Fighting Software Erosion with Automated Refactoring

Gábor Szőke

Department of Software Engineering
University of Szeged

Szeged, 2019

Supervisor:

Dr. Rudolf Ferenc

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

OF THE UNIVERSITY OF SZEGED

Veritas
Virtus
Libertas

University of Szeged

PhD School in Computer Science

"I am a leaf on the wind. Watch how I soar."

— Wash

Preface

As a child, I always liked technology and my favorite characters on TV were always the scientists. Whether it was the main protagonist in *Dexter's Laboratory* or a science officer on *Star Trek*, I never thought that once I would be this close to becoming one. My journey started when I first saw a computer at my cousins' place. Next year in 4th grade I joined a computer class in my school. It was staggering for me and I loved every bit of it. In a year or so I got a computer from my parents and I have been hooked ever since.

First of all, I would like to express my gratitude to my supervisor Dr. Rudolf Ferenc, who kept me motivated while doing my studies. Thank you for inspiring me in times when I needed motivation and that for keeping me on the right path. Thank you for providing me with freedom in my work and for allowing me to travel a lot and, consequently, to meet with many fellow researchers. My special thanks goes to my article co-author and mentor Dr. Csaba Nagy, for guiding me and teaching me a lot of indispensable things about research. I would like to thank Dr. Lajos Jenő Fülöp, who inspired me to take the scientific path. I would also like to thank Dr. Tibor Gyimóthy, for supporting my research work, providing useful ideas, comments, and interesting research directions. I wish to thank David P. Curley for reviewing and correcting my work from a linguistic point of view. My many thanks also go to my colleagues and article co-authors, namely Dr. László Vidács, Dr. Péter Hegedűs, Gábor Antal, Norbert István Csiszár, Dr. Zoltán Ujhelyi, Dr. Ákos Horváth, Dr. Dániel Varró, Zoltán Sógor, Péter Siket, Dr. István Siket, Gergő Balogh and Gergely Ladányi. My thanks also go to Dr. Árpád Beszédes and Dr. Éva Gombás for helping me to arrange a scholarship in Singapore. Many thanks also to all the members of the department over the years, in one way or another, they all have contributed to the creation of this thesis.

An important part of the thesis deals with the Refactoring Project. Most of this research work would not have been possible without the cooperation of the project members. Special thanks to all the colleagues within the department and all participating company members.

Finally, above all I would like to thank my family for providing a pleasant background conducive to my studies, and also for encouraging me to go on with my research.

Gábor Szőke, 2019

Contents

List of Tables

List of Figures

To my family

"Code smells."
— Martin Fowler

1 Introduction

1.1 Software Erosion

At some stage in their career every developer will encounter the code that no one understands and that no one wants to touch in case it breaks down. But how did the software get that bad? Presumably no one set out to make it like that. The process that the software is suffering from is called *software erosion* – the constant decay of a software system that occurs in all phases of software development and maintenance.

The process is also known as software rot, software entropy or software decay. However, these do not adequately capture the notion that it is forces external to the software that are ultimately the cause of problems within the software. The software does not actually decay, but rather suffers from a lack of being updated with respect to the changing environment in which it resides. However, slow deterioration of software over time will eventually lead to performance problems and the software becoming faulty or unusable. Erosion is not something that just happens to the code without someone actively making changes. Rain shapes hills and mountains slowly over time and by analogy change can shape software.

Pressure for change comes from a variety of sources. Most commonly, new features are added to a product to increase its sales value and to satisfy their current users' demands. Similarly, changes in the environment within which the software is deployed happens frequently. Sometimes it is the software environment e.g. different operating system or GUI standards, and technical changes, such as adapting new coding standards. Other times, it is the hardware that changes, like different architecture, better CPUs, Wi-Fi connection instead of cable. They all have an impact on the software.

Software that is being continuously modified may lose its integrity over time if proper mitigating processes are not consistently applied. However, much software requires continuous changes to meet new requirements and correct bugs, and re-engineering software each time a change is made is rarely practical. This creates what is essentially an evolution process for the program, causing it to depart from the original engineered design. As a consequence of this and a changing environment, assumptions made by the original designers may be invalidated, introducing bugs. Where the initial vision for the software does not allow for change, such erosion effects will be seen very quickly.

Stopping software erosion requires management commitment. If managers are only interested in the short-term viability of their software projects then it is hard for developers to find the time and make the effort to tackle the problem. This does not excuse developers from doing what they can to fight decay but this will inevitably make their struggle less effective. With management support you can create a work culture where stopping erosion is valued. This culture is likely to have characteristics such as – an emphasis on regular refactoring, clear assignment of responsibilities, sharing of architectural knowledge and work, frequent communication between the whole group.

1.2 Code Refactoring

In life, software erosion is inevitable. It is typical of software systems that they evolve over time, so they get enhanced, modified, and adapted to new requirements. As a consequence, the source code usually becomes more complex, and drifts away from its original design, hence the maintainability costs of the software increases. This is one reason why a major part of the total software development cost (about 80%) is spent on software maintenance tasks [10]. One solution for preventing the undesirable effects of software erosion, and to improve maintainability is to perform refactoring tasks regularly.

The term *refactoring* was introduced in the PhD dissertation of Opdyke [11]. Refactoring is a kind of reorganization, and it is defined as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*". Technically, it comes from mathematics when an expression is factored into an *equivalence* – the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical.

Refactoring is typically done in small steps. After each small step, the working system's functionally is unaltered. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring does not preclude changing functionality, it just says that it is a different activity from rearranging code. A key insight is that it is easier to rearrange the code correctly if at the same time no change is being made its functionality. Another is that it is easier to change functionality when the code is clean (refactored). Practically speaking, refactoring means making code clearer and cleaner and simpler and elegant. Or, in other words, "*clean up after yourself when you code*". Examples range from renaming a variable to introducing a method into a third-party class which source is unavailable.

1.3 Code Smells and Anti-Patterns

The term refactoring became popular after Fowler published a catalog of refactoring transformations [12]. These transformations were meant to fix so-called 'bad smells' (a.k.a. 'code smells'). Bad smells usually indicate badly constructed and hard-to-maintain code segments. For example, the method at hand may be very long, or it may be a near duplicate of another similar method. Code smells are usually not bugs – they are not technically incorrect and do not currently prevent the program from functioning. Instead, they suggest weaknesses in design that may slow development or increase the risk of bugs or failures in the future. Once recognized, such problems can be addressed by refactoring the source code, that is transforming it into a new form that behaves in the same way as before but it no longer ‘smells’. Refactoring is usually motivated by noticing a code smell.

There are *application-level* code smells: duplicated code, contrived complexity; class-level smells: feature envy, cyclomatic complexity, downcasting; and *method-level* smells: too many parameters, long method. Determining what is and is not a code smell is subjective, and varies by programming language, developer and development methodology. There are tools such as Checkstyle, PMD and FindBugs for Java which to automatically check for certain kinds of code smells.

The benefit of understanding code smells is that is helps you to discover and correct the anti-patterns and bugs that are the real problems. To understand what an anti-pattern is, we have to know first what a pattern is [13]. Over time, many different software developers have had to solve the same or similar problems. How many different developers needed to restrict user access to portions of an application? Or had to communicate object states between threads or machines? Some developers come up with good solutions, while others are able to solve the problem, but do it poorly or inefficiently. These are patterns, and it took a while before we started giving them names. For example, when good developers solve the problem of securing an application, most of the time it may look like what we now call the Role Based Access Control pattern.

Anti-patterns are patterns, but they are just undesirable ones. Taking the previous example, when bad developers solve the problem of securing an application, one may end up with poorly-designed objects, resulting in what is called the Divergent Change anti-pattern (or any number of others). In software engineering there are several well-known anti-patterns, such as Spaghetti Code, Golden Hammer, The Blob, Lava Flow, and Cut-and-Paste Programming (a non-exhaustive list).

1.4 Software Quality

“Quality software is reasonably bug or defect free, delivered on time and within budget, meets requirements and/or expectations, and is maintainable.” [14] The quality of a piece of software is assessed by a number of variables. These variables can be divided into external and internal quality criteria. External quality is what a user experiences when running the software package in its operational mode. Internal quality refers to aspects that are code-dependent, and that are not visible to the end-user. External quality is critical to the user, while internal quality is only meaningful to the developer. [15]

Internal quality is mainly evaluated through the analysis of the software inner structure, namely its source code. A better structure represents an easier maintainable code and a poor structure mirrors hard-to-maintain code. Hence it is also called software maintainability. Measuring maintainability is not a straightforward task. It is usually done by using static analysis techniques which measure different software properties, such as size, complexity, coupling, duplicated code ratio, and the number of coding violations or bad smells.

Maintainability has a direct connection with software evolution costs. For example, if a system is easier to maintain, adding a new feature to it will be straightforward because it is more changeable. Similarly, it will be safer as well, because such a system is less error-prone and modifying existing code will be less likely to cause unwanted bugs.

Keeping software maintainability high is in everybody’s interest. The users get their new features faster and with fewer bugs, the developers have an easier job modifying the code, and the company should have lower maintenance costs. Good maintainability can be achieved via very thorough specification and elaborated development plans. However, this is very rare and only specific projects have the ability to do so. Because software is always evolving, in practice, the continuous-refactoring approach seems more feasible. This means that developers from time to time should refactor the code to make it more maintainable. Maintenance activity like this keeps the code ”fresh” and extends its lifetime.

1.5 Goals of the Thesis

A key goal of this thesis is to contribute to the automated support of software system maintenance. In particular, in the thesis we propose methodologies, techniques and tools for:

1. analyzing software developers behavior during hand-written and tool-aided refactoring tasks;
2. evaluating the beneficial and detrimental effects of refactoring on software quality;
3. adapting local-search based anti-pattern detection to model-query based techniques in general, and to graph pattern matching in particular.

1.6 Research questions

This thesis research is driven by the following research questions:

RQ1: *What will developers do first when they have given the time and money to do refactoring tasks?*

RQ2: *What does an automatic refactoring tool need to meet developers requirements?*

RQ3: *How does manual and automatic-tool aided refactoring activity affect software maintainability?*

RQ4: *Can we utilize graph pattern matching to identify anti-patterns as the starting point of the refactoring process?*

1.7 Outline of the Thesis

The thesis contains 6 chapters. This includes an introduction, a conclusion, and a research domain chapter along with the three main chapters which discuss the results of the thesis. The present thesis is structured as follows.

Chapter 1 provides a short introduction to the basic concepts used in the thesis.

Chapter 2 presents the research project this thesis is built on; and provides background to a few terms and technologies that we will use in later chapters.

Chapter 3 investigates how programmers re-engineer their code base if they have the time and extra money to improve the quality of their software systems. In a project we worked together with five companies where one of the goals was to improve the quality of some systems being developed by them. It was interesting to see how these companies optimized their efforts to achieve the best quality improvements at the end of the project. They are all profit-orientated companies, so they really tried to get the best ROI in terms of software quality. To achieve it, they had to make important decisions on what, where, when and how to re-engineer. We investigated how developers decided to improve the quality of their source code and what was the real effect of the manual refactorings on the quality. We collected this information as experimental data and here we present our evaluation in the form of a case study.

