioning. Because such integration tests are out of scope of this study, we decided to write a few simple unit tests for them and further leave them untouched. These unit tests check whether the returned SQL-string equals some expected string. Changes to these functions are likely to cause the unit tests to fail, forcing the developers to update these tests and having a look at the resulting SQL-string again.

The bugs that have been fixed for the PriceTools component have all been fixes on the PriceTools or EntryPriceTools classes, and neither the PriceTools nor the EntryPriceTools classes are related to the ItemPriceTools class. Combined with the other factors (lack of complex code and limited unit testability) at the cost of extra migration effort and risk, we decided not to transfer the functionality of the ItemPriceTools class to the reengineered
For the remainder of this Chapter, when the PriceTools component is mentioned, only the PriceTools and EntryPriceTools classes are meant.

5.1.2 Characterization Tests

In order to create a unit test suite that fully covers the PriceTools component, we need to gain a thorough understanding of the PriceTools component first. We do so by writing Characterization Tests. According to Feathers: “A characterization test is a test that characterizes the actual behavior of a piece of code. [...] The tests document the actual behavior of the system”[1].

The functions of the PriceTools and EntryPriceTools classes can be invoked with many parameters set to a null value (“Nothing” in VB.NET). It can easily be found which parameters absolutely need to be set and which can be nothing by looking at the code. Unit tests that leave as many parameters null as possible form the starting point for writing our characterization tests and makes us understand the basics. Once these basics are clear, it becomes much easier to understand the working of each parameter by writing additional unit tests.

Eventually, a total of 33 data-driven\(^1\) unit tests have been written in this phase (accounting for a total of 169 individual tests). Of these tests, 6 are written for the ItemPriceTools class, so for the PriceTools component a total of 27 data-driven unit tests have been written.

Because the source code turned out to be untestable, and no changes are made before having a test harness in place, a mocking framework is used.

5.1.3 Identified Issues

Using the characterization tests, a wide range of issues have been identified. Without giving an exhaustive overview, we discuss the most important issues that are likely to cause the PriceTools component to be bug-prone.

Complex Functions

Some functions assume too much responsibilities causing the code to become tangled and hard to test. As an example, without a mocking framework to replace the calls to database cache functions it is impossible to unit test the calculate function, which is one of the most important functions of the PriceTools component. The main responsibility of the function is to perform a certain calculation, and should not have any knowledge of where the data is being retrieved from.

No clear Behavior

The behavior of the classes and functions are not clear from the outside. There is no documentation of the classes or functions, and by just looking at the interface it is hard to

\(^1\) a unit test that is run repeatedly for each row in a data source
understand how functions are supposed to be used (e.g. what parameters are optional). Even after writing a vast amount of characterization tests, there are some cases left where the expected outcome is still unknown. In these cases (e.g. where certain exceptions are raised) it is impossible to decide whether this is part of the expected behavior or a bug. As an example, the PriceTools class first needs to be set up before the calculate function is called. But it remains unclear what should happen when it has not been properly set up.

**Code Duplication**

Parts of the code have been duplicated, for example because a few conditions that determine how certain calculations should be performed differ. The duplicated code could be extracted into a single generalized function, and called according to the conditions.

**Confusing Interface**

The PriceTools class can partly be set up using the constructor, and partly needs additional set up using setters. This is confusing and may cause unexpected results when not properly set up (e.g. because default values are used).
5.2 Step 2: Reengineering

In the second step of the unit testing and reengineering procedure, the PriceTools component gets reengineered into the PriceCalculator. We now have gained a thorough understanding of the working and what parts need improvements.

The requirements of the new component are:

1. It is covered in unit tests.

2. It provides the same public interface as the non-obsolete EntryPriceTools functions, to allow for smooth migration.

3. It is easier to maintain and require less unit test effort.

4. It should provide the same functionality as the PriceTools component.

In addition, the reengineering possibilities are limited by:

5. The changes should be made within the file.

The component needs to be reengineered with this limitation because (1) otherwise it would heavily complicate the process by limiting the traceability of the new code (technical reason) and (2) having more than one changed file makes it harder to sell the new component to the responsible team as the change impact is higher (political reason).

5.2.1 The Process

The PriceCalculator has been built by duplicating the functions from PriceTools component, and using refactoring techniques such as those described by Feathers in [1]. During the process, additional unit tests have been written to verify that new functionality works as expected and will not be broken by changes made later. This proved to be very useful as existing functionality was indeed broken multiple times.

By using tools that highlight the code coverage of a unit tests, it was made sure that a unit test does indeed “follow the path” that was expected.
5.2.2 Improvements

This Section discusses the improvements made as a result of the reengineering effort. For the sake of easy migration, some additional functions have been added to the PriceCalculator. These functions provide backwards compatibility by providing the interface of the existing functions, and simply call the new functions (by mapping parameters). These functions have not been included in the comparison, to provide a better (more fair) comparison.

Overview

A summary of the changes that have been made to the PriceCalculator in comparison to the PriceTools component:

- Redesigning the code in such a way that the need for proper configuration before performing calculations is removed.
- Split the functions according to responsibility, by introducing small private functions for a specific purpose (untangling spaghetti code).
- Extracted the retrieval of data to new functions, to split calculations (business logic) from data retrieval (database access).
- Added comments to the interface of public functions, describing their use and exceptions.
- Added small comments to the code, making the internal steps clear without the need to understand the code.
- Using the correct types instead of "Object" types in parameters.
- Removed code duplications (e.g. retrieving the same data from the database cache multiple times).
- Introduced enumerations to replace "magic numbers".
- Replaced unclear function names with better names.
- Replaced unclear variable names with clear names.
5. TESTING AND REENGINEERING THE SELECTED COMPONENTS

Maintainability and Code Quality Improvements

The improvements mentioned in the overview intuitively seem to have increased the level of maintainability and improved the quality of the code. However, a better comparison can be made by comparing metrics that are related to maintainability and code quality.

Table 5.1 shows some metrics of the PriceTools and PriceCalculator components, including getters, setters, and constructors. Most of the metrics are measured using Visual Studio 2012 (Code Metrics)\(^2\), and some manually (max nesting level and number of functions).

<table>
<thead>
<tr>
<th>Metric</th>
<th>PriceTools</th>
<th>PriceCalculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. level of nesting</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Cyclomatic complexity (CC)</td>
<td>57</td>
<td>40</td>
</tr>
<tr>
<td>Logical lines of code (LLOC)</td>
<td>107</td>
<td>78</td>
</tr>
<tr>
<td>Unit Test Block Coverage</td>
<td>0%</td>
<td>96.4%</td>
</tr>
</tbody>
</table>

Table 5.1: Metrics including getters, setters, and constructors

The PriceTools class contains a number of simple getter and setter functions and two constructors that do not contain any complex logic. For these functions no separate unit tests are created and do not require reengineering effort. But in this study we are interested in those functions that are worth unit testing and should be refactored. It was therefore decided to leave out getters, setters, obsolete functions, and constructors in the metrics presented in Table 5.2, to be able to better compare only the important functions.

<table>
<thead>
<tr>
<th>Metric</th>
<th>PriceTools</th>
<th>PriceCalculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of functions</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Cyclomatic complexity (CC)</td>
<td>34</td>
<td>39</td>
</tr>
<tr>
<td>Avg. CC per function</td>
<td>5.7</td>
<td>3.3</td>
</tr>
<tr>
<td>Logical lines of code (LLOC)</td>
<td>78</td>
<td>77</td>
</tr>
<tr>
<td>Avg. LLOC per function</td>
<td>13.0</td>
<td>6.4</td>
</tr>
</tbody>
</table>

Table 5.2: Metrics excluding getters, setters, obsolete functions and constructors

The biggest improvement that the measurements show is the doubling of the number of functions. As a result, the average function became less complex (CC per function) and smaller (LLOC per function). Understanding, using, and adapting such smaller functions is easier, thereby the maintainability and quality of the code have been increased.

Testability Improvements

As a consequence of the reengineering procedure, the following most notable testability improvements have been achieved:

1. By splitting some functions it is now possible to test parts of the code individually.

2. For some functions no mocking framework (JustMock) is required any longer, resulting in a significant speed-up (on average, a decrease from 113.4ms to 3.6ms per test has been measured).

In Chapter 3 we estimated the required unit testing effort for the components (see Section 3.4). We can use the same procedure to compare the estimated testing effort of the PriceTools to the PriceCalculator component. Note that the PriceTools component consists of the PriceTools and EntryPriceTools classes, and getters, setters, and constructors are left out so only the interesting functions are compared.

<table>
<thead>
<tr>
<th></th>
<th>PriceTools</th>
<th>PriceCalculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Amount of Tests</td>
<td>34</td>
<td>39</td>
</tr>
<tr>
<td>Estimated Avg. Test Complexity</td>
<td>5.7</td>
<td>3.3</td>
</tr>
<tr>
<td>Estimated Unit Test Effort</td>
<td>193.8</td>
<td>128.7</td>
</tr>
</tbody>
</table>

Table 5.3: Estimated required unit test effort

Based on these estimations, writing a unit test harness for the PriceCalculator component takes 66.4% of the effort that is required to write a unit test harness for the PriceTools component. This estimation matches very well with our experiences.
5.3 Step 3: Verification

5.3.1 Transferring Unit Tests

The goal of the third and final step of the unit testing and reengineering procedure is to ensure that the new component is functionally equal to the existing component. This is done by using the unit tests suite that is created for the PriceTools component as verification. For the functions in PriceCalculator that replace the functions of the EntryPriceTools class, the interface is kept the same to allow a smooth migration. The unit tests that have been written for these functions can therefore be transferred without any modifications. Other unit tests have become obsolete (due to a different structure of the functions in the PriceCalculator components) and some require minor adaptations.

Eventually after migrating the unit tests from the PriceTools component, the PriceCalculator is covered by 29 data-driven (153 individual) unit tests that achieve a unit test block coverage of 96.4%.
5.4 Discussion

Further Improvements

The code quality and testability of the PriceCalculater could be improved even further by splitting up the functions according to (sub)functionality and moving functions (such as those that retrieve data from cache) to the class that it actually belongs to. The reason for not performing these refactorings is because then it will not adhere to the reengineering limitations described in Section 5.2.

The current test-suite and improvements made by the refactorings are sufficient for this study, so therefore it is not necessary to perform such additional refactorings.

5.4.1 Skipping LogisticsTools

To increase the data set that is used for answering the research question, it was planned to repeat the reengineering procedure for other components such as LogisticsTools. However, while writing characterization tests it became clear that this component contains a lot of functions that should actually be moved to the class they “do something with”. These components can not be used for our research as splitting them up would cause them to become untraceable by our bug repository mining approach.

As an example the function GetVATReferenceDate (figure 5.4) decides for a given BusinessComponent whether that BusinessComponent is an exceptional case and returns a special or default property accordingly. It does so by retrieving the name of that BusinessComponent and returns the corresponding property when the name matches one of the exceptions.

One of the vulnerabilities in the example that we would like to solve is the fact that for a newly written BusinessComponent it should be manually checked by the programmer whether this is an exceptional case. When it is indeed an exceptional case, the programmer must add code to the GetVATReferenceDate function in LogisticsTools such that the correct property is being returned instead of the default value. Because this code is located in a different class and file, it can easily be forgotten and cause a bug. This vulnerability can be solved in the following two ways:

1. by moving the function that is currently located in the LogisticsTools class to the class of the component, and forcing by design (using inheritance) that a new type of the component should explicitly implement the function that either returns the default or a special value.

2. by making a unit test fail whenever a new type of the component is being added while the expected result for this type has not been added to the unit test.

The problem with the first option is that reengineering would require too much effort since some fundamental changes would be required. Also lots of extra tests would have to be written in order to make sure that nothing elsewhere in the system breaks (tests that might even influence this study). In addition, comparing the old version of the LogisticsTools class
5. Testing and Reengineering the Selected Components

```vbnet
Public Shared Function GetVATReferenceDate(BC As BusinessComponent, ...) As Object

... 

Dim propertyName As String = "EntryDate"
Select Case BC.BcName
    Case "SalesQuotationLine"
        propertyName = "QuotationDate"
    Case "SalesOrderLine"
        propertyName = "DeliveryDate"
        ' Should be added in scope of fix VAT for Purchase orders
    Case "PurchaseOrderLine"
        propertyName = "ReceiptDate"
    Case "SubscriptionLine"
        propertyName = "FromDate"
End Select
If BC.HasProp(propertyName) Then
    refDate = BC.Prop(propertyName).Value
End If
...

Return refDate
End Function
```

Figure 5.4: The GetVATReferenceDate function in LogisticsTools.

with the new version would become very difficult, if possible at all. The combination of too much required effort with low usability for this study renders this option useless.