Chapter 4 describes the results of a case study conducted in practice to investigate whether automated refactorings improve code quality. We introduce FaultBuster, a refactoring toolset which is able to support automatic refactoring: identifying the problematic code parts via static code analysis and running automatic algorithms to fix selected code smells. We elaborate on the requirements which make a refactoring tool appealing to developers. We share our experiences which we learned while working with developers who were fixing coding issues with the help of our automated tool in an industrial case study.

Chapter 5 presents a detailed comparison of anti-pattern detection techniques. We provide an observation of memory usage in different ASG representations (dedicated vs. EMF); and run time performance of different program query techniques. For the latter, we evaluate four essentially different solutions: (i) hand-coded visitor queries, (ii) queries implemented in native Java code over EMF models, (iii) generic model queries following a local search strategy and (iv) incremental model queries using a caching technique. We compare the performance characteristics of these query technologies by using the source code of open-source Java projects.

Chapter 6 summarizes the contributions of the thesis with respect to the above research questions. After, the appendix contains a summary of the thesis in English and Hungarian.

1.8 Publications

Most of the research results presented in this thesis were published in journals or proceedings of international conferences and workshops. The *Corresponding publications of the Thesis* section provides a list of selected peer-reviewed publications. Table 1.1 is a summary of which publications cover which results of the thesis.

Chapter	Contribution - short title	Publications
3.	Case study of a large-scale refactoring project	[1], [2], [3]
4.	An automated refactoring framework and an industrial case study	[4], [5], [6], [7]
5.	Benchmarking different anti-pattern detection techniques	[8], [9]

Table 1.1. Relation between the thesis topics and the corresponding publications.

Here, the author should mention that although the results presented in this thesis are his major contribution, from this point on, the term ‘we’ will be used instead of ‘I’ for self-reference to acknowledge the contribution of the co-authors of the papers that this thesis is based on.

"Discovery consists of looking at the same thing as everyone else and thinking something different."
— Albert Szent-Györgyi

2 Research Domain

2.1 The Refactoring Research Project

Much of the research work presented in this thesis was motivated by an R&D project called the *Refactoring Project*. This two-year long project was supported by the EU and a Hungarian national grant. The author was one of the involved researchers in the project and the results of the thesis are connected to this project. Here, we briefly present the project and its goals.

The aim of the project was to develop software tools to support the '*continuous reengineering*' methodology, hence provide support to identify problematic code parts in a system and to refactor them in order to enhance maintainability. Continuous refactoring has many benefits [16], as Kerievsky says "*by continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code*" [17]. This included the development of an automatic refactoring framework and the testing of it on the source code of the industrial partners. Hence, we had an *in vivo* environment and continuous feedback from using the tools. Moreover, the project provided the companies with a good opportunity to refactor their code and improve its maintainability.

Table 2.1. Companies involved in the project

Company	**Primary domain**
Company I	Enterprise Resource Planning
Company II	Integrated Business Management
Company III	Integrated Collection Management
Company IV	Specific Business Solutions
Company V	Web-based PDF Generation

Five experienced software companies were involved in this project. They were founded in the last two decades, and they started developing some of their systems before the millennium. The systems that they refactored in the project consisted of about 2.5 million lines of code altogether, which had been written mostly in Java, and were related to different areas like ERPs (business process management), ICMs (integrated collection management systems), and online PDF Generators (see Table 2.1). By taking part in this project, they got extra budget to refactor their own source code.

2.1.1 Project design

Figure 2.1 offers an overview of the main stages of the project. In the first stage (*Analysis*), we asked the companies to refactor their code manually. We gave them support by using static code analyzers to help them identify code parts that should be refactored in their code (anti-patterns or coding issues, for instance). We asked the developers to provide detailed documentation of each refactoring phases, and explain the main reasons and the steps of how they improved the code fragment in question.

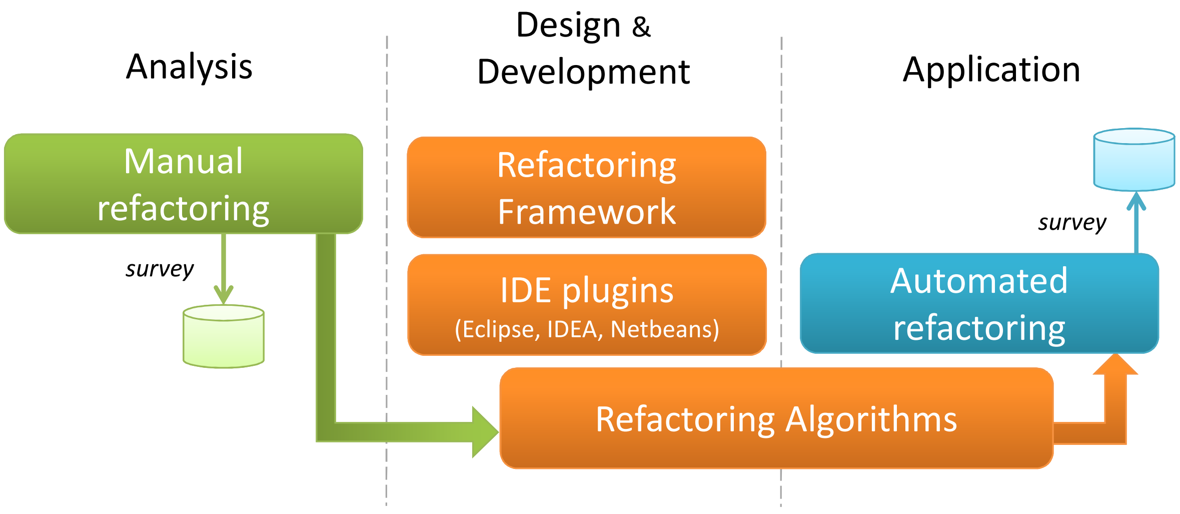


Figure 2.1. Overview of the refactoring project.

In the second stage (*Design & Development*), we designed and implemented a refactoring framework based on the results of the manual refactorings. This framework was implemented as a server-side component that provided three types of services:

- A static source code analyzer toolset to derive low-level quality indicators that could be used to identify refactoring candidates.
- A persistence layer above a database for storing and querying analysis data (with a complete history).
- A set of web services capable of automatically performing various refactoring operations to eliminate certain coding issues and generate a source code patch to be applied on the original code base.

As can be seen from the above list, the framework not only provided refactoring algorithms for the developers, but it also helped to identify possible targets for refactoring by analyzing their systems using a static source code analyzer. The tool is able to give a list of problematic code fragments including coding issues, anti-patterns (e.g. duplicated code, long functions) and source code elements with problematic metrics at different levels (e.g. classes/methods with excessive complexity and classes with bad coupling or cohesion metrics). However, the framework only supports the refactoring of 40 different coding issues, so the companies were just asked to fix issues from this list.

The participating companies took part in the development of the refactoring tools as well. One of their tasks was to develop IDE plugins for their own working environments (Eclipse, IDEA, and Netbeans). So it was the responsibility of the framework to perform the refactoring transformations and generate patches. The IDE plugins were responsible for providing an interface to all the features of the framework by taking advantage of the UI elements of the IDEs. This way, the refactoring process was controlled by the framework and the developers worked in their familiar workspace.

In the third stage of the project (*Application*), the developers used the automatic tool to refactor their code base. Over 7,800 issues got fixed, which fell into about 30 different kinds of issues. Thanks to the project requirements, all the refactorings were well documented.

Taking advantage of this controlled environment, we collected a large amount of data during the refactoring phases. By measuring the maintainability of the given subject systems before and after the refactorings, we got valuable insights into the effect of these refactorings on large-scale industrial projects.

2.2 Measuring Source Code Maintainability

Several maintainability models exist which try to express the maintainability of a software system numerically. Most of these models rely on the observation that the increase of some code metrics (e.g. length of the code, or complexity) indicates a decrease in the maintainability, hence software quality. Chidamber and Kemerer [18] defined several object-oriented metrics; these definitions are *de facto* standards employed in many studies. Gyimóthy et al. [19] validated empirically that the increase of some of the defined metrics (e.g. CBO) indeed increase the probability of faults.

To calculate the absolute maintainability values of systems involved in the Refactoring Project we used the ColumbusQM probabilistic software maintainability model[1]. The ColumbusQM quality model is based on the ISO/IEC 25010 [21] international standard for software product quality. Thanks to this probabilistic approach, the model integrates the objective, measurable characteristics of the source code (e.g. code metrics) and expert knowledge, which is usually ambiguous. At the lowest level, the following properties are considered by the model:

- source code metrics (e.g. some C&K metrics),
- source code duplications (copy&pasted code fragments),
- coding rule violations (e.g. coding style guidelines, coding issues).

The computation of the standard's high-level quality characteristics is based on a *directed acyclic graph* (DAG), whose nodes correspond to quality properties that can be considered low-level or high-level attributes (see Figure 2.2). The nodes without input edges are low-level nodes (*sensor nodes* – shown in white). These characterize a software system from the developers' view, so their calculation is based on source code metrics, or other source code properties (e.g. violating coding conventions). These properties can be calculated by static source code analysis. For this analysis, we use

[1]A detailed description of ColumbusQM is available in the work of Bakota et al. [20]

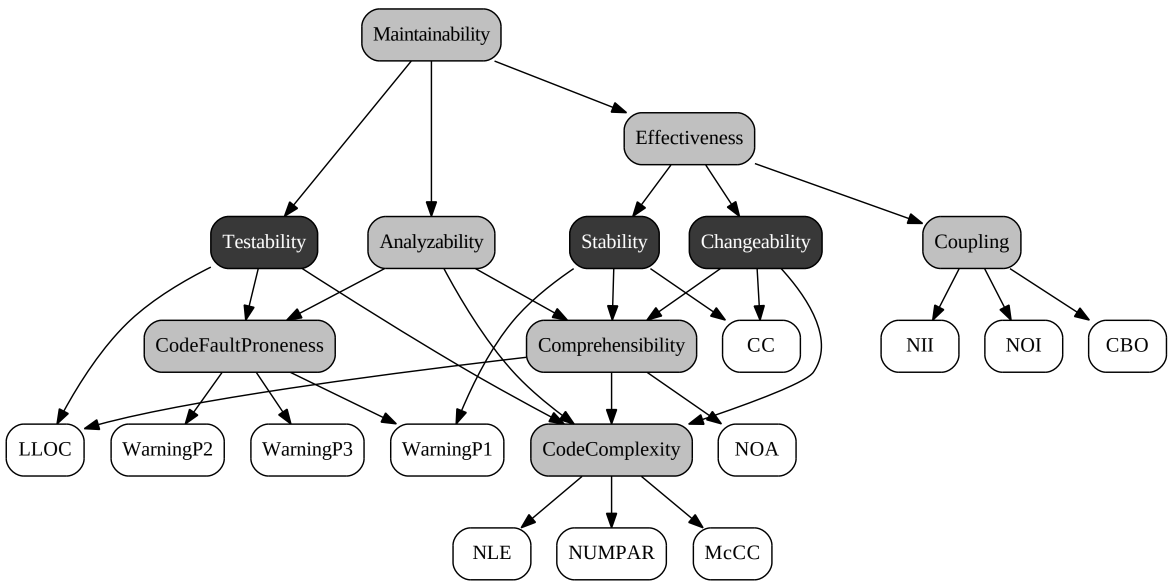


Figure 2.2. An overview of the attribute dependency graph of ColumbusQM [20]. Unfilled nodes represent the sensor nodes (code metrics, number of coding rule violations, number of code clones, etc.) in the model. Aggregated nodes (both light and dark gray nodes) are calculated from these sensor nodes or other aggregated nodes. They were either defined by the ISO/IEC 25010 standard (dark gray) or introduced to show other maintainability attributes (light gray).

an implementation of the ColumbusQM model, called QualityGate [22]. QualityGate uses the free SourceMeter [23] tool, which builds an *abstract semantic graph* (ASG) from the source code, and it uses this graph to calculate metrics, find code clones (duplications) and to find coding issues such as unused code and empty catch blocks.

High-level nodes (called *aggregate nodes*) characterize a software system from the end user's view. They are calculated as an aggregation of the low-level and other high-level nodes. In addition to the aggregate nodes which are defined by the standard (dark gray nodes), there are also some new ones that were introduced to show further external maintainability attributes (light gray nodes). These nodes have input and output edges as well. The edges of the graph show the dependencies between sensor nodes and aggregated nodes. Evaluating all the high-level nodes is performed by an aggregation along the edges of the graph, which is called the *attribute dependency graph* (ADG).

Typically, we wish to know how good or bad an attribute is in terms of maintainability. We use the term *goodness* to express this with the help of the model. To include some degree of uncertainty in the value of goodness, it is represented as a random variable with a probability density function, which is called the *goodness function*. The goodness function is based on the metric histogram over the code elements, as it characterizes the system from the aspect of one metric (from one aspect). As goodness is a relative term, it is expected to be measured by means of a comparison with other histograms. After applying the distance function between two histograms, we get a goodness value for the subject histogram. This value will be relative to the other histogram, but the goal is to be independent. Although, the result will always depend on the histograms in the benchmark (see below), we can get a better estimate by repeating the comparison with a larger set of systems in the benchmark. For every comparison, we get a goodness value which can be basically regarded as a sample of a random variable over the range $[-\infty, \infty]$. Interpolation of the empirical density function leads us to the goodness function of the low-level nodes. There is also a way to aggregate the sensor nodes along the edges of the ADG. Bakota et al. [20] held an online survey, where they asked academic and industrial experts for their opinions about the weights of the quality attributes. The number assigned to an edge is considered to be the degree of contribution of source goodness to target goodness. Taking into account every possible combination of goodness values and weights, and the probability values of their result, they defined a formula to compute the goodness function for each aggregate node. In the end, the top-level node in the ADG, maintainability, will have an aggregated value over the interval $[0, 10]$.

As we mentioned before, each histogram gets compared to several other histograms. In order to do this, it is necessary to have a reference database (benchmark) which contains source code properties and histograms of numerous software systems. This benchmark is the basis for a comparison of the software system to be evaluated. By applying the same benchmark, quality becomes comparable among different software systems, or different versions of the same system.

This qualification methodology is general and independent of the ADG and the votes of the experts. But the latter is language specific, resulting in the need for language-specific ADGs. The ADG for Java is shown in Figure 2.2, which was constructed based on the opinions of over 50 experts. The benchmark for Java contains the analysis results of over 100 industrial and open-source Java systems.

In chapters 3 and 4 we use QualityGate with *ColumbusQM for Java* to objectively measure the maintainability of software systems.

2.3 Refactoring Definition

After the term *refactoring* was introduced in the PhD dissertation of Opdyke [11], Fowler published a catalog of refactoring transformations, where he defined refactoring as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*" [12]. However, there is some controversy between researchers about how to interpret this definition. The common academic reading of this definition is that *"a function has to be semantically equivalent before and after refactoring"*. This is a strict interpretation, for example, when a function contains a fault (i.e. crashes on a `null` input) it has to preserve this behavior even after the refactoring. Kim et al. [24] found in their study that, in practice, developers' views on refactoring usually differ from the academic ones. They found that developers define refactoring simply as *"Rewriting code to make it better in some way."*.

In the following chapters we use the term *"refactoring"* in the form of the latter definition. This perspective is closer to the Refactoring Project, where we investigate developers in industrial contexts; moreover, we asked the participants to separate refactoring commits from normal tasks (i.e. adding features or fixing bugs). Therefore, we were able to analyze refactorings in its more natural state.

"I choose a lazy person to do a hard job.
Because a lazy person will find an easy way to do it."
— Bill Gates

3 Evaluation of Developers' Refactoring Habits

Refactoring source code has many benefits (e.g. improving maintainability, robustness and source code quality), but it means that less time can be spent on other implementation tasks and developers may neglect refactoring steps during the development process. But what happens when they know that the quality of their source code needs to be improved and they can have the extra time and money to refactor the code? What will they do? What things will they consider the most important for improving source code quality? What sort of issues will they address first (or last) and how will they solve them? Is it possible to reflect these changes on a unified quality scale? If so, are the refactoring efforts of developers chime in with the measured metrics?

In this chapter, we assess these questions in an *in vivo* context, where we analyzed the source code and measured the maintainability of six large-scale, proprietary software systems in their manual refactoring phase. We surveyed developers during their refactoring tasks and got insights into what their motives and habits were during the examined period.

3.1 Developers' Insights on Hand-written Refactoring Tasks

One of the major goals of the Refactoring Project was to create automated tool support for refactoring tasks. To create a tool that will *actually* help developers in their everyday work we decided to do a study on how they operate in normal circumstances. This way, we could learn more about what developers think of refactoring, what they preferences are, and what they think they want in an automated tool. Therefore, in the initial step of the Refactoring Project we asked developers of participating companies to manually refactor their own code.

Figure 3.1 gives a brief overview of this phase of the project. Here, we requested developers to provide a detailed documentation of each refactoring, explaining what they did and why to improve the targeted code fragment. We gave them support by continuously monitoring their code base and automatically identifying problematic code parts using a static code analyzer based on the Columbus technology of the University of Szeged [25], namely the SourceMeter product of FrontEndART Ltd. [26] Companies had to fill in a survey with questions targeting the initial identification of steps; that is, evaluating the reports of SourceMeter looking for really problematic code fragments and explaining in the survey why that part of the code was actually a good target for refactoring. After identifying coding issues, they refactored each issue one-by-one and filled out another questionnaire for each refactoring, to summarize their experiences after improving the code fragment. There were around 40 developers involved in the project (5-10 on average from each company) who were asked to complete the survey and carry out the refactorings.

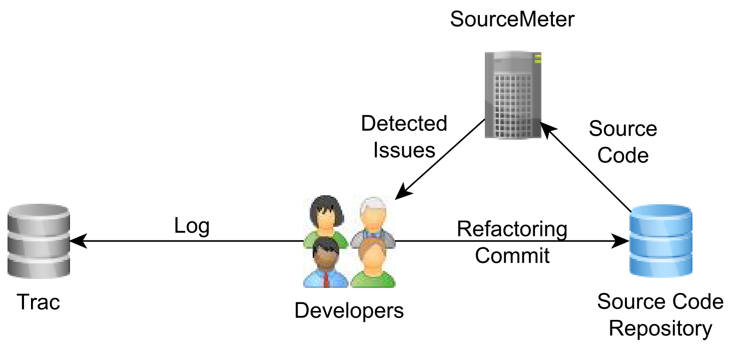


Figure 3.1. Overview of the refactoring process. SourceMeter provided a list of potential problems in the code. Developers could freely choose one of these, or identify a new one, which they fixed and committed to the version control system. They also had to complete a survey for each refactoring in the ticketing system (Trac).

3.1.1 Survey questions

The survey consisted of two parts for each issue. The developers had to fill in the first part before they began refactoring the code, and the second part after the refactoring. In the first part, they asked the following questions:

- Which PMD rule violations helped you identify the issue?
- Which Bad Smells helped you to find the issue?
- Estimate how much it would take to refactor the problem.
- How big is the risk in carrying out the refactoring? (1-5)
- How do you think the refactoring will improve the quality of the whole system's code? (1-5)
- How do you think the refactoring will improve the quality of the current local code segment? (1-5)
- How much do you think the refactoring will improve the current code segment? (1-5)

- How many files will the refactoring have an impact on?
- How many classes will the refactoring have an impact on?
- How many methods will the refactoring have an impact on?

We asked some more questions after developers had finished the refactoring task. These were the following:

- Which PMD rule violations did the refactoring fix?
- Which Bad Smells did the refactoring fix?
- How much time did the refactoring task take?
- Did any automated solution help you to fix the problem?
- How much of the fix for this problem could be automated? (1-5)

With most of the questions, we provided some basic options. For the first question for example we provided a list of PMD rule violations with their names, to help the developers answer the questions quickly. In the questions on the classes and methods impacted, we provided different ranges, namely 1-5, 5-10, 10-25, 25-50, 50-100, 100+. Each question had a text field where the developers could explain their answers and they could also suggest possible improvements and add comments.

3.1.2 Case study

RQ1: What kinds of issues did the companies find most reasonable to refactor?

Our first research question focused on which issue types the companies considered the most important to refactor. We asked the companies which indicators helped them best in finding problematic code fragments in their systems. In our survey, companies could select Bad Code Smells and Rule Violations as indicators on how they found the issues.

In our evaluation, we distinguish a special kind of bad smell which suggests code clones in the system. In Figure 3.2, a distribution can be seen for the issues which helped the companies to identify the problematic code fragments in their code. The intersections in the figure came from the fact that developers could select more than one indicator per issue. The reason why bad smells and clones had no elements in their intersection was because a clone is a special kind of bad smell, as mentioned earlier. The same applies for the intersection of the former group and the rules group (an empty set cannot intersect anything).

When we look at the results in Figure 3.2, we see that the companies found the majority of issues that lay in the sets of rule violations and bad smells. It can also be seen that rule violations alone cover 85% of all the issues found. This also includes 75% of all the bad smells (because of the intersection). So the assumption here is that rule violations are the best candidates for highlighting issues. However to verify this, we also had to look at how many issues the companies fixed in order to choose the best indicator of refactorings.

Figure 3.3 shows the percentage of each fixed issue found from our survey. When we examine the ratio of fixed issues, we see that the bad smells are mostly refactored

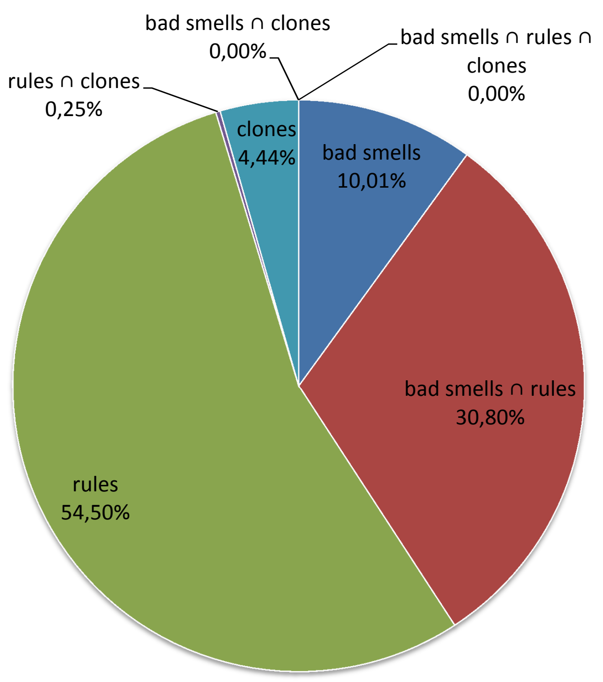


Figure 3.2. Distribution of issue indicators

issues. However if we include the total number of issues, it is clear that rule violations offered the most benefits.