The second possibility to solve the vulnerability is a trick that manages the problem as a workaround rather than correctly solving it. It requires a unit test to be able to loop over all the possible types that can be used as input for the function.

For each of these input values, the expected outcome of the test needs to be known in advance. Whenever a new type of component is added by a programmer, the expected outcome for that type has to be added to the unit test or the unit test will fail. This forces the programmer to explicitly state whether the new component is an exceptional case. Unfortunately for many of the functions in the LogisticsTools class, a string instead of enum is used so the possibilities are unknown. Therefore writing a unit test as described above is impossible for these functions.

Multiple functions in the LogisticsTools component contain the described vulnerability where both the first and second option to resolve it are very difficult or impossible. Therefore it has been concluded that the LogisticsTools component will not be reengineered and tested, and is excluded from this study.
5.5 Conclusions (RQ4)

5.5.1 Reengineering Conclusions

The main goal of the reengineering procedure is to replace the bug-prone PriceTools component with the PriceCalculator component that:

1. is covered with unit tests
2. is easy to migrate
3. is better maintainable
4. is easier to test
5. keeps existing behavior
6. is implemented in one file

At the start, the PriceTools component was not covered by any unit tests. Its replacement, the PriceCalculator, is covered by unit tests that achieve a block coverage of 96.4%, where the most important functions (such as those performing calculations) are tested extensively.

A smooth migration to the PriceCalculator is guaranteed by providing the same interface that is provided by the EntryPriceTools. These functions are the ones that are being used in many different places in the codebase. Because the interfaces themselves are flawed (using “Object” types instead of the correct subtypes), a comment is added in which developers are suggested to use the replacements of these functions instead.

The maintainability of the PriceCalculator component has increased due to improvements such as splitting the functions according to responsibility, adding comments, and removing code duplications.

We estimate that writing a unit test harness for the PriceCalculator component takes 66.4% of the effort that is required to write a unit test harness for the PriceTools component. In addition, most functionality can now be tested individually (because functions have been split) and some of the important functions can now be tested without using a mocking framework. The PriceCalculator component is therefore much easier to test.

By using the unit tests that have been written for the PriceTools component as verification of the behavior of the reengineered PriceCalculator component, it has been verified that the behavior is indeed equal and the PriceCalculator thus forms a correct replacement.

The PriceCalculator component consists of only the PriceCalculator class and is implemented in the file called “PriceCalculator.vb”. Therefore, it can easily be traced using
5. Testing and Reengineering the Selected Components

the bug repository mining approach described in Chapter 2.

The combination of improvements makes the PriceCalculator component less likely to contain any (latent) bugs and more resilient to bug-introducing changes.

5.5.2 Research Question 4

RQ4 What issues can be identified for the selected components and how can these be resolved?

The four most important issues that cause the PriceTools component to be bug-prone are:

1. Complex functions
2. No clear behavior of the classes and functions
3. Duplicated code
4. Confusing interface

These issues have been resolved by the combination of a number of refactorings, of which the most important are:

- splitting the complex functions into small, clear functions.
- making decisions about corner cases and documenting them in comments
- removing duplicated code by introducing a new function
- replacing the “Object” types in parameters with the correct types
- improving the readability by lowering the depth of nestings and adding small comments

In addition, the unit test harness that is being executed on a daily basis prevents the existing behavior from changing unexpectedly, which makes it much more resilient to bug-introducing changes.
Chapter 6

Switching to Classical Testing

In Chapter 4 we identified and discussed the main reasons for untestable code. A testing engineer is promoting these concepts and the use of classical testing by giving training sessions to other developers. Although this subject is not a necessity to answering our main research question, it provides us the opportunity to evaluate how these developers perceive the concepts and how they experience the switch from a mockist to a classical unit testing approach. In this chapter, we present and discuss the results of a survey.

With the results of the questionnaire presented in this Chapter, we seek an answer to the following research question:

RQ5 Do the developers understand that applying the classical unit testing approach is reducing the legacy problem?

6.1 Introduction

The developers are used to working with legacy code and sometimes write unit tests using a mocking framework to mock out the dependencies. Although the developers do have the opportunity to write unit tests this way, it is our experience that this does not happen very often. In any case, the developers are not forced to writing testable code, and the legacy problem is not being reduced but instead grows further. By making the developers understand the advantages of the classical approach, the production of new legacy code can be stopped by writing testable code, and the existing legacy code can be reduced.
6. SWITCHING TO CLASSICAL TESTING

The questions (Survey Questions) that we would like to have answered with the survey are the following:

**SQ1** Do the developers understand why they should write unit tests? Can they give reasons other than “finding bugs”?

**SQ2** Do the developers perceive classical testing as a better and more enjoyable approach than mockist testing?

**SQ3** What do the developers believe are obstacles of classical testing compared to mockist testing?

**SQ4** Are the developers likely to start writing testable code?

**SQ5** Do the developers consider writing testable code a burden?

**SQ6** Do the developers aim at achieving a certain minimal percentage of code coverage with unit tests?

**SQ7** Given a list of reasons for writing unit tests, how would the developers prioritize them?

6.2 Method

The survey is being filled in approximately two weeks after the training session, and before a feedback session, so the developers have had the opportunity to gain some experience with the new approach and the feedback can be used in the feedback session.

Some of the respondents have answered some of the questions wrong. Especially for the prioritization question (question 14), the respondents have often used the same priority multiple times instead of each once. These responses have been filtered out.

The survey results have been split into two groups, based on the team the respondent is in. These groups are:

**Developers of functional teams** They form the main target audience of the survey and have gained experience in classical testing by attending the training session.

**Developers of other teams** Developers that gained their experience in classical testing without the training session. This group is considered to be more experienced in unit testing. Examples of the teams they are in are research and platform.

By splitting these groups, a comparison can be made between the developers that are relatively new to classical testing and the developers that are more experienced. All respondents have a minimal (training) experience with both unit testing with a mocking framework (mostly JustMock) and classical testing.
6.3 Survey Results

6.3.1 Question 1

“What do you believe are the most important reasons for writing unit tests?”

Functional Teams

<table>
<thead>
<tr>
<th>Reason</th>
<th>Times Mentioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improving regression efficiency / As safety net</td>
<td>6</td>
</tr>
<tr>
<td>Improving code quality</td>
<td>5</td>
</tr>
<tr>
<td>Improving system quality in general</td>
<td>5</td>
</tr>
<tr>
<td>Finding bugs</td>
<td>3</td>
</tr>
<tr>
<td>Improving code maintainability</td>
<td>3</td>
</tr>
<tr>
<td>Increasing readability</td>
<td>2</td>
</tr>
<tr>
<td>Ensuring code correctness</td>
<td>2</td>
</tr>
<tr>
<td>Improving code testability</td>
<td>1</td>
</tr>
<tr>
<td>Increasing code analyzability</td>
<td>1</td>
</tr>
<tr>
<td>As prove of testing effort</td>
<td>1</td>
</tr>
<tr>
<td>Promoting reusability</td>
<td>1</td>
</tr>
<tr>
<td>Supporting code refactoring</td>
<td>1</td>
</tr>
<tr>
<td>Increase productivity</td>
<td>1</td>
</tr>
</tbody>
</table>

Other Teams

<table>
<thead>
<tr>
<th>Reason</th>
<th>Times Mentioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improving regression efficiency / As safety net</td>
<td>6</td>
</tr>
<tr>
<td>Improving code quality</td>
<td>4</td>
</tr>
<tr>
<td>Early bug prevention</td>
<td>3</td>
</tr>
<tr>
<td>Maintaining system integrity</td>
<td>3</td>
</tr>
<tr>
<td>Improving code maintainability</td>
<td>2</td>
</tr>
<tr>
<td>Documenting function requirements</td>
<td>2</td>
</tr>
<tr>
<td>Ensuring code correctness</td>
<td>2</td>
</tr>
<tr>
<td>Getting control over technical debt</td>
<td>1</td>
</tr>
<tr>
<td>Supporting code refactoring</td>
<td>1</td>
</tr>
<tr>
<td>Guarding the contract of your class</td>
<td>1</td>
</tr>
<tr>
<td>Better understanding of dependencies</td>
<td>1</td>
</tr>
<tr>
<td>The ability to write scenarios with predictable outcomes</td>
<td>1</td>
</tr>
<tr>
<td>Verifying that all function points are touched</td>
<td>1</td>
</tr>
<tr>
<td>Providing functionality examples for other developers</td>
<td>1</td>
</tr>
</tbody>
</table>
6. SWITCHING TO CLASSICAL TESTING

6.3.2 Question 2

“How often do you write unit tests?”
Possible answers are on a 5-point scale from 1 (never) to 5 (all the time).

Functional Teams

![Chart showing the distribution of responses for Functional Teams]

1 Never: 0 0%
2 Almost never: 13 76%
3 Sometimes: 3 18%
4 Often: 1 6%
5 All the time: 0 0%

Median: 2
Mean: 2.3

Other Teams

![Chart showing the distribution of responses for Other Teams]

1 Never: 1 8%
2 Almost never: 5 42%
3 Sometimes: 5 42%
4 Often: 0 0%
5 All the time: 1 8%

Median: 2.5
Mean: 2.6
6.3.3 Question 3

“How experienced in unit testing with JustMock do you consider yourself?”
Possible answers are on a 5-point scale from 1 (no experience) to 5 (expert).

**Functional Teams**

- No experience: 5 (29%)
- Beginner: 9 (53%)
- Intermediate: 3 (18%)
- Advanced: 0 (0%)
- Expert: 0 (0%)

Median: 2
Mean: 1.9

**Other Teams**

- No experience: 3 (25%)
- Beginner: 4 (33%)
- Intermediate: 4 (33%)
- Advanced: 1 (8%)
- Expert: 0 (0%)

Median: 2
Mean: 2.3
6. Switching to Classical Testing

6.3.4 Question 4

“How experienced in classical testing do you consider yourself?”
Possible answers are on a 5-point scale from 1 (no experience) to 5 (expert).

Functional Teams

<table>
<thead>
<tr>
<th>Experience Level</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>No experience</td>
<td>6%</td>
</tr>
<tr>
<td>Beginner</td>
<td>53%</td>
</tr>
<tr>
<td>Intermediate</td>
<td>29%</td>
</tr>
<tr>
<td>Advanced</td>
<td>12%</td>
</tr>
<tr>
<td>Expert</td>
<td>0%</td>
</tr>
</tbody>
</table>

Median: 2
Mean: 2.5

Other Teams

<table>
<thead>
<tr>
<th>Experience Level</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>No experience</td>
<td>0%</td>
</tr>
<tr>
<td>Beginner</td>
<td>25%</td>
</tr>
<tr>
<td>Intermediate</td>
<td>33%</td>
</tr>
<tr>
<td>Advanced</td>
<td>33%</td>
</tr>
<tr>
<td>Expert</td>
<td>8%</td>
</tr>
</tbody>
</table>

Median: 3
Mean: 3.3
6.3.5 Question 5

“How would you rate the approach of unit testing with JustMock?”
Possible answers are on a 5-point scale from 1 (useless) to 5 (fantastic).

**Functional Teams**

<table>
<thead>
<tr>
<th>Rating</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Useless</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Not so good</td>
<td>7</td>
<td>41%</td>
</tr>
<tr>
<td>Usable</td>
<td>8</td>
<td>47%</td>
</tr>
<tr>
<td>Great</td>
<td>2</td>
<td>12%</td>
</tr>
<tr>
<td>Fantastic</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Median: 3  
Mean: 2.7

**Other Teams**

<table>
<thead>
<tr>
<th>Rating</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Useless</td>
<td>1</td>
<td>11%</td>
</tr>
<tr>
<td>Not so good</td>
<td>2</td>
<td>22%</td>
</tr>
<tr>
<td>Usable</td>
<td>4</td>
<td>44%</td>
</tr>
<tr>
<td>Great</td>
<td>2</td>
<td>22%</td>
</tr>
<tr>
<td>Fantastic</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Median: 3  
Mean: 2.8
6. Switching to Classical Testing

6.3.6 Question 6

“How much do you enjoy writing unit tests with JustMock?”
Possible answers are on a 5-point scale from 1 (I hate it) to 5 (I love it).