Based on the fact that 85% of all issues were rule violations and developers mostly fixed these issues instead of the others, in future RQs we will just focus on rule violations.

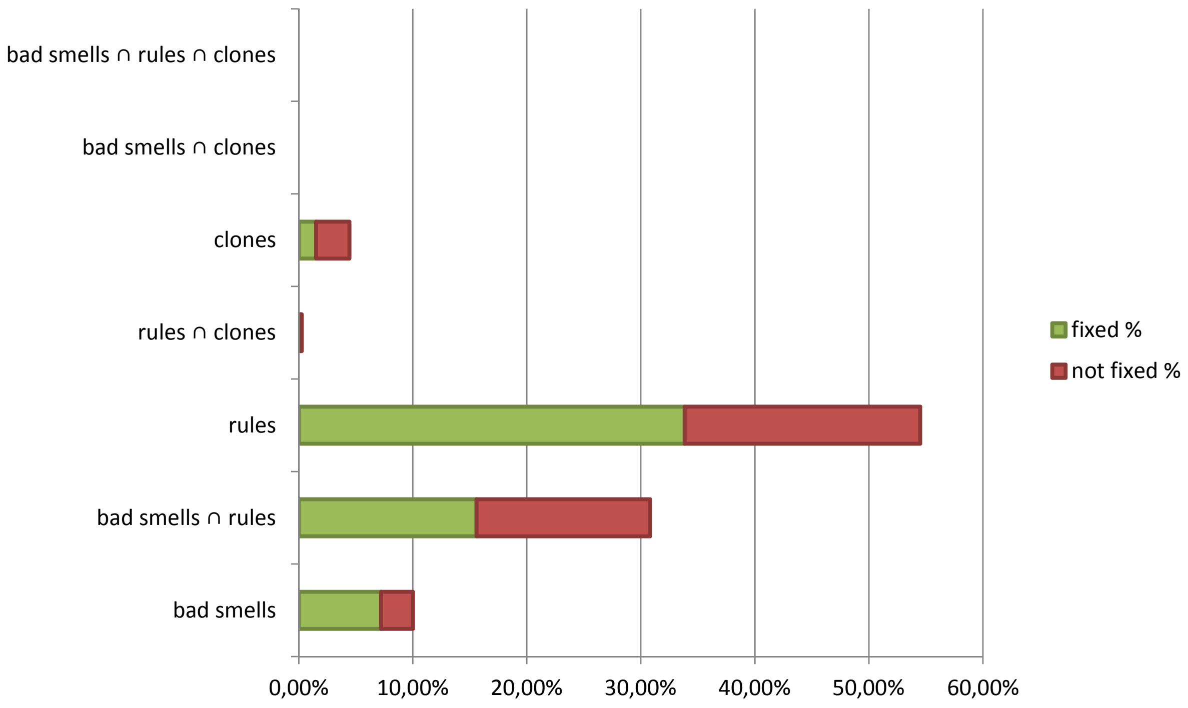


Figure 3.3. Percentage scores of fixed issues for different problem types

RQ2: What are those attributes of refactorings that can help in selecting them?

The rule violations in the survey were provided by the *PMD source code analyzer* tool. In our study, we categorized and aggregated these rules into groups. The groups we used were the Rulesets taken from the PMD website [27]. The companies filled in the survey for 961 PMD refactorings altogether. These 961 refactorings produced 71 different rule violation types over 19 rulesets.

Below, we will examine these rulesets based on different attributes. Based on our survey questions, we created the following attributes:

number of refactorings indicates how many issues were fixed for a certain kind of PMD or ruleset.

average and total time required tells us the total and average time that companies spent on a refactoring. (Values are in work hours.)

estimated time shows how companies estimated the time that a refactoring operation would take. (Values were enumerated between 1 hour and 3-4 days.)

local improvement indicates the subjective opinion of developers on how much the local code segment was improved by the refactoring (Values are between 1-5.)

global improvement indicates the subjective opinion of developers on how much the code improved globally. (Values are between 1-5.)

risk indicates the subjective opinion of developers on how risky the refactoring is. (Values are between 1-5.)

impact is an aggregated number that tells us how many files/classes/methods a refactoring affected. (Values are enumerated between 1-100.)

priority tells us how dangerous a rule violation is, and how important it is to fix it. The priority attribute did not come from the survey; we used the prioritisation of the underlying toolchain. (Values lie between 1-3.)

RQ3: Which refactoring operations were the most desirable based on to the attributes defined above?

The attributes above tell us how risky a refactoring operation is and how much time it will usually take to fix. By combining these attributes, we can discover which rules or rulesets are the most beneficial or riskiest; or by aggregating the first two attributes with time required, we can see which rules will best return the effort we invested in refactoring. Next, we investigate the number of refactorings, time required, improvement and risk.

Number of refactorings Now let us examine the most obvious attribute, namely the number of refactorings the companies performed. The results in Figure 3.4 indicate that the companies dealt with almost every kind of rule violation. The majority of refactored rule violations were found in the *Design* ruleset. This ruleset contains rules that flag suboptimal code implementations, so fixing these code fragments should significantly improve the software quality and perhaps even improve the overall performance. The *Design* ruleset is followed by the *Strict Exceptions*, *Unused Code* and *Braces* categories, which focuses on throwing and catching exceptions correctly, removing unused or ineffective code, and also the use and placement of braces. Some rule violations in the following categories were also fixed in large numbers under the *Basic, Migration, Optimization, String and StringBuffer* rulesets. The other rulesets scarcely came up (like *Empty Code*) or not at all (like *Android*). This is probably due to the fact that the projects did not contain these kinds of violations or contained only false positives.

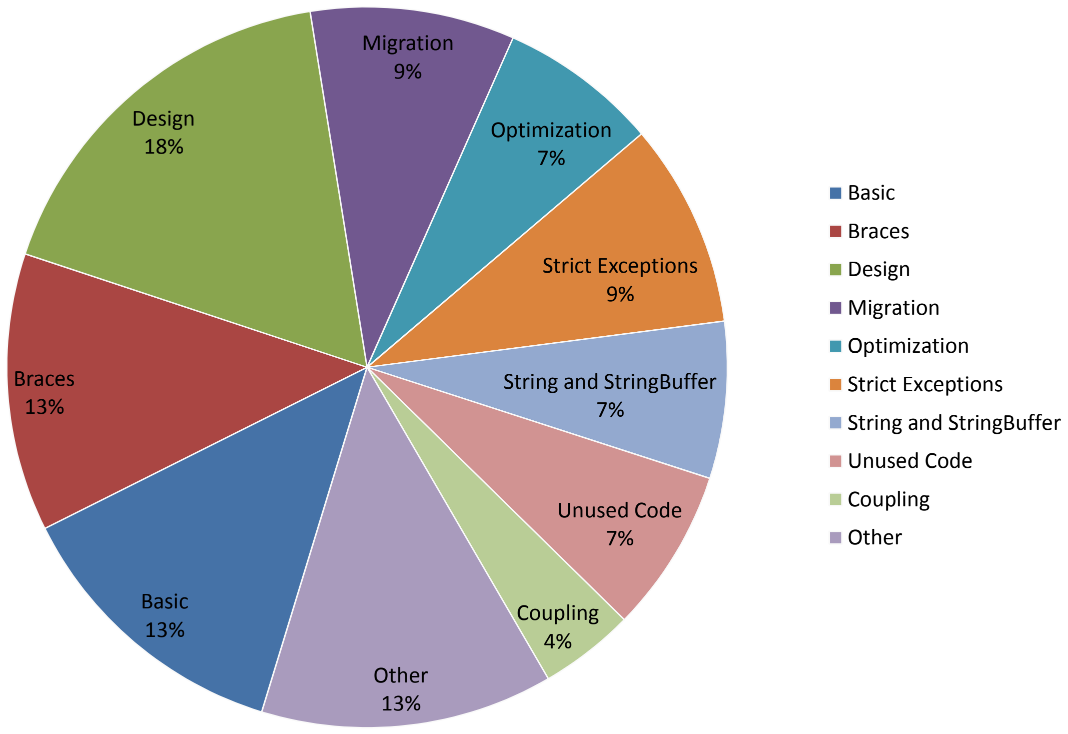


Figure 3.4. Distribution of refactorings by PMD rulesets

Average and total time required After investigating how many refactorings the companies made, we will now examine how much time a refactoring operation took. (Here, we consider the time the developers spent on refactoring their source code, excluding the time they spent on testing and verifying the code.)

When we look at the total time needed for the categories in Figure 3.5, we see that the time distribution of the refactorings shows a similar tendency to the number of refactorings. A linear correlation can be seen between the number of refactorings and the total time spent on them. However, other interesting things were observed when we looked at the average time spent on the different kinds of PMD categories in Figure 3.6. It seems as if the companies spent most of the time on average on *Code Size*, *Security Code Guidelines* and *Optimization* rules. The least time was spent on average on Braces, *Import Statements* and *Java Beans* rules (excluding those rules where no time was spent at all). The *Code Size* ruleset contains rules that relate to code size or complexity (e.g. CyclomaticComplexity, NPathComplexity), while the *Security Code Guidelines* rules check the security guidelines defined by Oracle. The latter guidelines describe violations like exposing internal arrays or storing the arrays

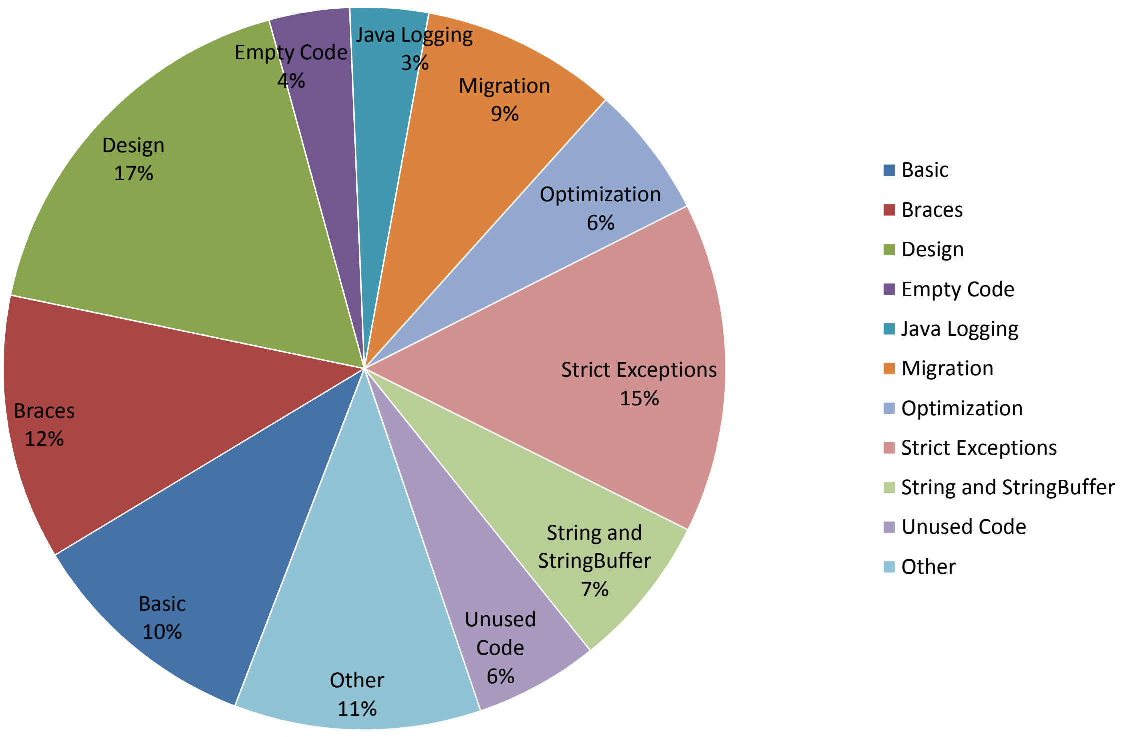


Figure 3.5. Total refactoring durations by PMD rulesets

directly. *Optimization* rules focus on different optimizations that generally apply to best practices. Reducing the complexity of the code, making the application more robust or optimizing it takes time. Apparently, these take the most time. Removing unused import statements or adding or removing some braces usually can be performed quickly, but to find which independent statements should be extracted so as to reduce the complexity is a hard task.

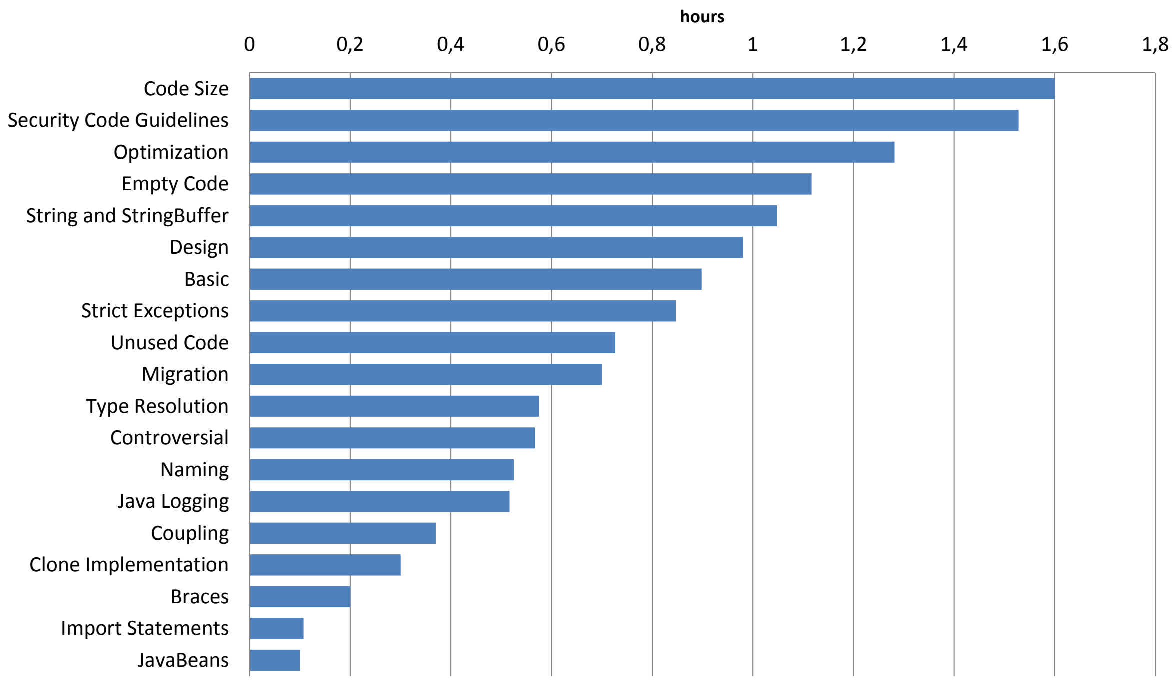


Figure 3.6. Average refactoring durations by PMD Rulesets

Global and local improvement To learn which PMD rule violations fit the attributes best, we summarized and averaged both the global and local improvement values got from the survey. We ranked both sets of values by their position in their data set. The average of the two former values gave us a list of the best improving PMD rulesets. From our results, the best improvements locally and globally are given by the *Strict Exceptions, Coupling and Basic* PMD rulesets. However, rulesets contain a lot of different rules, and hence the categories alone did not give us the proper information we sought. To get further information, a per-rule statistic was required.

For the per-rule statistics, we filtered the results with those cases where the companies did fewer than 4 refactorings of a single kind of PMD rule. This ensured that only relevant data was included in the statistics, and a single-refactored PMD rule could not adversely affect the average values.

Table 3.1 shows a top list of the best improving PMD rule violations. The top list was made by taking the average of the local improvements and summing the average of global improvements, in descending order.

PMD rule violation	**Rank**
PMD_LoC - LooseCoupling	1.
PMD_PLFIC - PositionLiteralsFirstInComparisons	2.
PMD_CCOM - ConstructorCallsOverridableMethod	3.
PMD_ALOC - AtLeastOneConstructor	4.
PMD_ATRET - AvoidThrowingRawExceptionTypes	5.
PMD_ULV - UnusedLocalVariable	6.
PMD_USBFSA - UseStringBufferForStringAppends	7.
PMD_OBEAH - OverrideBothEqualsAndHashcode	8.
PMD_AICICC - AvoidInstanceofChecksInCatchClause	9.
PMD_MRIA - MethodReturnsInternalArray	10.

Table 3.1. Top 10 PMD rules with the best improvements

Risk Table 3.2 shows the riskiest PMD rules used to refactor based on the replies by company experts. We observe that in most cases the riskiest refactorings are for basic Java functionalities. The list includes rules concerning *java.lang.Object*'s *clone*, *hashCode* and *equals* method implementation, proper *catch* blocks and *throws* definitions, array copying and unused variables. All of the previous refactorings increased the quality of the software system (by definition), but fixing these rule violations can have some unexpected consequences. These unexpected consequences are also caused by a previous improper implementation. Of course, if the software code had been written properly in the first place, these unexpected results would have been appeared earlier, and could have been fixed during the development phase.

RQ5: How can we schedule refactoring operations efficiently?

Now we will describe a way of scheduling refactoring operations. First, we will examine how the industrial partners scheduled their refactorings and then we will make recommendations based on these observations.

PMD rule violation	**Rank**
PMD_PCI - ProperCloneImplementation	1.
PMD_ALOC - AtLeastOneConstructor	2.
PMD_SDTE - SignatureDeclareThrowsException	3.
PMD_ACNPE - AvoidCatchingNPE	4.
PMD_LoC - LooseCoupling	5.
PMD_OBEAH - OverrideBothEqualsAndHashcode	6.
PMD_AICICC - AvoidInstanceofChecksInCatchClause	7.
PMD_ULV - UnusedLocalVariable	8.
PMD_AISD - ArrayIsStoredDirectly	9.
PMD_ATNPE - AvoidThrowingNullPointerException	10.

Table 3.2. Top 10 riskiest PMD rules to refactor

How did companies schedule their refactorings? We asked the companies how they scheduled their refactoring operations when fixing rule violations. Each of the companies used the priority attribute that was given for each kind of rule violation, by using the toolchain that was used to extract the rule violations. Priorities were 1, 2, 3, which indicate different levels of threat for each rule violation.

Priority 1 indicates dangerous programming flows.

Priority 2 indicates not so dangerous, but still risky or unoptimized code segments.

Priority 3 indicates violations to common programming and naming conventions.

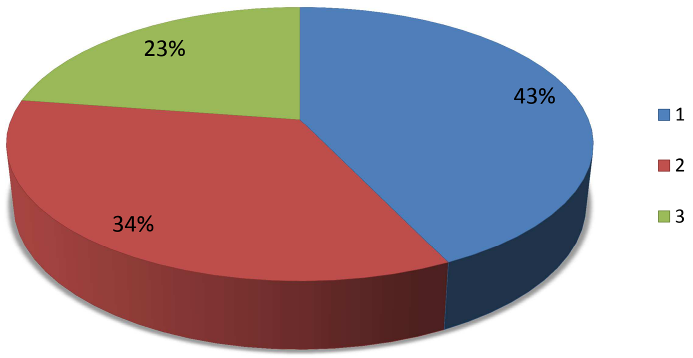


Figure 3.7. Fix rate according to Priority

In Figure 3.7, we can see the percentage scores of all the issues that were fixed for each priority level. They reveal that companies fixed Priority 1 issues the most and Priority 2 issues the second most. This means that companies here opted to fix the most threatening rule violations detected in the code.

Given these attributes, the most efficient way is to start refactoring those issues that had Priority 1 level rule violations. To find out how the companies actually scheduled their refactorings, we split the refactorings into two sets. The first set contains refactorings which were made in the first half of the project, and the other set contains refactorings made in the second half. The results of these experiments are represented in Figure 3.8. They tell us in percentage terms how much was fixed for each priority level in the first half and second half of the project. They reveal that the companies fixed most Priority 1 rule violations in the first half of the project and fixed most Priority 2 rules in the second half. This is consistent with what the companies told us and they provided good feedback on how they scheduled their refactoring process.

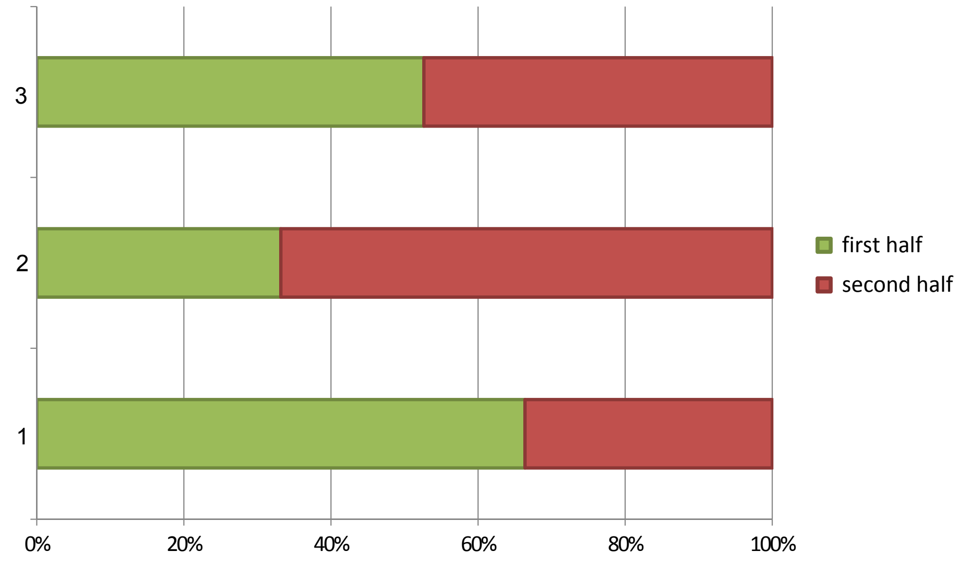


Figure 3.8. Fixes in the first and second half according to Priority

3.1.3 Discussion

Next, we will elaborate on potential threats to validity and some other interesting results that we obtained from our survey.

Threats to Validity

We identified some threats that can affect the construct, internal and external validity of our results.

The first one we encountered was the subjectivity of the survey. The answers to our survey questions were given by developers on a self-assessment basis. We did not measure the time needed or enhancement of refactorings with any automated solution; instead we let the developers answer the survey freely. Nevertheless, we carried out the survey with five industrial partners and therefore with many experts, which surely makes the results statistically relevant.

Another threat that we anticipated was that developers got 'unlimited' extra money and time to do the refactorings, so we could monitor how they refactored their system without any budget pressure. Although they had extra time and money in part of the project, there were still limits that might affect the results and the refactoring process.