Functional Teams

![Bar chart showing the distribution of responses for Functional Teams]

- 1 I hate it: 2 (12%)
- 2 I don’t like it: 6 (35%)
- 3 Neutral: 7 (41%)
- 4 I like it: 1 (6%)
- 5 I love it: 1 (6%)

Median: 3
Mean: 2.6

Other Teams

![Bar chart showing the distribution of responses for Other Teams]

- 1 I hate it: 2 (22%)
- 2 I don’t like it: 4 (44%)
- 3 Neutral: 2 (22%)
- 4 I like it: 0 (0%)
- 5 I love it: 1 (11%)

Median: 2
Mean: 2.3
6.3.7 Question 7

“How would you rate the approach of classical testing?”
Possible answers are on a 5-point scale from 1 (useless) to 5 (fantastic).

**Functional Teams**

![Bar chart showing ratings for Functional Teams]

1 Useless: 1 6%
2 Not so good: 3 18%
3 Usable: 5 29%
4 Great: 7 41%
5 Fantastic: 1 6%

Median: 3
Mean: 3.2

**Other Teams**

![Bar chart showing ratings for Other Teams]

1 Useless: 0 0%
2 Not so good: 0 0%
3 Usable: 2 17%
4 Great: 7 58%
5 Fantastic: 3 25%

Median: 4
Mean: 4.1
6. Switching to Classical Testing

6.3.8 Question 8

“How much do you enjoy writing unit tests with the classical approach?”
Possible answers are on a 5-point scale from 1 (I hate it) to 5 (I love it).

Functional Teams

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 I hate it:</td>
<td>1 6%</td>
</tr>
<tr>
<td>2 I don't like it:</td>
<td>2 12%</td>
</tr>
<tr>
<td>3 Neutral:</td>
<td>8 47%</td>
</tr>
<tr>
<td>4 I like it:</td>
<td>5 29%</td>
</tr>
<tr>
<td>5 I love it:</td>
<td>1 6%</td>
</tr>
</tbody>
</table>

Median: 3
Mean: 3.2

Other Teams

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 I hate it:</td>
<td>0 0%</td>
</tr>
<tr>
<td>2 I don't like it:</td>
<td>0 0%</td>
</tr>
<tr>
<td>3 Neutral:</td>
<td>3 25%</td>
</tr>
<tr>
<td>4 I like it:</td>
<td>5 42%</td>
</tr>
<tr>
<td>5 I love it:</td>
<td>4 33%</td>
</tr>
</tbody>
</table>

Median: 4
Mean: 4.1
6.3.9 Question 9

“What do you believe are the advantages of classical testing (testing without JustMock) over testing with JustMock? Please list them.”

Functional Teams

<table>
<thead>
<tr>
<th>Advantage</th>
<th>Times Mentioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependency on a third party framework</td>
<td>8</td>
</tr>
<tr>
<td>Easier to learn and use</td>
<td>6</td>
</tr>
<tr>
<td>Faster test execution</td>
<td>4</td>
</tr>
<tr>
<td>Better focus on testing rather than mocking</td>
<td>2</td>
</tr>
<tr>
<td>Promotes refactorings</td>
<td>2</td>
</tr>
<tr>
<td>Being able to better integrate with the existing framework</td>
<td>2</td>
</tr>
<tr>
<td>Faster delivery to customer</td>
<td>1</td>
</tr>
<tr>
<td>Classical testing seems to be less blackbox testing</td>
<td>1</td>
</tr>
<tr>
<td>More realistic tests</td>
<td>1</td>
</tr>
<tr>
<td>More reliable</td>
<td>1</td>
</tr>
<tr>
<td>Being able to test unconventional scenarios</td>
<td>1</td>
</tr>
<tr>
<td>Easier to trace down the point of failure</td>
<td>1</td>
</tr>
</tbody>
</table>

Other Teams

<table>
<thead>
<tr>
<th>Advantage</th>
<th>Times Mentioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testable code improves the design and maintainability</td>
<td>5</td>
</tr>
<tr>
<td>Faster test execution</td>
<td>4</td>
</tr>
<tr>
<td>Test are easier to understand</td>
<td>3</td>
</tr>
<tr>
<td>More reliable</td>
<td>2</td>
</tr>
<tr>
<td>More accountability among unit test writers to setup good tests</td>
<td>2</td>
</tr>
<tr>
<td>Easier deployment</td>
<td>1</td>
</tr>
<tr>
<td>No need to adjust the build process</td>
<td>1</td>
</tr>
<tr>
<td>Easier to use</td>
<td>1</td>
</tr>
<tr>
<td>No dependencies between unit tests</td>
<td>1</td>
</tr>
<tr>
<td>No need to learn a new framework</td>
<td>1</td>
</tr>
<tr>
<td>No dependency on a third party framework</td>
<td>1</td>
</tr>
</tbody>
</table>
6. Switching to Classical Testing

6.3.10 Question 10

“What do you believe are the disadvantages of classical testing (testing without JustMock) to testing with JustMock? Please list them.”

Functional Teams

<table>
<thead>
<tr>
<th>Disadvantage</th>
<th>Times Mentioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires time consuming refactoring effort</td>
<td>8</td>
</tr>
<tr>
<td>Refactoring adds risk</td>
<td>2</td>
</tr>
<tr>
<td>The writing of test code (fakes)</td>
<td>1</td>
</tr>
<tr>
<td>Unable to mock private functions</td>
<td>1</td>
</tr>
<tr>
<td>Classical testing lacks standardization amongst developers</td>
<td>1</td>
</tr>
<tr>
<td>A team needs to wait for adjustments to the fakes</td>
<td>1</td>
</tr>
</tbody>
</table>

Other Teams

<table>
<thead>
<tr>
<th>Disadvantage</th>
<th>Times Mentioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires time consuming refactoring effort</td>
<td>3</td>
</tr>
<tr>
<td>Cannot be used as safety net before refactoring</td>
<td>3</td>
</tr>
<tr>
<td>More testing code required</td>
<td>2</td>
</tr>
<tr>
<td>Makes you less productive</td>
<td>2</td>
</tr>
<tr>
<td>Code might become less readable</td>
<td>1</td>
</tr>
<tr>
<td>After a learning curve, JustMock is superior</td>
<td>1</td>
</tr>
<tr>
<td>More emphasis on the unit test writer to setup own scenarios</td>
<td>1</td>
</tr>
</tbody>
</table>
6.3.11 Question 11

“How likely is it that you will write testable code? (e.g. by splitting functions and using interfaces)”

Possible answers are on a 5-point scale from 1 (very unlikely) to 5 (very likely).

Functional Teams

1 Very unlikely: 0 0%
2 Unlikely: 3 17%
3 Possibly: 7 39%
4 Likely: 6 33%
5 Very likely: 2 11%

Median: 3
Mean: 3.4

Other Teams

1 Very unlikely: 0 0%
2 Unlikely: 0 0%
3 Possibly: 3 25%
4 Likely: 5 42%
5 Very likely: 4 33%

Median: 4
Mean: 4.1
6. Switching to Classical Testing

6.3.12 Question 12

“Do you consider writing testable code during development an advantage or a burden? Why?”

Functional Teams

Most developers that consider writing testable code an advantage mention the “improvement of code quality”, “improvement of maintainability” and “being able to see more clearly what scenarios need to be tested” as reasons. One of the developers that does not know whether writing testable code should be considered an advantage or burden is concerned about the usefulness of refactoring existing code, because it is “time consuming” and “risky”.

Other Teams

The developers from this group mention the “improvement of code quality and design” as main reason for seeing writing testable code as an advantage. However, some also criticise writing testable code because “it might take the speed out of the implementation when prototyping”, “it adds time to the development process” and “the time costs are higher than the benefits when fixing a bug”. These developers actually liked to answer the first part of the question with “Both”, and used either “Advantage” or “Don’t know” as answer.
6.3.13 Question 13

“Do you aim at achieving a minimal code coverage? If yes, what percentage?”
Possible answers: “No”, “Yes (20%)”, “Yes (40%)”, “Yes (60%)”, “Yes (80%)” and “Yes (100%)”.

Functional Teams

| Yes: 14 83% |
| No: 3 17% |
| 20%: 1 7% |
| 40%: 2 13% |
| 60%: 5 33% |
| 80%: 6 40% |
| 100%: 1 7% |
Median: 60%
Mean: 65%

Other Teams

| Yes: 9 75% |
| No: 3 25% |
| 20%: 0 0% |
| 40%: 0 0% |
| 60%: 3 33% |
| 80%: 4 44% |
| 100%: 2 22% |
Median: 80%
Mean: 78%
6. Switching to Classical Testing

6.3.14 Question 14

“Please prioritize the following reasons from priority 1 (most important) to priority 6 (least important). Use each priority once!”

The given reasons that need to be prioritized:

- Finding bugs
- Trust in own code
- Trust in code of other developers
- Documentation of source code
- Improving design of source code
- Fast production of correct code

The results are ranked on median.

**Functional Teams**

The results are based on 10 responses.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Reason</th>
<th>Median</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Improving design of source code</td>
<td>2</td>
<td>2.4</td>
</tr>
<tr>
<td>2</td>
<td>Trust in own code</td>
<td>2.5</td>
<td>2.9</td>
</tr>
<tr>
<td>3</td>
<td>Finding bugs</td>
<td>3</td>
<td>2.9</td>
</tr>
<tr>
<td>4</td>
<td>Trust in code of other developers</td>
<td>3</td>
<td>3.8</td>
</tr>
<tr>
<td>5</td>
<td>Faster production of correct code</td>
<td>3.5</td>
<td>3.7</td>
</tr>
<tr>
<td>6</td>
<td>Documentation of source code</td>
<td>6</td>
<td>5.3</td>
</tr>
</tbody>
</table>

**Other Teams**

The results are based on 11 responses.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Reason</th>
<th>Median</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Improving design of source code</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>2</td>
<td>Trust in own code</td>
<td>2</td>
<td>2.7</td>
</tr>
<tr>
<td>3</td>
<td>Finding bugs</td>
<td>3</td>
<td>3.2</td>
</tr>
<tr>
<td>4</td>
<td>Trust in code of other developers</td>
<td>4</td>
<td>3.8</td>
</tr>
<tr>
<td>5</td>
<td>Documentation of source code</td>
<td>4</td>
<td>3.9</td>
</tr>
<tr>
<td>6</td>
<td>Faster production of correct code</td>
<td>6</td>
<td>4.9</td>
</tr>
</tbody>
</table>
6.4 Conclusions (RQ5)

6.4.1 Survey Conclusions

SQ1 Do the developers understand why they should write unit tests? Can they give reasons other than “finding bugs”?

Yes, the developers from both teams mention reasons such as “as safety net”, “improving code quality”, and “improving maintainability” at least as many times as “finding bugs”. A notable difference between the teams is the fact that the developers from functional teams mention “finding bugs” as reason whereas the developers from the more experienced teams mention “early bug prevention” instead.

SQ2 Do the developers perceive classical testing as a better and more enjoyable approach than mockist testing?

The developers from the functional teams rate the classical testing approach slightly higher (both “usable”), and the developers from the other teams rate the approach of classical testing much higher (from “usable” to “great”). The developers from the functional teams also enjoy writing unit tests with the classical testing approach a little more (both “neutral”), whereas the developers from the other teams enjoy the classical approach much more (from “I don’t like it” to “I like it”).

SQ3 What do the developers believe are the advantages and obstacles of classical testing compared to mockist testing?

The developers from the functional teams massively mention “not having to use learn and rely on a third party framework” as biggest advantage, whereas the developers from the other group mention reasons such as “testable code improves the design and maintainability” and “faster test execution” as biggest advantages. The developers from both groups mention that classical testing “requires time consuming refactoring effort” as most important disadvantage. The developers from the other group also note that the “classical approach cannot be used as safety net before any refactorings”.

SQ4 Are the developers likely to start writing testable code?

The developers in functional teams “possibly” start writing testable code, and the developers from other teams are “likely” to start writing testable code.

SQ5 Do the developers consider writing testable code a burden?

No, not a single developer considers writing testable code a burden. From the functional teams 83% consider writing testable code during development an advantage while 91% of the developers from the other group consider writing testable code an advantage. However, a few developers are concerned about the usability of refactoring existing code as this might be “risky”, “take the speed out of the implementation when prototyping”, and be “time consuming” (especially when fixing a bug).
SQ6 Do the developers aim at achieving a certain minimal percentage of code coverage with unit tests?

83% of the developers from the functional teams and 75% of the developers in the other group answer “yes”. Of those developers, the developers in the functional teams focus on achieving 60% code coverage whereas the developers from the other group aim at achieving 80% code coverage.