Turning to external validity, the generalizability of our results depends on whether the selected programming language and rule violations are representative for general applications. The Java programming language was selected in the assessment together with the companies. These refactorings were made mostly on issues identified by PMD rule violations, hence they were Java specific. However, most of these rules could be generalized to abstract Object-Orientated rules, or they can be specifically defined for other programming languages.

Another threat is that whether fixing PMD rule violations can be viewed as refactoring or not. PMD refactorings are not like traditional refactoring operations that most studies examine (e.g. pull up, push down, move method, rename method, replace conditional with polymorphism). Despite this, Fowler [12] defined refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." During the project we encountered several PMD rule violations and our general experience is that the refactoring of these violations does not alter external behavior, so they can by definition be treated as refactoring.

Overall, our methods were evaluated on large-scale industrial projects, with contributions from expert developers, on a big set of data, which is a rather unique case study in the refactoring research area.

Other results

In our case study (see Section 3.1.2) we summarized our results based on research questions addressed to experts working in five IT companies. However we ran into several interesting cases which were worth mentioning, but could not be incorporated into our research questions.

One of the interesting cases we found was when we searched for the longest-lasting refactorings. We found that *Company A* carried out a *SignatureDeclareThrowsException* refactoring, which lasted 16 hours. The issue occurred in a method of a widely implemented interface, and the problem was that the method threw a simple *java.lang.Exception* Exception-type. This is not recommended because it hides information and it is harder to handle exceptions. The developer assigned to the issue estimated that the work took 1-2 days, and said that the risk was high because it impacted 10-25 files, but it was worth refactoring because the extra information they gained after the refactoring helped improve the maintainability of the source code.

Another intriguing example was with the same search as before. We found that *Company D* performed several *AvoidDuplicateLiterals* refactorings, which took them 7 hours on average to do; and each of the refactorings impacted on more than 100 classes. According to the comments in the survey, they used NetBeans IDE [28] to fix these kinds of issues. NetBeans IDE has a integrated refactoring suite that helps developers to refactor their source code. Here, they used this suite to extract duplicated literals to constant variables. The survey comments revealed that the refactoring suite really helped them in this refactoring task, and it would be a great help if automated solutions could be devised and implemented to tackle other issues as well.

3.2 Case Study on the Effects of Refactoring on Software Maintainability

In Section 3.1 we analyzed questionnaires that the developers filled before and after they manually refactored the code. Here, we investigate how developers decided to improve the quality of their source code and what the real effect of the manual refactorings was on the quality. In this study, we measured the maintainability of six selected systems of four companies who participated in the project. We calculated the quality for the revisions before and after the developers applied refactoring operations. We showed which code smells developers decided to fix and how each refactoring changed the quality of the systems. We examine the data set that we gathered by addressing the following motivating research questions:

- Is it possible to recognize the change in maintainability caused by a single refactoring operation with a probabilistic quality model based on code metrics, coding issues and code clones?
- Does refactoring increase the overall maintainability of a software system?
- Can it happen that refactoring decreases maintainability?

3.2.1 Methodology

In the project, the companies' programmers were required to refactor their own code, hence improve its maintainability, but they were free to choose how they wanted to do it. They could freely choose any coding issues or metrics from the reported problems, and they were also free to identify additional problems in the code by themselves. However, the project required that they filled out the survey (in a Trac ticketing system) and that they gave a thorough explanation on what, why and how they refactored during their maintenance task. Besides completing the survey, we asked them to provide revision information so we could map one refactoring to a Trac ticket and a revision in the version control system (Subversion, Mercurial).

After the manual refactoring phase, we analyzed the selected revisions to assess the change in the maintainability of the systems caused by refactoring commits. Figure 3.9 gives an overview of this process. It was not a requirement of the developers that they commit only refactorings to the version control system, or that they create a separate branch for this purpose. It was more realistic, and some developers asked us in particular to commit these changes to the trunk or development branches so they could develop their system in parallel with the refactoring process. Hence, for each system we identified the revisions (r_{t_1}, ..., r_{t_i}, ..., r_{t_n}) that were reported in the Trac system as refactoring commits, and we analyzed all these revisions along with the revisions prior to them. As a result, we considered the set of revisions $r_{t_1-1}, r_{t_1}, ..., r_{t_i-1}, r_{t_i}, ..., r_{t_n-1}, r_{t_n}$, where r_{t_i} is a refactoring commit and r_{t_i-1} is the revision prior to this commit, which is actually not a reported refactoring commit.

We performed an analysis of these revisions of the source code via the QualityGate SourceAudit tool, mentioned earlier in Section 2.2. To be able to calculate the changes in the maintainability, we had to analyze the whole code base for each revision. That is, a commit with a small local change may also have an impact on some other parts of the source code. E.g., a small modification in a method may result in the appearance of a new clone instance, or changes in coupling metric values of some other classes. Besides analyzing the maintainability of these revisions, we collected data from the version control system as well, like diffs and log messages.

We will now illustrate the use of a simple refactoring through a coding issue that was actually fixed by the developers. In this example, we show the 'Position Literals First In Comparisons' coding issue. In Listing 3.1, there is a Java code example with a simple String comparison. This code works perfectly until we call the 'printTest' method with a null reference. By doing so, we would call a method of a null object, and the JVM would throw a NullPointerException.

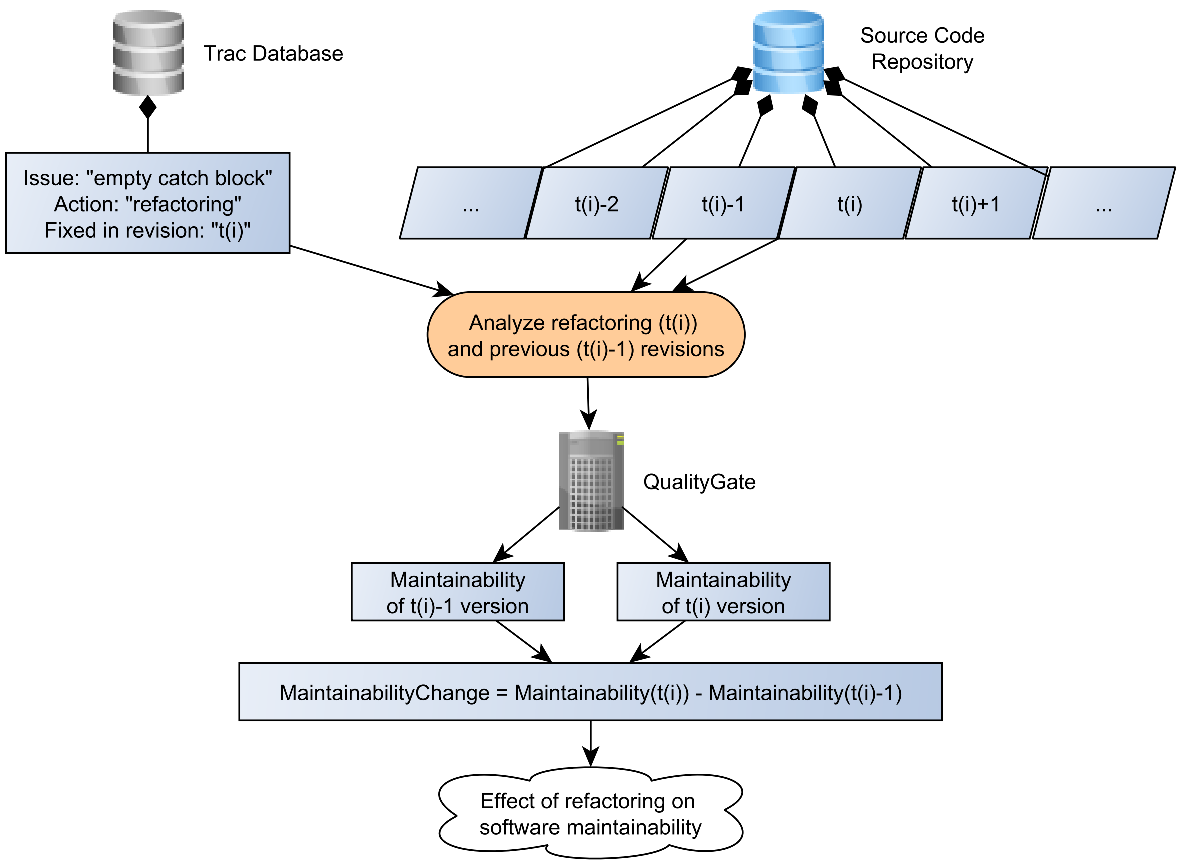

Trac Database
Source Code Repository
Issue: "empty catch block"
Action: "refactoring"
Fixed in revision: "t(i)"
...
t(i)-2
t(i)-1
t(i)
t(i)+1
...
Analyze refactoring (t(i)) and previous (t(i)-1) revisions
QualityGate
Maintainability of t(i)-1 version
Maintainability of t(i) version
MaintainabilityChange = Maintainability(t(i)) - Maintainability(t(i)-1)
Effect of refactoring on software maintainability

Figure 3.9. Overview of the analysis process. We identified the refactoring commits based on the tickets in Trac, and analyzed maintainability of the revisions before/after refactoring commits.

```java
public class MyClass{
 public static void printTest(String a){
  if(a.equals("Test")) {
   System.out.println("This is a test!");
  }
 }
 public static void main(String[] args) {
  String a = "Test";
  printTest(a);
  a = null;
  printTest(a); // What happens?
 }
}
```

Listing 3.1. A code with a Position Literals First In Comparisons issue

To avoid this problem, we have to compare the String literal with the variable instead of comparing the variable with the literal. So to fix this issue, we simply swap the literal and the variable in the code, as can be seen in Listing 3.2. Thanks to this fix, one can safely call the 'printTest' method with a null object and we do not have to worry about a null pointer exception. This and similar refactorings are easy to fix, and with this fix we can avoid critical or even blocker errors.

```
public class MyClass{
 public static void printTest(String a){
  if("Test".equals(a)) {
   System.out.println("This is a test!");
  }
 }
 public static void main(String[] args) {
  String a = "Test";
  printTest(a);
  a = null;
  printTest(a); // What happens?
 }
}
```

Listing 3.2. Sample refactoring of the code in Listing 3.1

3.2.2 Overall Change of Maintainability of the Systems

Table 3.3 shows the size of the six chosen systems and the number of analyzed revisions including the number of refactoring commits. Recall that we determined the refactoring revisions from the ticketing system as those revisions which were marked by the developers as refactoring commits. In addition, we analyzed the non-refactoring revisions prior to the refactoring revisions in order to calculate the change in maintainability (see Section 3.2.1). All in all, we analyzed around 2.5 million lines of code with 732 revisions, out of which 315 were refactoring commits. Developers made 1,273 refactoring operations with these commits. Notice that the project allowed the developers to commit more refactorings together in one patch, but one commit had to consist of the same type of refactoring operations. So one commit possibly included the necessary code transformations to fix more Position Literals First issues, and we did not allow it to have a different type of coding issue in the same changeset.

Table 3.3. The main characteristics of the selected systems: lines of code, total number of analyzed revisions, number of refactoring commits, number of refactoring operations.

System	Company	kLOC	Analyzed Revisions	Refactoring Commits	Refactorings
System A	Comp. I.	1,740	269	136	470
System B	Comp. II.	440	180	38	78
System C	Comp. III.	170	78	15	597
System D	Comp. IV.	38	37	16	18
System E	Comp. IV.	11	57	40	40
System F	Comp. IV.	50	111	70	70
	Total	**2,449**	**732**	**315**	**1,273**

The first diagram in Figure 3.10 shows the change in the maintainability (between each pair of refactoring and its predecessor) of *System A* during the refactoring period. The diagram indicates that maintainability of the system increased over time; however, this tendency includes the normal development commits as well and not just the refactoring commits.

The second diagram in Figure 3.10 shows a sub-period and highlights in red those revisions that were marked as refactoring commits, while the green part indicates the rest of the revisions (i.e, the ones preceding a refactoring commit) which were the normal development commits. It can be seen that those commits that were marked as

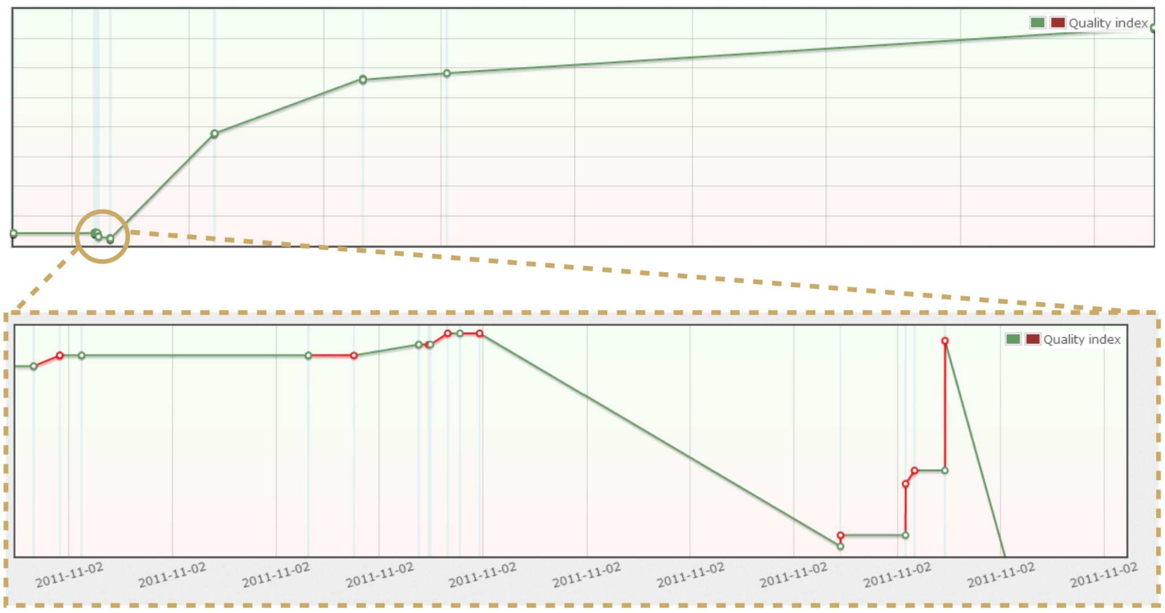


Figure 3.10. Maintainability of *System A* over the refactoring period and a selected subperiod where we highlighted in red the changes in maintainability caused by refactoring commits

refactorings noticeably increased the maintainability of the system, but in some cases the change does not seem to be significant and the maintainability remains unaltered. However, commits of normal development sometimes increase and sometimes decrease the maintainability with larger variance.

Table 3.4. Number of commits which increased or decreased the maintainability of the systems

System	Negative	Zero	Positive
System A	17	94	25
System B	3	18	17
System C	2	5	8
System D	1	7	8
System E	13	9	18
System F	8	30	32
Total	**44**	**163**	**108**

Table 3.4 lists the number of commits for each system which had a positive (or negative) impact on maintainability. If a commit increased the maintainability value it had a positive (beneficial) impact; if it decreased, it had a negative (detrimental) impact; otherwise it did not affect the sensors of the quality model and its impact is considered zero (neutral). As can be seen in Figure 3.11, the results show that for all of the systems the beneficial effects outnumber the detrimental ones. Interestingly, it also indicates that a large proportion of the commits did not have an observable impact on maintainability. The main reason for this is that ColumbusQM does not recognize all the coding issues that were fixed by the developers. As the developers were not aware of the ColumbusQM model, their aim was simply to improve their own code. This included some fixes of coding issues that were detected by the ColumbusQM only

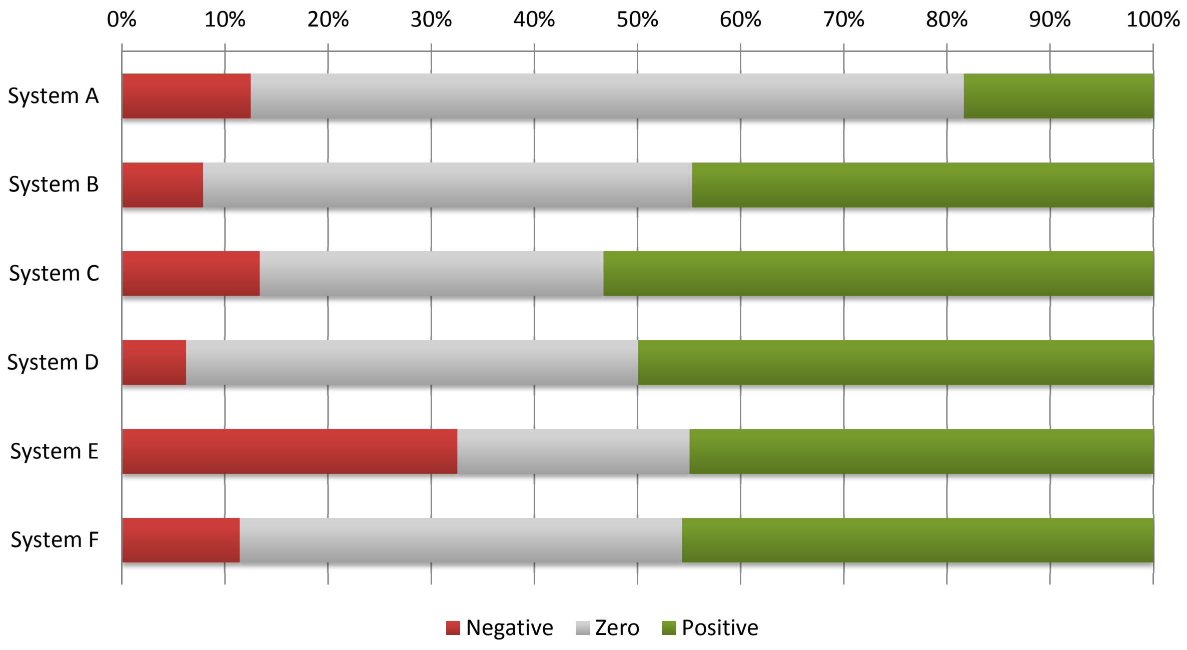


Figure 3.11. Normalized percentage scores of commits with a negative/zero/positive impact on maintainability (negative - red, zero - gray, positive - green)

when the refactoring affected some source code metrics. (Section III elaborates on these refactorings.)

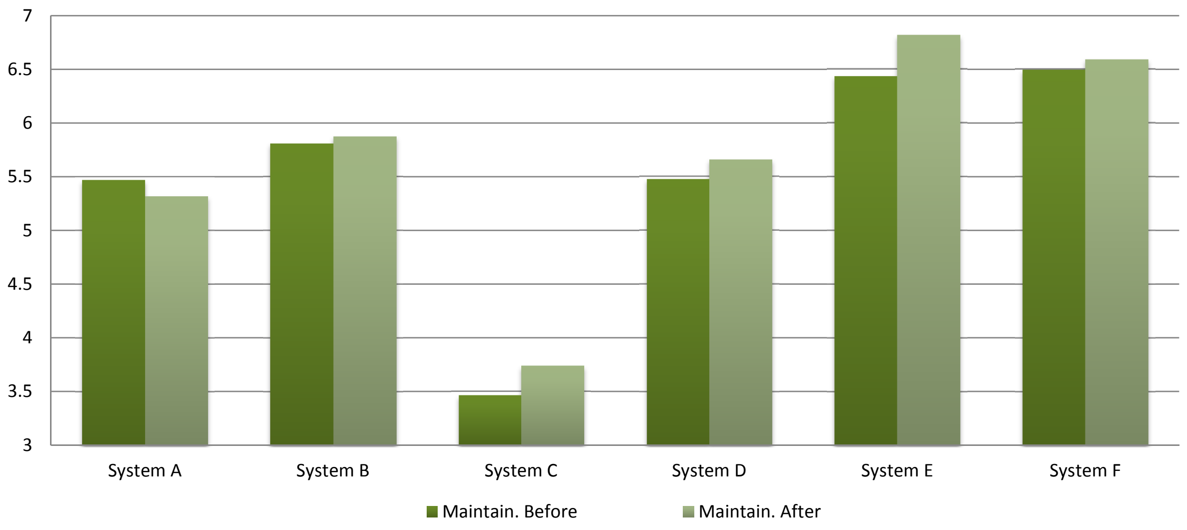


Figure 3.12. Maintainability of the projects before and after the refactoring period

Figure 3.12 shows the maintainability values that we measured before and after the refactoring period of each system in question, and Table 3.5 tells us how the maintainability increased or decreased for these systems. Recall that the value of maintainability can be between 0 and 10, where 0 denotes a system with the hardest maintainability, and 10 denotes a system that is very easy to maintain. The 'Metrics', 'Antipatterns' and 'Coding Issues' columns show for each system the number of different kinds of refactorings that were fixed. Note that they could have fixed more issues with one commit, so it might happen that the aim of a fix was to improve some metrics and eliminate antipatterns together. The 'Total Impr.' column shows the difference; that is, the maintainability improvement at the end of the project. 'Ref. Impr.' shows the total value of the maintainability changes caused by refactoring commits only; hence it shows how refactoring commits improved the maintainability.

Table 3.5. Maintainability of the systems before and after the refactoring period

System	**Metrics**	**Anti-patterns**	**Coding Issues**	**Maintain. Before**	**Maintain. After**	**Total Impr.**	**Ref. Impr.**
System A	0	0	470	5.4699	5.3193	-0.1506	-0.0030
System B	32	34	43	5.8095	5.8762	0.0667	0.0135
System C	15	13	595	3.4629	3.7354	0.2725	0.0767
System D	3	0	17	5.4775	5.6594	0.1819	0.0151
System E	14	8	31	6.4362	6.8190	0.3828	0.0436
System F	15	11	42	6.4972	6.5926	0.0954	0.0716

We measured positive change in the maintainability of five systems out of six and in the case of System F, 75.05% of the maintainability improvement was caused by refactoring commits. Notice, however, that for System A, maintainability decreased by the end of the refactoring period (it had the biggest detrimental impact ratio in Table 3.4). Also, this system had the largest code base among the systems analyzed and its developers decided to fix only coding issues.

3.2.3 Effect of Different Types of Refactorings on the Maintainability

To further investigate the changes made during the refactoring period, we will study the impact of each type of refactoring. For each refactoring ticket, we asked the developers to select what they wanted to improve with the commit:

- Did they try to improve a certain metric value?
- Did they try to fix an antipattern?
- Did they try to fix a coding issue?