SQ7 Given a list of reasons for writing unit tests, how would the developers prioritize them?

Both groups rank “improving design of source code” and “trust in own code” as most important reasons, and “faster production of source code” and “documentation as source code” as least important. The reason “finding bugs” ends up on the third place for both groups. The only difference between the rankings of the two groups is the slightly higher ranking of “documentation of source code” by the other group.

6.4.2 Research Question 5

RQ5 Do the developers understand that applying the classical unit testing approach is reducing the legacy problem?

They partially do. The developers seem to understand the advantages of testable code to the maintainability of it and its use as safety net, but at the same time are afraid to break existing functionality during refactoring and do not seem to believe that the refactoring effort is worth the investment. The developers do not consider themselves to be very experienced in unit testing and do not write unit tests often. In addition, the developers from the functional teams do not really enjoy writing unit tests (either with or without a mocking framework). The combination of lack of experience and not really enjoying writing unit tests could be the reason why they do not believe that refactoring existing code is worth the effort. However for new code the developers do state that they are likely to write testable code, meaning a slowdown in the growth of legacy code.
Chapter 7

Discussion

Figure 7.1: Research Procedure

In this Chapter we discuss the migration of the old PriceTools to the new PriceCalculator component in Section 7.1, and discuss the results of the bug measurement over the period that the replacement was active in Section 7.2. Section 7.3 discusses the threats to the validity of our results.

7.1 Migration process from PriceTools to PriceCalculator

Figure 1.2 in Section 1.4 shows a \textit{replacement active} period of 2 months, from the 13th of May to the 13th of July. In fact, this period slightly differs since the PriceCalculator component has been migrated over a period of time instead of at once. The precise migration dates are as follows:

\textbf{26th of April} First part (\textit{purchase}).

\textbf{7th of May} Second part (\textit{sales}).

\textbf{8th of May} Demigration of a minor part.

\textbf{16th of May} Remigration to the new component.

On the 8th of May, a small part has accidentally been migrated back to the old component (the PriceTools component). This has been fixed by the remigration on the 16th of
7. Discussion

May. Overall, we believe it is fair to say that the PriceCalculator component has been in use for at least 2 months. After full migration, the PriceCalculator component is used on a daily basis by an estimated 35% of the customers of Exact Online.

7.2 Results for the Main Question

Using the second bug set snapshot made on the 13th of July (see Figure 1.2 in Section 1.4) we extract the number of bugs that have been fixed for the components in the evaluation period.

Figure 7.2 shows the list of components along with the predicted bug-proneness (bug score), the number of bugs that have been fixed during the prediction period (bugcount pre), and the number of bugs that have been fixed in the 5 subsequent months after the prediction (bugcount post). The components that required at least one bug to be fixed have been highlighted.

<table>
<thead>
<tr>
<th>Component (File)</th>
<th>Feasibility Score</th>
<th>Test Effort</th>
<th>Pre (6 months)</th>
<th>Post (5 months)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact.Financial.UI.Controls/VATPercentagesComboBox.vb</td>
<td>12.40</td>
<td>121.00</td>
<td>1500</td>
<td>3</td>
</tr>
<tr>
<td>Exact.QuoteCore/QuoteCoreDetailTools.vb</td>
<td>9.31</td>
<td>107.37</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>Exact.Logistics.Core/PriceCalculator.vb*</td>
<td>7.39</td>
<td>148.78</td>
<td>1100</td>
<td>3</td>
</tr>
<tr>
<td>Exact.XML/NodeProvider.vb</td>
<td>2.64</td>
<td>369.32</td>
<td>1050</td>
<td>2</td>
</tr>
<tr>
<td>Exact.MailBox.Job/LinkBankAccount.vb</td>
<td>2.58</td>
<td>406.69</td>
<td>1050</td>
<td>3</td>
</tr>
<tr>
<td>Exact.Logistics.Core/LogisticsTools.vb</td>
<td>2.49</td>
<td>804.76</td>
<td>2000</td>
<td>3</td>
</tr>
<tr>
<td>Exact.Contract.Core/ContractTools.vb</td>
<td>2.16</td>
<td>1944.49</td>
<td>4200</td>
<td>9</td>
</tr>
<tr>
<td>Exact.VAT.Core/VATTools.vb</td>
<td>2.16</td>
<td>1460.51</td>
<td>3150</td>
<td>6</td>
</tr>
<tr>
<td>Exact.MailBox.Core/MailBoxTools.vb</td>
<td>2.15</td>
<td>511.17</td>
<td>1100</td>
<td>4</td>
</tr>
<tr>
<td>Exact.Sales.Orders/CreateInvoice.vb</td>
<td>1.86</td>
<td>538.24</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>Exact.Cashflow/Link/Tools.vb</td>
<td>1.41</td>
<td>709.25</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>Exact.Logistics.Core/ItemTools.vb</td>
<td>1.26</td>
<td>875.68</td>
<td>1100</td>
<td>3</td>
</tr>
<tr>
<td>Exact.Cashflow/MatchFinder.vb</td>
<td>1.06</td>
<td>1416.53</td>
<td>1500</td>
<td>2</td>
</tr>
<tr>
<td>Exact.Cashflow/NU/MT1940.vb</td>
<td>0.65</td>
<td>3918.76</td>
<td>2550</td>
<td>6</td>
</tr>
<tr>
<td>Exact.Topics.Core/TopicTools.vb</td>
<td>0.62</td>
<td>1699.32</td>
<td>1050</td>
<td>2</td>
</tr>
<tr>
<td>Exact.VAT.Return/VATReturn.vb</td>
<td>0.38</td>
<td>2912.56</td>
<td>1100</td>
<td>3</td>
</tr>
<tr>
<td>Exact.Cashflow/Cashflow/Planning.vb</td>
<td>0.32</td>
<td>8022.24</td>
<td>2550</td>
<td>4</td>
</tr>
<tr>
<td>Exact.Core/Environment/Environment.vb</td>
<td>0.24</td>
<td>2256.20</td>
<td>550</td>
<td>2</td>
</tr>
<tr>
<td>Exact.Jobs.NL/BAPI_OnDemand.vb</td>
<td>0.11</td>
<td>4812.36</td>
<td>550</td>
<td>2</td>
</tr>
<tr>
<td>Exact.Financial.BE/GLAccount.vb</td>
<td>0.01</td>
<td>150416.67</td>
<td>1000</td>
<td>1</td>
</tr>
</tbody>
</table>

*The PriceCalculator component is added as reference, it has been in use for 2 months

Figure 7.2: Component bug scores six months before and five months after the snapshot of February 13 2013

In the two months that the PriceCalculator component has served as replacement of the PriceTools component, it required no new bug fixes. For the other components in the top 20, on average 0.842 bugs have been fixed in a period of two months. In the evaluation period, 63.2% of the other components required at least one bug to be fixed.
In addition to these results, no bugs have been fixed for the (old) PriceTools component when it was still active in the first 3 months of the evaluation period.

7.3 Threats to Validity

7.3.1 Incomplete Bug Sets

We argue that for studies that make use of bug data sets (for bug prediction and evaluation purposes), the characteristics of the development and bug administration procedures should be included. In Section 2.1.5 we gave an overview of factors that influence the success of the bug set approach, and provide an evaluation of these factors for this study to gain a level of confidence for the correctness of the used bug sets. Although the configuration for EOL development looks promising and we thus expect a low chance of bias, this is just an indication and therefore forms a threat to the validity of our results. We have for example found evidence that bugs go incognito.

7.3.2 Test Effort Estimation

The estimation of the test effort that is required to reengineer and unit test a component forms an important part of our study. Although it is based on proven concepts (e.g. using cyclomatic complexity as indicator for the number of required tests [23]), we have no verification of the overall performance of our test effort estimator.

7.3.3 Excluding Other Factors

We leave other factors such as code churn out of our research, while the number of changes that have been made to a component is very likely to influence the bug-proneness of a component. For example for a component for which no changes have been made, no future bug fixes might be required. We could then theoretically draw the wrong conclusion that the decrease in bugs was due to reengineering and unit testing effort. To mitigate this effect, either code churn needs to be included, or a much bigger data set needs to be used.
Chapter 8

Related Work

Being able to determine what parts of a software system are most likely to cause failures of the system is very valuable, and therefore well studied by many researchers. Section 8.1 is devoted to related work that discusses approaches for predicting defects in software systems. Section 8.2 discusses research related to the testability of software, and finally an overview of related work that discusses reengineering techniques that focus on testability of a legacy system is given in section 8.3.

8.1 Defect Prediction

“Knowing the locations of future software defects allows project managers to optimize the resources available for the maintenance of a software project by focusing on the most problematic components” [24]. The subject of defect prediction is still a hot topic in software research, even though it has been investigated by many researchers over the past decades. This is caused by the fact that defect prediction is a challenging problem, for which many approaches have been proposed. Catal et al. provide extensive reviews of software fault prediction literature in [25] and [26]. The research field can be divided into two main categories: (1) the category that uses historical data and (2) the category that uses code complexity metrics in order to determine the defect-proneness of components.

A relatively new and interesting way to gain insight into the most vulnerable parts of a software system is to extract usable information from software repositories, such as version control systems and bug trackers. This technique, “Mining Software Repositories” (MSR), falls into the category that uses historical data to predict future failures. An overview of research related to a particular MSR technique called “Mining Bug Repositories” is given in section 8.1.1. Next, literature that discusses the use of software complexity metrics for predicting defects is described in section 8.1.2, and research that focuses on combining historical data with repository mining techniques to form predictors is discussed in section 8.1.3. There are also multiple other approaches proposed for defect prediction such as using code churn [27, 28], using process metrics extracted from versioning system repositories [12], and approaches that use the information contained in e-mail archives [24]. However
because these approaches fall out of the scope of this study, they are not being discussed here.

### 8.1.1 Mining Bug Repositories

The information that is captured inside software repositories can be “mined” in support of multiple tasks, these include:

- predicting bugs, bug density, and the likelihood of component failure (i.e. [2, 29, 30])
- predicting whether a change introduces bugs (i.e. [17, 31, 32])
- investigating other research problems (i.e. Zaidman et al. [33])
- predicting the fix time of a certain bug (i.e. Giger et al. [34])
- suggesting a developer to fix a bug (i.e. Anvik et al. [35])
- taking managerial decisions (i.e. as discussed by D’Ambros et al. in [36] and Arisholm et al. in [13])

Jung et al. provide an extensive survey about Mining Software Repositories in [37], that builds upon the work of Kagdi et al. from 2007 [38]. Although software repositories can be mined for multiple purposes, one of our main interests is to use the bug repository to predict what parts or components of a software system are the weakest ones (defect prediction or likelihood of component failure). The idea here, is that components that have caused problems in the past (for which many bugs have been fixed) are more likely to cause problems in the future as well.

In this section, we will discuss relevant literature that focuses specifically on using bug repository mining techniques for defect prediction. In addition, some related approaches that use historical data for defect prediction are discussed.

### Bug Data Sets: Linking Bug Reports to Changesets

Since recent years, researchers have further investigated techniques based on the insight that information about the past of a component can be used to say something about the future of that component. The bug database contains useful historical data about components that can be gathered by mining the bug repository. As described by Nagappan et al. the key idea of this technique is “that one can map problems (in the bug database) to fixes (in the version database) and thus to those locations in the code that caused the problem” [2].

Zimmermann et al. have mapped the defects from the bug database of the Eclipse project to source code locations, and describe their work in the paper “Predicting Defects for Eclipse” [39]. In their paper, the authors explain how they computed the Eclipse bug data set and encourage others to use this data set to investigate research problems that they point out. They ask themselves the question “where bugs come from”, and claim that the
data set “offers the opportunity to research this question” because it can be used to evaluate models for defect prediction [39]. As an example, they use their data set to evaluate whether a combination of complexity metrics can be used to predict software defects.

The authors make use of traditional heuristics that “rely on the premise that developers leave hints or links about bug fixes in the change logs” [10]. To discover what parts of the source code have been changed in order to fix a bug, Zimmermann et al. link the bug reports to a changeset (or “commit”) by searching for references to bug reports in the commit messages. The heuristics that are traditionally used include the searching for keywords such as “Fixed” or “Bug” and bug IDs such as “#26510” or “Fixed 26510” [10]. These references are often added to the commit messages by the developers. Śliwerski et al. give a detailed description of the matching process they use in [17]. Their approach gives each (possible) link that is found two independent levels of confidence: (1) a syntactic level based on regular expressions and (2) a semantic level based on the information in the referenced bug report. Based on these confidence levels it is then decided whether a link is taken into account for further experiments [17].

Quality Concerns

The traditional approach of searching for references to bug reports in the commit messages has been applied extensively by other researchers as well, such as Čubranić et al. [40], Fischer et al. [41, 42], and Śliwerski et al. [17]. In [39], Zimmermann et al. do not name any percentages of the bug reports that they have been able to successfully link to fixes. But in the paper called “When Do Changes Induce Fixes?”, Śliwerski et al. describe that using the textual reference technique they were able to link 47% of the bug reports with a changeset (or transaction as they call it) for the Eclipse¹ project [17]. For the Mozilla² project, the authors managed to connect 55% of the bug reports to a changeset. Multiple studies that were conducted later used a similar approach and show consistent link-success rates of 40% to 60% for different projects: Bird et al. reach 46% for the Apache HTTP server³ [6], and Wu et al. are able to link 40.7% of the bugs for ZXing⁴ and 53.5% of the bugs for OpenIntents⁵ [10].

These numbers suggest that a large part of the links are not being discovered, which is confirmed by Bird et al. in [6]. Data sets such as the Eclipse bug data set are therefore a sample of the actual set of bug fixes and predictions made from samples can be wrong, especially “if the samples are not representative of the population” [6]. In their paper, Bird et al. found evidence of bug-feature bias in open source data sets and conclude that this bias probably affects the performance of defect prediction algorithms. This raises concerns

¹http://www.eclipse.org
²http://www.mozilla.org
³http://httpd.apache.org
⁴http://code.google.com/p/zxing
⁵http://www.openintents.org
8. Related Work

about the validity of studies that rely on bug and version control data, such as bug prediction models. Recently, many software prediction models have indeed been built and evaluated using historical defect data sets such as the Eclipse bug data set [14, 30, 12, 15]. In addition, Liebchen et al. already expressed their concerns regarding data sets in general in their 2008 paper called “Data Sets and Data Quality in Software Engineering” [43].

Another factor that influences the quality and usability of bug data sets is the renaming, splitting, and merging of components. Whenever for example the name (or filename) of a component gets changed, its history is lost. Therefore it would be feasible to apply mechanisms that take these factors into account during the mining process, such that the old and new version of a component are correctly linked. The process of *origin analysis* is addressed by several researchers, for example Godfrey et al. [44], Fluri et al. [45], and Weissgerber et al. [46].

**Link Recovery**

In a response to the findings by Bird et al. some researchers focused on investigating and removing the *noises* from the data sets. Kim et al. measure “the impact of noise on defect prediction models” and “provide guidelines for acceptable noise level” in [47]. They conclude that in general, the performance of defect prediction models are not affected significantly when the data set that is used to evaluate the performance is noisy. However, a noise level of 20% to 35% (of both false negatives and false positives) does cause the prediction performance to decrease significantly [47].

When a link between a bug report and a changeset is missing in a dataset, this doesn’t necessarily mean that the link does not exist or cannot be found at all. In many cases, the used algorithm is just not capable enough to discover the link while it can be retrieved by manual inspection. Bachmann et al. present their tool called *Linkster* in [7] that facilitates “link reverse-engineering”. The tool allows for manual addition of missing links to a data set and therefore improving the quality of the set for research purposes such as the evaluation of bug prediction models. To successfully discover additional links, however, the manual inspection often requires a domain expert that knows the code well and is a very time consuming operation. The authors asked an expert of the Apache HTTP server to analyze 493 commits and he was able to add 65 additional linked bugs to the data set that already contained 1576 bugs. During their research, the authors also found reasons to doubt some of the core assumptions made about a bug data set because (1) “bugs often go incognito” (missing bugreport for a fixed bug) and (2) “commits not always clearly change the functionality of the program” [7]. While missing links might still be retrieved at a later moment in time using alternative link recovery methods, bugs that go incognito are much harder to discover especially using automated approaches.

In contrast to the work of Bachmann et al. (which requires manual inspection), Wu et al. developed an automated link recovery algorithm. Their algorithm called “ReLink” is described in [10]. Their approach is based on the observation that “the links exhibit certain
The algorithm automatically learns the criteria of features from explicit links to discover additional links. Evaluation shows that ReLink is able to discover more links when compared to an approach that uses traditional heuristics, but also introduces false positives.

The large amount of missing links suggest that the developers that commit the changes are not very consistent in adding a reference to bug reports in their commit messages. The main reason for this is because developers are not required to add a reference to a bug report when committing a change, and not required to add a reference to a changeset or commit when closing a bug report (by setting the state of the bug report to “released” or “fixed”). Therefore, these linkages become irregular and inconsistent [6].

The bug tracking system and source version repositories that are used in the research mentioned before to create the bug data sets are generally two separate systems. A tighter integration of these systems would be desirable to support the linking process and allow for better traceability. Systems that do have both a bug tracking and version control system integrated exist, but are unfortunately mostly used in closed-source, commercial environments. These environments are generally less suitable for research purposes because others cannot reproduce, verify or refute the results. Examples of these systems include Microsoft Team Foundation Server[^6] (TFS) and IBM Jazz[^7]. Both support the explicit linking of bug reports and changesets, and even require it in certain cases. Herzig et al. provide an early evaluation of mining opportunities with Jazz in [48] and Nguyen et al. present an experience report in [49]. In “Mining Software Metrics from Jazz” [50] Finlay et al. describe their attempt to predict build success and/or failure for a software product. They mine the source code contained in Jazz for this purpose and provide a detailed description of their approach. In this study, we report on our findings of repository mining using Microsoft TFS.

### 8.1.2 Static Software Metrics

In 1994, Chidamber and Kemerer published their paper called “A Metrics Suite for Object Oriented Design” in which they respond to the demand for software metrics (or measures). They propose and evaluate the following six now well-known design metrics which according to them can be used by managers for process improvement [51]:

**WMC (Weighted Methods per Class):** The sum of the complexities of the methods in a class. The complexity of a method is deliberately left undefined and left as an implementation decision. The number of methods per class in combination with the complexity of the methods predicts how much time and effort is required to develop and maintain a class. Classes with a high number of methods are also likely to be more application specific.

**DIT (Depth of Inheritance Tree):** The depth of inheritance of a class defined by the (maximum) length from the node to the root of the tree. The deeper a class is in the hier-
arch, the complexer it is to predict its behavior and the greater the potential of reuse of inherited methods. Also, deeper trees imply greater design complexity.

**NOC (Number Of Children):** The number of immediate subclasses of a class. A greater number implies greater reuse and likelihood of improper abstraction, and the number of children gives an idea of the potential influence of a class on the design.

**CBO (Coupling Between Object classes):** The number of classes with which the class is coupled through any relation. High coupling between classes prevents reuse and shows a lack of modular design. A high number of couples makes the class sensitive to changes in other parts of the design, reducing maintainability and testability.

**RFC (Response For a Class):** The number of methods that may be executed when a method of a class is called. Testing and debugging becomes more complicated when a large number of other methods are being called in response to a method call.

**LCOM (Lack of Cohesion in Methods):** The number of methods with “pairwise disjoint sets of instance variables referenced within their respective method bodies” [52]. Cohesiveness of the methods of a class promotes encapsulation, and a lack thereof suggests that a class needs splitting.

The proposed metrics of Chidamber et al. say something about software quality in general, but do not focus on defect prediction. In continuation of this paper in 2005, Gyimothy et al. further investigated the proposed metrics in relation to defect prediction in particular. They added two metrics: (1) the LCOMN (Lack of Cohesion in Methods allowing Negative value) metric and (2) the well-known LOC (Lines Of Code) metric. They conclude that the CBO metric “seems to be the best in predicting the fault-proneness of classes” [53], the LOC metric performed fairly good as well, the LCOM and DIT do not perform very well, and finally the NOC metric is not usable at all for fault-proneness prediction.

However some researchers report different findings than Gyimothy et al. For example Fenton et al. concluded in [54] that complexity (as measured by “complexity metrics”) does not explain fault-prone behavior, and complexity “is not significantly better at predicting fault- and failure-prone modules than simple size measures” [54]. In response to the difference in results, D’Ambros et al. conclude that “what is sorely missing is a baseline against which the approaches can be compared”. They provide such a baseline in the form of an extensive dataset that is composed of multiple open-source systems, as described in [36], and claim that it is possible to evaluate several approaches using their dataset, because it contains the right information for this purpose.

The results on the performance of static code metrics for defect prediction varies quite a lot, often due to a number of problems in many studies. Fenton et al. give a critique of software defect prediction models in [55]. An interesting insight into the usability of metrics for defect prediction is given by Menzies et al. They conclude the following: “we endorse the use of static code attributes for predicting defects with the following caveat:
Defect Prediction

Those predictors should be treated as probabilistic, not categorical, indicators. While our best methods have a nonzero false alarm, they also have a usefully high probability of detection (over two-thirds). Just as long as users treat these predictors as indicators and not definite oracles, then the predictors learned here would be pragmatically useful for focusing limited verification and validation budgets on portions of the code base that are predicted to be problematic.” [56].

The definition of code complexity metrics as used in multiple papers differs (often only slightly). Most include the metrics proposed by Chidamber and Kemerer (CK metrics) partly. For example in [57], Nagappan et al. determine the complexity of code based on the following complexity metrics:

**Cyclomatic Complexity:** The number of linearly-independent paths through a program module such as a function [58].

**Fan-In:** The number of functions calling a function.

**Fan-Out:** The number of functions called by a function.

**Lines of Code:** The LOC as defined by Chidamber and Kemerer.

**Weighted Methods per Class:** WMC as defined by Chidamber and Kemerer.

**Depth of Inheritance:** DIT as defined by Chidamber and Kemerer.

**Coupling Between Objects:** CBO as defined by Chidamber and Kemerer.

**Number of Sub classes:** NOC as defined by Chidamber and Kemerer.

**Total Global Variables** The total number of global variables.

As reported in [27] Nagappan et al. investigated the use of static analysis tools to predict pre-release defect density. In short, they conclude that static analysis defect density can indeed be used to “predict pre-release defect density at statistically significant levels”, and can be used to “discriminate between components of high and low quality” [27]. The static analysis defect density of a component is defined as the number of defects per KLOC (thousand lines of code) found by static analysis tools.

8.1.3 Defect Predictors: Combining Historical Data and Static Metrics

There are many researchers who propose approaches that focus on combining both historical data or MSR techniques with software quality metrics, in order to take advantage of the strengths of both techniques. Information retrieved from historical data such as bug repositories is about actual problems that components caused, where information retrieved from static metrics provides insight into the current state and quality of components.
In “Emerald: software metrics and models on the desktop” [59] published in 1996, Hudepohl et al. show that they successfully predicted whether a module would be defect-prone using the combination of historical data with software metrics.

Nagappan et al. were among the first to include bug repository mining techniques in 2006, and present an approach to create failure predictors as described in “Mining Metrics to Predict Component Failures” [2]. They first mine the software repository archives to map failures to entities and then compute a wide range of “standard complexity metrics” for these entities. A failure predictor that determines the failure probability of components is then created based on the results of the computed complexity metrics.

The authors explain that previous work has used object-oriented metrics to predict pre-release defect density of components, while their approach predicts post-release defects and therefore actual failures. An important conclusion that the authors draw is that for each verified project they are able to find a set of complexity metrics that correlates with post-release defects, but unable to select a single set of metrics that fits all projects. The predictors can however be reused for the same or similar projects. Therefore they advice not to use complexity metrics without validating them for your project, but metrics that are validated from history can indeed be used to identify low-quality components [2].

Arisholm et al. conducted research similar to Nagappan et al. in 2006 as well, as presented in [13]. They constructed and validated fault-proneness prediction models in the context of an object-oriented evolving legacy system. In their research, they also include a cost-effectiveness analysis to demonstrate the usability of such predictors. The most important conclusion is as follows: “our study shows that building such fault-proneness models is promising as it could potentially save verification effort in the context of a constantly changing legacy system. It also suggests that using history change and fault data about previous releases is paramount to developing a useful prediction model on a given release.” [13].

Kim et al. describe a slightly different approach in [8]. They try to predict the most fault prone entities and files using the version history of 7 software systems, under the assumption that faults do not occur in isolation but in bursts of several related faults. Their approach uses a cache history that contains locations that are likely to have faults. “By consulting the cache at the moment a fault is fixed, a developer can detect likely fault-prone locations.” [8]. In their paper, the authors describe and evaluate two algorithms for maintaining this cache: BugCache and FixCache. When a fault has occurred, the FixCache algorithm is able to predict further faults with high accuracy at file level (73%-95%). A noteworthy aspect of their approach is the fact that because fault occurrences directly affect the cache, FixCache is able to adapt more quickly than static models.
8.2 Software Testability

8.2.1 Defining Testability

According to the ISO/IEC 9126 standard, testability is a quality sub-characteristic of the maintainability quality dimension [52]. However, the testability of software is now considered to be a distinct software quality characteristic and has been well researched by many researchers over the past decades. They define testability in often different ways, matching their particular application area (such as embedded systems) [60, 61, 62]. More formal definitions are given by IEEE and ISO. Testability is defined by IEEE in 1990 as “the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met” [63], and by ISO in 1991 as “attributes of software that bear on the effort needed to validate the software product” [19].

Sneed et al. state that the costs of testing “are driven by the size and complexity of the software”, with size being a rather broad definition of “the number of elements making up the system” and complexity “the number of interactions between the elements” [22].

8.2.2 Metrics for Testability

According to Kout et al. metrics can be used for predicting the testability of components and to better manage the testing effort [64]. Many authors have researched the use of metrics to determine the testability of components. Feedman introduces the concept of domain testability and proposes test metrics that can be used to assess how much effort is required to transform a program in such a way that it becomes domain testable [60]. In 1995, Voas et al. published their paper “The new verification” in which they describe their approach which uses semantic information of specification and design documents [62].

But more recent work can also be found on the topic of metrics for testability. For example, Bruntink et al. discuss (class) testability into detail and provide an evaluation of well-known object-oriented metrics for predicting class testability [65, 66, 23]. Badri et al. report on their findings after investigating the relationship between object-oriented design metrics and testability of classes as well in [67, 18].
8. RELATED WORK

8.3 Reengineering for Testability

Although software testing is an important activity in all software development projects and organizations, many software systems have still not been built with maintainability as requirement today. This leads to the many cases of legacy systems for which maintaining and developing new features becomes increasingly harder as the system grows in size and complexity. Since maintainability and testability are two closely related fields (see [23]), these legacy systems generally also lack coverage with automated tests as well. It is often the case that before these systems can be covered in tests, thorough reengineering is required first.

8.3.1 Achieving Testability in Legacy Systems

Reengineering itself is a well known practise and widely applied by many companies in order to get their systems maintainable again. These reengineering efforts are therefore mainly focused on maintainability. But in our research we are interested in reengineering for a related though different goal: testability. An example of the difference with traditional transformations is that “testability transformation does not preserve the traditional semantics, so much as it preserves a new form of semantics defined by the test adequacy criterion” [68].

The reason for focusing reengineering effort on testability rather than maintainability, is that in this study, components that have been causing a lot of problems are being reengineered. In this case, design decisions in favor of better testability are preferred over a better maintainability since quality assurance is of higher importance for these components.

Although many articles that are related to reengineering and testability have been published, not much work can be found on the combination of the two. For a “Workshop on Software Reengineering (WSR)” the paper of Sneed called “Reengineering for Testability” [22] focuses specifically on reengineering techniques to improve the testability of a system. Harman reviews the theory of testability transformation in [68] and outlines its implications for testing. Hegedus et al. discuss the importance of consequences of refatorings in [69].

Many guides can be found on reengineering and refactorings in general (i.e. Feathers [1]), but Sneed provides 10 measures specifically for improving testability of a system. These measures can be used as a guide to perform testability transformation:

1. Restructuring to eliminate unnecessary paths
2. Refactoring to reduce complexity
3. Removing clones
4. Removing redundant parameters
5. Grouping database accesses into an access shell
6. Merging database tables

7. Eliminating unnecessary import/export interfaces

8. Simplifying user interfaces

9. Revising the Algorithms

10. Restructuring the Architecture

Using metrics such as those discussed in section 8.2.2, it is possible to measure the improvements in terms of testability after performing such transformations.
Chapter 9

Conclusions and Future Work

In this study we have investigated how to select bug-prone components in order to study the effectiveness of reengineering and unit testing on the number of bugs. Section 9.1 provides an overview of the contributions that we made. In Section 9.2 we present our final conclusions and Section 9.3 discusses directions for future work.

9.1 Contributions

In this study we made the following contributions:

C1 Provide a procedure to study the effectiveness of unit testing and reengineering bug-prone components.

C2 Provide a bug repository mining approach to TFS in an industrial setting, in order to create bug sets that can be used for prediction and evaluation purposes.

C3 Present and evaluate different predictors that predict the bug-proneness of components.

C4 Provide a selection procedure that ranks components by a combination of high bug-proneness and low estimated unit test effort.

C5 Identifying the biggest obstacles for testability in the Exact Online codebase, and provide a step-by-step guide to overcome them.

C6 Reengineering a bug-prone component to increase its testability and maintainability and make it more resilient to bug-introducing changes in an industry case study.

C7 Evaluate the transition from mockist to classical unit testing by a survey in an industry case study.
9.2 Research Questions Revisited

In Chapters 2 to 6 we concluded the following:

**RQ1** How can the bug repository (TFS) be used to extract information about the most bug-prone components?

By selecting the correct changeset, we can link bug fixes to source locations. This information (stored in bug data sets) can be used by predictors to give each component a bug score, and rank them accordingly. The bug score indicates the predicted bug-proneness of each component. The predictions of all evaluated predictors show a correlation with the actual bugs that have been fixed within 5 months after prediction. The predictor used in this study (Psla) shows a Pearson correlation of 0.275 and can thus be used to predict component bug-proneness, but it does not outperform the Pbugs predictor (0.368 Pearson correlation).

The success of bug repository mining for prediction and evaluation depends on a number of factors, including the correctness of the administrative tasks. We conclude that these factors at EOL development look promising for our study (e.g. low chance of bias in the results), and the information inside the TFS bug repository is of such quality that it can be used for development decisions as well.

**RQ2** How can bug-prone components be selected for which invested reengineering and unit testing effort is spent most efficiently?

Such components can be selected by including the required test effort: an estimation of the effort that is required to reengineer and write unit tests for a component. By calculating the bug score per test effort (called feasibility), components with the combination of a high bug score and low test effort rank highest.

Unfortunately, 80% of the components have been filtered out as they contain too much data related code which cannot be unit tested. Eventually the procedure results in a list of components, ranked by feasibility. Of this list, we have selected the PriceTools and LogisticsTools components as targets to be reengineered and unit tested.

The results of the validation of the actual feasibility show that in general it is better to spend reengineering and unit test effort on the components that require less test effort, as more bugs can be prevented per invested time for these components.
**RQ3** What are the obstacles for testability of the Exact Online codebase and how can these testability issues be resolved?

The two main reasons for untestable code are:

1. **Database Access**
2. **Dependencies on Environment**

To resolve the first obstacle, we propose to move the database calls to a database access shell called *dataprovder*, and injecting an interface of such a dataprovider into the code under test.

The second testability obstacle, the dependencies on Environment, occurs in three ways: method invocation on Environment, message chaining member objects of Environment and passing the Environment on. To remove the method invocation dependency, an interface of Environment instead of the Environment itself could be used. The message chaining problem can be resolved by injecting (an interface of) the required member object instead of the Environment. The dependency where the Environment is being passed on can be removed by moving the untestable code to a new (specialized) object, and injecting an interface of the newly created object.

**RQ4** What issues can be identified for the selected components and how can these be resolved?

The four most important issues that cause the PriceTools component to be bug-prone are:

1. Complex functions
2. No clear behavior of the classes and functions
3. Duplicated code
4. Confusing interface

These issues have been resolved by the combination of a number of refactorings, of which the most important are:

- splitting the complex functions into small, clear functions.
9. CONCLUSIONS AND FUTURE WORK

- making decisions about corner cases and documenting them in comments
- removing duplicated code by introducing a new function
- replacing the “Object” types in parameters with the correct types
- improving the readability by lowering the depth of nestings and adding small comments

In addition, the unit test harness that is being executed on a daily basis prevents the existing behavior from changing unexpectedly, which makes it much more resilient to bug-introducing changes.

| RQ5 Do the developers understand that applying the classical unit testing approach is reducing the legacy problem? |

They partially do. The developers seem to understand the advantages of testable code to the maintainability of it and its use as safety net, but at the same time are afraid to break existing functionality during refactorings and do not seem to believe that the refactoring effort is worth the investment. The developers do not consider themselves to be very experienced in unit testing and do not write unit tests often. In addition, the developers from the functional teams do not really enjoy writing unit tests (either with or without a mocking framework). The combination of lack of experience and not really enjoying writing unit tests could be the reason why they do not believe that refactoring existing code is worth the effort. However for new code the developers do state that they are likely to write testable code, meaning a slowdown in the growth of legacy code.

| Main Research Question Does the reengineering and unit testing of a bug-prone component result in a measurable difference in reported bugs for that component? |

Initial results provide a first indication that by reengineering and unit testing a bug-prone component, the number of reported bugs go down. Based on the 40 bugs that have been fixed for the 19 other components of the top 20 feasible components, an estimated 0.842 bugs would have been fixed for the PriceTools component if it would have not been reengineered and unit tested. But only 63.1% of the other components required at least 1 bug to be fixed in the evaluation period, and the number of estimated bugs that would have needed to be fixed is very low. The chance that an actual bug would have been fixed for the PriceTools component is therefore very low. By reengineering and unit testing a larger group of components, and measuring over a longer period of time, a more precise answer of higher certainty can be formulated.
9.3 Future work

More components and longer measurement period In order to formulate a final answer on our main research question, our procedure has to be repeated for a larger number of components, and measured over a longer period of time.

Bug set confidence We propose further investigation of the influence of incorrect information inside bug data sets on the conclusions that are being drawn from them. Bird et al. showed that bias is likely to exist in research that uses bug sets [6], but it is very difficult to determine how much. By having a procedure that includes the characteristics of the development and bug administration procedures, and factors such as the stability of the code, we might be able to calculate a level of confidence of how correct the bug sets and results drawn from them most likely are.

Optimal bug-proneness predictor configuration The Psla bug predictor shows a Pearson correlation of 0.368 between predicted and actual bugs, which is the highest correlation between predicted and actual bugs of all our evaluated predictors. By tweaking the configuration of such predictors we would like to find an optimum that would result in an even better predictor. In addition, we would like to evaluate other factors such as number of changes, that might further improve the predictor of component bug-proneness.

Test effort estimation Our estimations of the required reengineering and unit test effort are made on class-level. We believe these estimations are a nice first step, but could improve by estimating the required effort per function, and adding them to determine the effort required for an entire component.
Bibliography


## Appendix A

### Top 100 Bug-prone Components

<table>
<thead>
<tr>
<th>Rank</th>
<th>Bugscore</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4200</td>
<td>Exact.Contracts.Core/ContractTools.vb</td>
</tr>
<tr>
<td>2</td>
<td>3500</td>
<td>Exact.VAT/VATGLTransactionsEntryFC.vb</td>
</tr>
<tr>
<td>3</td>
<td>3150</td>
<td>Exact.Sales.Orders/SalesOrderLineFC.vb</td>
</tr>
<tr>
<td>4</td>
<td>3150</td>
<td>Exact.VAT.Core/VATTools.vb</td>
</tr>
<tr>
<td>5</td>
<td>3100</td>
<td>Exact.Manufacturing/ShopOrderCopy.vb</td>
</tr>
<tr>
<td>6</td>
<td>2550</td>
<td>Exact.CashFlow.NL/MT940.vb</td>
</tr>
<tr>
<td>7</td>
<td>2550</td>
<td>Exact.Logistics.Sales/SalesInvoiceLineFC.vb</td>
</tr>
<tr>
<td>8</td>
<td>2550</td>
<td>Exact.Purchase.Invoice/PurchaseInvoiceLineFC.vb</td>
</tr>
<tr>
<td>9</td>
<td>2550</td>
<td>Exact.Financial/FixedEntry.vb</td>
</tr>
<tr>
<td>10</td>
<td>2550</td>
<td>Exact.Cashflow/CashflowPlanning.vb</td>
</tr>
<tr>
<td>11</td>
<td>2500</td>
<td>Exact.Logistics/BaseEntryFC.vb</td>
</tr>
<tr>
<td>12</td>
<td>2200</td>
<td>Exact.Layouts/LayoutDataSalesInvoice.vb</td>
</tr>
<tr>
<td>13</td>
<td>2150</td>
<td>Exact.Quotation/SalesQuotationLineFC.vb</td>
</tr>
<tr>
<td>14</td>
<td>2150</td>
<td>Exact.Inventory/StockTransactionLineFC.vb</td>
</tr>
<tr>
<td>15</td>
<td>2150</td>
<td>Exact.Manufacturing/ShopOrderFC.vb</td>
</tr>
<tr>
<td>16</td>
<td>2050</td>
<td>Exact.Purchase.Orders/PurchaseOrderLineFC.vb</td>
</tr>
<tr>
<td>17</td>
<td>2050</td>
<td>Exact.Logistics.Sales/FixedSalesInvoiceLineFC.vb</td>
</tr>
<tr>
<td>18</td>
<td>2050</td>
<td>Exact.XML.Financial/GLTransaction.vb</td>
</tr>
<tr>
<td>19</td>
<td>2050</td>
<td>Exact.VAT.Cache/Queries.vb</td>
</tr>
<tr>
<td>20</td>
<td>2050</td>
<td>Exact.Logistics/ItemMaterialFC.vb</td>
</tr>
<tr>
<td>21</td>
<td>2000</td>
<td>Exact.Logistics.Core/LogisticsTools.vb</td>
</tr>
<tr>
<td>22</td>
<td>2000</td>
<td>Exact.XML.Financial/VATCode.vb</td>
</tr>
<tr>
<td>23</td>
<td>2000</td>
<td>Exact.Logistics/ItemVersionFC.vb</td>
</tr>
<tr>
<td>24</td>
<td>1700</td>
<td>Exact.XML.Financial/Queries.vb</td>
</tr>
<tr>
<td>25</td>
<td>1650</td>
<td>Exact.Quotation/SalesQuotationEntry.vb</td>
</tr>
</tbody>
</table>
### A. Top 100 Bug-prone Components

<table>
<thead>
<tr>
<th>Rank</th>
<th>Bugscore</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>1550</td>
<td>Exact.MailBox.MailMessageAttachmentsFC.vb</td>
</tr>
<tr>
<td>27</td>
<td>1550</td>
<td>Exact.EInvoice.UBL.UBL.vb</td>
</tr>
<tr>
<td>28</td>
<td>1550</td>
<td>Exact.Logistics.Sales.SalesInvoiceLineBCEExtra.vb</td>
</tr>
<tr>
<td>29</td>
<td>1500</td>
<td>Exact.Sales.Orders.SOConsistency.vb</td>
</tr>
<tr>
<td>30</td>
<td>1500</td>
<td>Exact.Web.Api.Read.HostingDataSource.vb</td>
</tr>
<tr>
<td>31</td>
<td>1500</td>
<td>Exact.EOLConvert.CFD.CFDFunctions.vb</td>
</tr>
<tr>
<td>32</td>
<td>1500</td>
<td>Exact.EOLConvert.CFD.CFDGLTransactions.vb</td>
</tr>
<tr>
<td>33</td>
<td>1500</td>
<td>Exact.EOLConvert.CFD.CFDMasterData.vb</td>
</tr>
<tr>
<td>34</td>
<td>1500</td>
<td>Exact.Cashflow.MatchFinder.vb</td>
</tr>
<tr>
<td>35</td>
<td>1500</td>
<td>Exact.Core.CustomerLicense.vb</td>
</tr>
<tr>
<td>36</td>
<td>1500</td>
<td>Exact.Financial.UI.Controls.VATPercentagesComboBox.vb</td>
</tr>
<tr>
<td>37</td>
<td>1100</td>
<td>Exact.VAT.Return.VATReturn.vb</td>
</tr>
<tr>
<td>38</td>
<td>1100</td>
<td>Exact.Payroll.BCEmploymentContract.vb</td>
</tr>
<tr>
<td>40</td>
<td>1100</td>
<td>Exact.Contracts.ContractLineFC.vb</td>
</tr>
<tr>
<td>41</td>
<td>1100</td>
<td>Exact.Logistics.Core.ItemTools.vb</td>
</tr>
<tr>
<td>42</td>
<td>1100</td>
<td>Exact.MailBox.Core.MailBoxTools.vb</td>
</tr>
<tr>
<td>43</td>
<td>1100</td>
<td>Exact.Cashflow.NL.Rabobank.vb</td>
</tr>
<tr>
<td>44</td>
<td>1100</td>
<td>Exact.Logistics.Core.PriceTools.vb</td>
</tr>
<tr>
<td>45</td>
<td>1100</td>
<td>Exact.Manufacturing.Core.ShopOrderEntryTools.vb</td>
</tr>
<tr>
<td>46</td>
<td>1050</td>
<td>Exact.EOLConvert.EvD.DOSGLTransactions.vb</td>
</tr>
<tr>
<td>47</td>
<td>1050</td>
<td>Exact.MailBox.Jobs.LinkBankAccount.vb</td>
</tr>
<tr>
<td>48</td>
<td>1050</td>
<td>Exact.Logistics.ItemFC.vb</td>
</tr>
<tr>
<td>49</td>
<td>1050</td>
<td>Exact.Declarations.VatDeclaration.vb</td>
</tr>
<tr>
<td>50</td>
<td>1050</td>
<td>Exact.Logistics.Sales.FixedSalesInvoiceLineBCEExtra.vb</td>
</tr>
<tr>
<td>51</td>
<td>1050</td>
<td>Exact.XML.Manufacturing.TimeTransactions.vb</td>
</tr>
<tr>
<td>52</td>
<td>1050</td>
<td>Exact.XML/BaseNodeProvider.vb</td>
</tr>
<tr>
<td>53</td>
<td>1050</td>
<td>Exact.Topics.Core.Enums.vb</td>
</tr>
<tr>
<td>54</td>
<td>1050</td>
<td>Exact.Topics.Core.TopicTools.vb</td>
</tr>
<tr>
<td>55</td>
<td>1050</td>
<td>Exact.Quotation/QuotationManufacturingLineFC.vb</td>
</tr>
<tr>
<td>56</td>
<td>1050</td>
<td>Exact.Quotation/QuotationQuantityFC.vb</td>
</tr>
<tr>
<td>57</td>
<td>1050</td>
<td>Exact.Quotation/QuotationToShopOrderCopy.vb</td>
</tr>
<tr>
<td>58</td>
<td>1050</td>
<td>Exact.Manufacturing.ShopOrderMaterialFC.vb</td>
</tr>
<tr>
<td>59</td>
<td>1000</td>
<td>Exact.Financial.BE/GLAccounts.vb</td>
</tr>
<tr>
<td>60</td>
<td>1000</td>
<td>Exact.Payroll.BCPayrollYear.vb</td>
</tr>
<tr>
<td>61</td>
<td>1000</td>
<td>Exact.EOLConvert.CFD.CFDAutoConvert.vb</td>
</tr>
<tr>
<td>62</td>
<td>1000</td>
<td>Exact.EOLConvert.CFD.CFDFile.vb</td>
</tr>
<tr>
<td>63</td>
<td>1000</td>
<td>Exact.EOLConvert.CFD.CFDHandler.vb</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>Exact.EOLConvert.CFD.CFDSQLInsert.vb</td>
</tr>
<tr>
<td>65</td>
<td>1000</td>
<td>Exact.EOLConvert.CFD/DBFMemFile.vb</td>
</tr>
<tr>
<td>Rank</td>
<td>Bugscore</td>
<td>File</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>66</td>
<td>1000</td>
<td>Exact.VAT.BE/VATReturn.BE.vb</td>
</tr>
<tr>
<td>67</td>
<td>1000</td>
<td>Exact.Cashflow.Link/Tools.vb</td>
</tr>
<tr>
<td>68</td>
<td>1000</td>
<td>Exact.Financial/FinEntryClientWriter.vb</td>
</tr>
<tr>
<td>69</td>
<td>1000</td>
<td>Exact.Financial/FinancialTransactionFC.vb</td>
</tr>
<tr>
<td>70</td>
<td>1000</td>
<td>Exact.XML.Financial/Journal.vb</td>
</tr>
<tr>
<td>71</td>
<td>1000</td>
<td>Exact.XML.Financial/Journals.vb</td>
</tr>
<tr>
<td>72</td>
<td>1000</td>
<td>Exact.Layouts.Processor/LayoutProcessor.vb</td>
</tr>
<tr>
<td>73</td>
<td>1000</td>
<td>Exact.Purchase.Invoice/PurchaseInvoiceLineBCEXtra.vb</td>
</tr>
<tr>
<td>74</td>
<td>1000</td>
<td>Exact.Quotation/SalesQuotationLineBCEXtra.vb</td>
</tr>
<tr>
<td>75</td>
<td>1000</td>
<td>Exact.Sales.Orders/CreateInvoice.vb</td>
</tr>
<tr>
<td>76</td>
<td>1000</td>
<td>Exact.EOLConvert.EvD/DOSMasterData.vb</td>
</tr>
<tr>
<td>77</td>
<td>1000</td>
<td>Exact.Cashflow/CashflowEntryOffsetLineTransform.vb</td>
</tr>
<tr>
<td>78</td>
<td>1000</td>
<td>Exact.Topics.Broadcast/BroadcastComponent.vb</td>
</tr>
<tr>
<td>79</td>
<td>1000</td>
<td>Exact.Topics.Core/TopicLinker.vb</td>
</tr>
<tr>
<td>80</td>
<td>1000</td>
<td>Exact.Topics.Jobs/BroadcastPushBg.vb</td>
</tr>
<tr>
<td>81</td>
<td>1000</td>
<td>Exact.Quotation.Core/QuotationDetailTools.vb</td>
</tr>
<tr>
<td>82</td>
<td>1000</td>
<td>Exact.Logistics/ItemRoutingStepFC.vb</td>
</tr>
<tr>
<td>83</td>
<td>1000</td>
<td>Exact.Manufacturing.Jobs/MaterialBackflushTask.vb</td>
</tr>
<tr>
<td>84</td>
<td>900</td>
<td>Exact.Payroll.Returns/TaxReturn.vb</td>
</tr>
<tr>
<td>85</td>
<td>650</td>
<td>Exact.CRM.Apps/CRMAccountCardData.vb</td>
</tr>
<tr>
<td>86</td>
<td>600</td>
<td>Exact.B2B/RepairCrmActivationAction.vb</td>
</tr>
<tr>
<td>87</td>
<td>600</td>
<td>Exact.Layouts/LayoutDataQuotation.vb</td>
</tr>
<tr>
<td>88</td>
<td>600</td>
<td>Exact.Sales.Core/SalesOrderStatusTools.vb</td>
</tr>
<tr>
<td>89</td>
<td>600</td>
<td>Exact.Quotation.Core/QuotationSettings.vb</td>
</tr>
<tr>
<td>90</td>
<td>600</td>
<td>Exact.Sales.Invoices/SlsInvPlnInvoiceCreator.vb</td>
</tr>
<tr>
<td>91</td>
<td>550</td>
<td>Exact.VAT/VATCodeFC.vb</td>
</tr>
<tr>
<td>92</td>
<td>550</td>
<td>Exact.Inventory/StockCountLineFC.vb</td>
</tr>
<tr>
<td>93</td>
<td>550</td>
<td>Exact.Logistics.Sales/SalesCopy.vb</td>
</tr>
<tr>
<td>94</td>
<td>550</td>
<td>Exact.Logistics.Sales/SalesInvoiceEntry.vb</td>
</tr>
<tr>
<td>95</td>
<td>550</td>
<td>Exact.Jobs.NL/BAPI_OnDemand.vb</td>
</tr>
<tr>
<td>96</td>
<td>550</td>
<td>Exact.Core/Environment/Environment.vb</td>
</tr>
<tr>
<td>97</td>
<td>550</td>
<td>Exact.Layouts/LayoutDataSubscription.vb</td>
</tr>
<tr>
<td>98</td>
<td>550</td>
<td>Exact.Documents.Core/DocTools.vb</td>
</tr>
<tr>
<td>99</td>
<td>550</td>
<td>Exact.Licensing/LicenseUpgradeFC.vb</td>
</tr>
<tr>
<td>100</td>
<td>500</td>
<td>Exact.Purchase.Orders/PurchaseOrderConsistency.vb</td>
</tr>
</tbody>
</table>
## Appendix B

### Top 20 Bug-prone non Data Related Components

<table>
<thead>
<tr>
<th>Rank</th>
<th>Bugscore</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4200</td>
<td>Exact.Contracts.Core/ContractTools.vb</td>
</tr>
<tr>
<td>2</td>
<td>3150</td>
<td>Exact.VAT.Core/VATTools.vb</td>
</tr>
<tr>
<td>3</td>
<td>2550</td>
<td>Exact.CashFlow.NL/MT940.vb</td>
</tr>
<tr>
<td>4</td>
<td>2550</td>
<td>Exact.Cashflow/CashflowPlanning.vb</td>
</tr>
<tr>
<td>5</td>
<td>2000</td>
<td>Exact.Logistics.Core/LogisticsTools.vb</td>
</tr>
<tr>
<td>6</td>
<td>1500</td>
<td>Exact.Cashflow/MatchFinder.vb</td>
</tr>
<tr>
<td>7</td>
<td>1500</td>
<td>Exact.Financial.UI.Controls/VATPercentagesComboBox.vb</td>
</tr>
<tr>
<td>8</td>
<td>1100</td>
<td>Exact.VAT.Return/VATReturn.vb</td>
</tr>
<tr>
<td>9</td>
<td>1100</td>
<td>Exact.Logistics.Core/ItemTools.vb</td>
</tr>
<tr>
<td>10</td>
<td>1100</td>
<td>Exact.MailBox.Core/MailBoxTools.vb</td>
</tr>
<tr>
<td>11</td>
<td>1100</td>
<td>Exact.Logistics.Core/PriceTools.vb</td>
</tr>
<tr>
<td>12</td>
<td>1050</td>
<td>Exact.MailBox.Jobs/LinkBankAccount.vb</td>
</tr>
<tr>
<td>13</td>
<td>1050</td>
<td>Exact.XML/BaseNodeProvider.vb</td>
</tr>
<tr>
<td>14</td>
<td>1050</td>
<td>Exact.Topics.Core/TopicTools.vb</td>
</tr>
<tr>
<td>15</td>
<td>1000</td>
<td>Exact.Financial.BE/GLAccounts.vb</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>Exact.Cashflow.Link/Tools.vb</td>
</tr>
<tr>
<td>17</td>
<td>1000</td>
<td>Exact.Sales.Orders/CreateInvoice.vb</td>
</tr>
<tr>
<td>18</td>
<td>1000</td>
<td>Exact.Quotation.Core/QuotationDetailTools.vb</td>
</tr>
<tr>
<td>19</td>
<td>550</td>
<td>Exact.Jobs.NL/BAPI_OnDemand.vb</td>
</tr>
<tr>
<td>20</td>
<td>550</td>
<td>Exact.Core/Environment/Environment.vb</td>
</tr>
</tbody>
</table>
Appendix C

Top 20 Feasible Components

<table>
<thead>
<tr>
<th>Component File</th>
<th>Remarks</th>
<th>Bug Score</th>
<th>Bugs</th>
<th>CC Functions</th>
<th>Avg. CC per Function</th>
<th>Bugginess</th>
<th>Test Effort</th>
<th>Feasibility Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact. Financial.UK Controls/ VAT Percentages Combo\Box.vb</td>
<td>UI Control</td>
<td>1500</td>
<td>3</td>
<td>33</td>
<td>9</td>
<td>3.67</td>
<td>1500.00</td>
<td>321.00</td>
</tr>
<tr>
<td>Exact. Quotation.Core/ Quotation Detail\Tools.vb</td>
<td>Tools</td>
<td>1000</td>
<td>2</td>
<td>74</td>
<td>51</td>
<td>1.45</td>
<td>1000.00</td>
<td>107.37</td>
</tr>
<tr>
<td>Exact. Logistics.Core/ Price\Tools.vb</td>
<td>Tools</td>
<td>1100</td>
<td>3</td>
<td>69</td>
<td>32</td>
<td>2.16</td>
<td>1100.00</td>
<td>148.78</td>
</tr>
<tr>
<td>Exact. XML/Base\Node Provider.vb</td>
<td>XML data-related</td>
<td>1050</td>
<td>2</td>
<td>107</td>
<td>31</td>
<td>3.45</td>
<td>1050.00</td>
<td>369.32</td>
</tr>
<tr>
<td>Exact. MailBox.Jobs/ Link\Bank Account.vb</td>
<td>Party data-related</td>
<td>1050</td>
<td>3</td>
<td>121</td>
<td>36</td>
<td>3.36</td>
<td>1050.00</td>
<td>406.69</td>
</tr>
<tr>
<td>Exact. Logistics.Core/ Logistics\Tools.vb</td>
<td>Tools</td>
<td>2000</td>
<td>3</td>
<td>130</td>
<td>21</td>
<td>6.19</td>
<td>2000.00</td>
<td>804.76</td>
</tr>
<tr>
<td>Exact. Contracts.Core/ Contract\Tools.vb</td>
<td>Tools: very big</td>
<td>4200</td>
<td>9</td>
<td>454</td>
<td>106</td>
<td>4.28</td>
<td>4200.00</td>
<td>1944.49</td>
</tr>
<tr>
<td>Exact. VAT.Core/ VAT\Tools.vb</td>
<td>Tools: big</td>
<td>3150</td>
<td>4</td>
<td>262</td>
<td>47</td>
<td>5.57</td>
<td>3150.00</td>
<td>1460.51</td>
</tr>
<tr>
<td>Exact. MailBox.Core/ MailBox\Tools.vb</td>
<td>Tools</td>
<td>1100</td>
<td>4</td>
<td>155</td>
<td>47</td>
<td>3.30</td>
<td>1100.00</td>
<td>511.17</td>
</tr>
<tr>
<td>Exact. Sales.Order/ Create\Invoice.vb</td>
<td>Party data-related and big</td>
<td>1000</td>
<td>1</td>
<td>116</td>
<td>25</td>
<td>4.64</td>
<td>1000.00</td>
<td>538.24</td>
</tr>
<tr>
<td>Exact. CashFlow.Link/ Tool\vb</td>
<td>Tools: big</td>
<td>1000</td>
<td>2</td>
<td>208</td>
<td>61</td>
<td>3.41</td>
<td>1000.00</td>
<td>709.25</td>
</tr>
<tr>
<td>Exact. Logistics.Core/ Item\Tools.vb</td>
<td>Tools</td>
<td>1100</td>
<td>3</td>
<td>180</td>
<td>37</td>
<td>4.68</td>
<td>1100.00</td>
<td>875.68</td>
</tr>
<tr>
<td>Exact. CashFlow/ Match\Finder.vb</td>
<td>Complex (not OOP) and big</td>
<td>1500</td>
<td>2</td>
<td>274</td>
<td>53</td>
<td>5.17</td>
<td>1500.00</td>
<td>1416.53</td>
</tr>
<tr>
<td>Exact. CashFlow.NL/ MT\40.vb</td>
<td>Complex (not OOP) and big</td>
<td>2550</td>
<td>6</td>
<td>333</td>
<td>25</td>
<td>12.52</td>
<td>2550.00</td>
<td>3918.76</td>
</tr>
<tr>
<td>Exact. Topics.Core/ Topic\Tools.vb</td>
<td>Tools: big</td>
<td>1050</td>
<td>2</td>
<td>357</td>
<td>75</td>
<td>4.76</td>
<td>1050.00</td>
<td>1699.32</td>
</tr>
<tr>
<td>Exact. VAT.Return/ VAT\Return.vb</td>
<td>Core: Very big</td>
<td>1100</td>
<td>3</td>
<td>795</td>
<td>217</td>
<td>3.66</td>
<td>1100.00</td>
<td>2912.56</td>
</tr>
<tr>
<td>Exact. CashFlow/CashFlow\Planning.vb</td>
<td>Core: Very big</td>
<td>2550</td>
<td>4</td>
<td>1205</td>
<td>181</td>
<td>6.64</td>
<td>2550.00</td>
<td>8022.24</td>
</tr>
<tr>
<td>Exact. Core/ Environment\Environment.vb</td>
<td>Core: Very big</td>
<td>550</td>
<td>2</td>
<td>745</td>
<td>246</td>
<td>3.03</td>
<td>550.00</td>
<td>2726.20</td>
</tr>
<tr>
<td>Exact. Jobs.NL/ BAPI On\Demand.vb</td>
<td>Core: Very big</td>
<td>550</td>
<td>2</td>
<td>809</td>
<td>136</td>
<td>5.95</td>
<td>550.00</td>
<td>4812.36</td>
</tr>
<tr>
<td>Exact. Financial.BE/ GL\Accounts.vb</td>
<td>Extreme test-complexity</td>
<td>1000</td>
<td>1</td>
<td>850</td>
<td>6</td>
<td>158.39</td>
<td>1000.00</td>
<td>1504.16</td>
</tr>
</tbody>
</table>

**Bug Score**: Weighted Sum of fixed Bugs
**Bugcount**: The total amount of fixed Bugs
**CC**: Total cyclomatic complexity
**CC per Function (AVG)**: The average Cyclomatic Complexity (CC / Functions)
**Bug Proneness**: The predicted level of bug-proneness (mapped from Bug Score)
**Test Effort**: Estimation of required test and reengineering effort (CC * Avg. CC per Function)
**Feasibility**: The amount of tackled bug-proneness per test effort (Bug Proneness / Test Effort)
# Appendix D

## Top 20 Bug-prone ASPX Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Bug Score</th>
<th>Bugcount</th>
<th>LOC</th>
<th>Bug Score per LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>docs/FinControlsCallback.aspx</td>
<td>1600</td>
<td>4</td>
<td>91</td>
<td>17.58</td>
</tr>
<tr>
<td>docs/ProProjectTermEntry.aspx</td>
<td>2100</td>
<td>6</td>
<td>151</td>
<td>13.91</td>
</tr>
<tr>
<td>docs/ConContractDetail.aspx</td>
<td>1650</td>
<td>5</td>
<td>130</td>
<td>12.69</td>
</tr>
<tr>
<td>docs/MfgShopOrder.aspx</td>
<td>1100</td>
<td>4</td>
<td>104</td>
<td>10.58</td>
</tr>
<tr>
<td>docs/MfgShopOrderRoutingStepPlans.aspx</td>
<td>1505</td>
<td>4</td>
<td>192</td>
<td>7.84</td>
</tr>
<tr>
<td>docs/LogItemVersionRoutingSteps.aspx</td>
<td>1550</td>
<td>4</td>
<td>222</td>
<td>6.98</td>
</tr>
<tr>
<td>docs/SFAOpportunity.aspx</td>
<td>1350</td>
<td>8</td>
<td>199</td>
<td>6.78</td>
</tr>
<tr>
<td>docs/MfgSubcontractDeliveryPrint.aspx</td>
<td>1500</td>
<td>2</td>
<td>267</td>
<td>5.62</td>
</tr>
<tr>
<td>docs/LogItemCopy.aspx</td>
<td>1500</td>
<td>2</td>
<td>288</td>
<td>5.21</td>
</tr>
<tr>
<td>docs/ProProject.aspx</td>
<td>2650</td>
<td>8</td>
<td>539</td>
<td>4.92</td>
</tr>
<tr>
<td>docs/FinVATCode.aspx</td>
<td>3400</td>
<td>12</td>
<td>715</td>
<td>4.76</td>
</tr>
<tr>
<td>docs/ConMutations.aspx</td>
<td>2050</td>
<td>5</td>
<td>438</td>
<td>4.68</td>
</tr>
<tr>
<td>docs/QuoQuotationRoutingSteps.aspx</td>
<td>1600</td>
<td>5</td>
<td>344</td>
<td>4.65</td>
</tr>
<tr>
<td>docs/MfgQuotationAcceptance.aspx</td>
<td>1500</td>
<td>2</td>
<td>366</td>
<td>4.10</td>
</tr>
<tr>
<td>docs/SlsShipmentReturn.aspx</td>
<td>1550</td>
<td>3</td>
<td>384</td>
<td>4.04</td>
</tr>
<tr>
<td>docs/MfgShopOrderMaterialPlans.aspx</td>
<td>1100</td>
<td>4</td>
<td>276</td>
<td>3.99</td>
</tr>
<tr>
<td>docs/QuoQuotationMaterials.aspx</td>
<td>1600</td>
<td>5</td>
<td>439</td>
<td>3.64</td>
</tr>
<tr>
<td>docs/QuoQuotationQuantities.aspx</td>
<td>1550</td>
<td>4</td>
<td>439</td>
<td>3.53</td>
</tr>
<tr>
<td>docs/ProCallback.aspx</td>
<td>1500</td>
<td>2</td>
<td>429</td>
<td>3.50</td>
</tr>
<tr>
<td>docs/ConMutationsStatistics.aspx</td>
<td>1500</td>
<td>3</td>
<td>437</td>
<td>3.43</td>
</tr>
</tbody>
</table>