In practice, it may happen that a developer wants to fix a coding issue and he may improve a metric value as well in the same commit. Also, many metrics correlate with antipatterns (e.g. large class/long method correlate with LOC). However, in the project developers mostly handled these separately. For coding issues, we asked them in particular to commit refactorings of only one certain kind of issue per commit. But this also means that they were allowed to refactor more from the same kind of coding issue in one commit.

I Metrics

Table 3.6 shows the change in maintainability caused by refactoring commits, where the goal of the developers was to improve certain metrics. (See Table 3.7 for a detailed description of these metrics.) The first thing that we notice here is that the number of these refactorings (74) is very small compared to the total number of refactorings (1,273). It was definitely not the primary goal of the developers to improve the metric values of their systems, although we told them about all the well-known complexity, coupling, and cohesion metrics at the package, class and method levels. One might doubt how well trained these developers were and whether they were really familiar with the meaning of these metrics. To eliminate this factor, for each company, we held a training session where we introduced the main concepts of refactoring and code smells, and then gave them an advanced introduction to metrics. Most of the participating developers attended this training session, including junior and senior developers as well.

Table 3.6. Change in maintainability caused by commits improving metrics

Metrics	**#**	**Average**	**Min**	**Max**	**Deviation**
NMD	1	0.005252	0.005252	0.005252	0.000000
COF	3	0.002691	0.000000	0.006546	0.003425
McCC + NOA	3	0.002299	0.002299	0.002299	0.000000
CLB	10	0.001662	-0.007803	0.017286	0.006616
NII	1	0.001645	0.001645	0.001645	0.000000
McCC	2	0.001323	0.000000	0.002647	0.001872
NA	1	0.001231	0.001231	0.001231	0.000000
LOC	38	0.001007	-0.007617	0.011233	0.003687
NUMPAR	5	0.000382	-0.000108	0.001113	0.000578
NM	1	0.000257	0.000257	0.000257	0.000000
NLE	4	0.000047	0.000047	0.000047	0.000000
NA	1	0.000000	0.000000	0.000000	0.000000
U	2	-0.000083	-0.000165	0.000000	0.000117
NOS	1	-0.000167	-0.000167	-0.000167	0.000000
NOI	1	-0.004062	-0.004062	-0.004062	0.000000

Table 3.7. Description of metrics

Abbreviation	**Description**
NMD	Number of defined methods
COF	Coupling factor
McCC	McCabe's cyclomatic complexity
NOA	Number of ancestors
CLB	Comment lines before class/method/function
NII	Number of incoming invocations
NA	Number of attributes (without inheritance)
LOC	Lines of code
NUMPAR	Number of parameters
NM	Number of methods (without inheritance)
NLE	Nesting level
NA	Number of attributes
U	Reuse ratio (for classes)
NOS	Number of statements
NOI	Number of outgoing invocations

Among those refactorings which fix metrics, it can be seen that complexity metrics (e.g. McCabe's cyclomatic complexity and Number of parameters) and size metrics (e.g. Lines of code) were the most familiar ones that developers intended to improve. The *Average* column of Table 3.6 lists the average of the measured changes in the maintainability caused by these commits. The first entry in the table shows a refactoring which was performed because of the high value of the Number of defined methods metric. In this case, developers realized that they had similar methods in a few of their classes (methods for serialization and deserialization). They did a *Pull-up method* refactoring, which reduced the number of defined methods in the code and had a beneficial impact on the maintainability. Developers also tried to decrease the Coupling factor in their systems with *Move method* and *Move field* refactorings (second row of the table). There were three refactorings where developers attempted to fix a class with high complexity and bad inheritance hierarchy at the same time. In 38 cases, developers wanted to decrease the LOC metric, and five times they fixed methods with too many parameters. It is also interesting to observe that once they targeted the reuse ratio (e.g. to simplify the inheritance tree) and this resulted in a decrease in maintainability. One explanation is that if they wanted a better reuse ratio, they probably needed to introduce a new class (inheriting from a superclass), which might increase the complexity or in the worst case introduce new coding issues or code clones.

II Antipatterns

Table 3.8 shows the average of changes in maintainability when developers fixed antipatterns. Some antipatterns were identified with automatic analyzers (e.g. Long Function and Long Parameter List), but developers could spot antipatterns manually as well and report them to the ticketing system. (Data Clumps is an example for an antipattern identified by a developer.)

Table 3.8. Change in maintainability caused by commits fixing antipatterns

Antipattern	**#**	**Average**	**Min**	**Max**	**Deviation**
Duplicated Code	11	0.003527	-0.007803	0.011233	0.005195
Long Function, Duplicated Code	3	0.002299	0.002299	0.002299	0.000000
Large Class Code	5	0.001586	0.000000	0.006670	0.002872
Shotgun Surgery	1	0.001526	0.001526	0.001526	0.000000
Data Clumps	1	0.001231	0.001231	0.001231	0.000000
Long Parameter List	5	0.000382	-0.000108	0.001113	0.000578
Long Function	40	-0.000084	-0.007617	0.007097	0.002703

As in the case of metrics, fixing antipatterns was not the primary concern of developers. Typically, they fixed Duplicated Code, Long Functions, Large Class Code or Long Parameter List. Most of these antipatterns could be also identified via metrics. In practice, the greatest influence on the maintainability among antipatterns was caused by fixing Duplicated Code segments. Removing code clones can be done for example by using *Extract Method*, *Extract Class* or *Pull-up Method* refactoring techniques. Removing duplications reduces the LOC of the system, increases reusability and improves the overall effectiveness. Interestingly, fixing Duplicated Code sometimes reduced maintainability, as can be seen in the *Min* column of Table 3.8. For instance, in one case, it decreased the maintainability by 0.0078. Developers of Company IV performed an *Extract Superclass* refactoring on two of their classes to remove clones. At first it was not clear why it had a detrimental effect on the maintainability because in most of the other cases it had a beneficial effect. Further investigation showed that they fixed the Duplicated Code, which in fact increased the maintainability as usual, but they introduced two new OverrideBothEqualsAndHashcode coding issues, which together had a bigger detrimental effect than the fix itself. (Fortunately, they fixed the new coding issues in later commits.)

Developers fixed Duplicated Code antipatterns 11 times, Long Function with Duplicated Code 3 times altogether, Large Class Code 5 times, and Long Function antipattern 40 times. Fixing these antipatterns require a larger, global refactoring of the code (e.g. using Extract Method refactoring). These global refactorings induced a larger change in maintainability compared to others. It is also interesting that the deviation of the effects on maintainability were the largest in the case of fixing Duplicated Code, Large Class Code and Long Function antipatterns.

III Coding Issues

Tables 3.9 and 3.10 list the average of measured maintainability changes where developers fixed coding issues. The relatively big number of refactorings tells us that this was what developers really wanted to fix when they refactored their code base. As we previously noted, it is not clear whether a code transformation which was intended to improve the maintainability, but slightly modifies the behavior, should be classified as a refactoring or not. Fixing a coding issue, for instance, a null pointer exception issue may perhaps change the execution (in a positive way), but it is questionable whether this change (fixing an unwanted bug) should be considered a change in the observed external functionality of the program. However, it is obvious that the purpose of fixing coding issues is to improve the maintainability of the code and not to modify its functionality. We will classify all these fixes as refactorings following the refactoring definition of Kim et al. [24], in which they say that refactoring does not necessarily preserve the semantics in all aspects. Nevertheless, we group the coding issues into two groups; namely (1) issues that can be fixed via semantic preserving transformations, and (2) issues which can be fixed only via transformations which do not preserve the semantics of the original code. The SP columns in Tables 3.9 and 3.10 show this information.

Tables 3.9 and 3.10 show the measured average, minimum, and maximum changes and the standard deviation. The coding issues in the rows are those issues which had at least one patch in the manual refactoring period of any system. Some of these coding issues are simple coding style guidelines which can be relatively easily fixed (e.g. IfElseStmtsMustUseBraces), while there are some issues which may indicate serious bugs and need to be carefully fixed (e.g. MethodReturnsInternalArray or OverrideBothEqualsAndHashCode). Issues that are easier to fix were refactored in larger quantities such as IntegerInstantiation and BooleanInstantiation. It is not that surprising that these issues had a relatively low impact on maintainability; however, it is interesting to observe that some of them caused a detrimental change in the maintainability.

The coding issue with the highest average maintainability improvement was UseLocaleWithCaseConversions. This issue warns the developer to use a Locale instead of simple String.toLowerCase()/toUpperCase() calls. This avoids common problems encountered with some locales, e.g. Turkish. The second highest average is the UnsynchronizedStaticDateFormatter issue, where the problem is that the code contains a static SimpleDateFormat field which is not synchronized. SimpleDateFormat is not thread-safe and Oracle recommends separate format instances for each thread. Company IV fixed this issue by creating a new SimpleDateFormat instance to guarantee thread-safety. However, using ThreadLocal would have provided a better solution for both readability and performance.

In the case of the IfElseStmtsMustUseBraces issues, the reason for the detrimental

Table 3.9. Positive maintainability changes caused by commits fixing coding issues. (The *SP* column shows whether a refactoring made a semantic preserving transformation or not.)

SP	**Coding issue**	#	**Avg**	**Min**	**Max**	**Dev**
✗	UseLocaleWithCaseConversions	4	0.008748	0.005894	0.012439	0.002938
✗	UnsynchronizedStaticDateFormatter	1	0.008618	0.008618	0.008618	0.000000
✓	AvoidInstanceofChecksInCatchClause	5	0.003825	0.000000	0.017286	0.007549
✓	ExceptionAsFlowControl	1	0.003139	0.003139	0.003139	0.000000
✗	NonThreadSafeSingleton	1	0.002977	0.002977	0.002977	0.000000
✓	AvoidCatchingNPE	3	0.002341	0.001627	0.003484	0.001000
✗	EmptyCatchBlock	11	0.002175	0.000000	0.007559	0.002849
✗	OverrideBothEqualsAndHashcode	8	0.001768	0.000000	0.005922	0.004241
✓	EmptyIfStmt	1	0.001286	0.001286	0.001286	0.000000
✓	UnusedPrivateField	9	0.000729	-0.004062	0.007533	0.003016
✓	PreserveStackTrace	11	0.000457	-0.000389	0.001942	0.000904
✗	SignatureDeclareThrowsException	23	0.000348	0.000000	0.001526	0.000692
✗	SwitchStmtsShouldHaveDefault	4	0.000323	-0.000167	0.000642	0.000364
✓	UseStringBufferForStringAppends	17	0.000289	-0.009357	0.012077	0.007609
✓	ArrayIsStoredDirectly	2	0.000273	0.000183	0.000363	0.000127
✓	UnusedLocalVariable	4	0.000223	-0.000247	0.000828	0.000463
✓	LooseCoupling	16	0.000212	0.000000	0.002647	0.000830
✓	AvoidDuplicateLiterals	454	0.000121	0.000121	0.000121	0.000000
✓	UnnecessaryLocalBeforeReturn	43	0.000108	0.000000	0.000585	0.000459
✓	UnnecessaryWrapperObjectCreation	118	0.000083	0.000083	0.000083	0.000000
✗	AvoidPrintStackTrace	32	0.000069	0.000000	0.000185	0.000304
✓	SimplifyConditional	39	0.000010	0.000000	0.000125	0.000061

Table 3.10. Zero or negative changes in maintainability by commits fixing coding issues. (*SP* column shows whether a refactoring did a semantic preserving transformation or not.)

SP	**Coding issue**	#	**Avg**	**Min**	**Max**	**Dev**
✓	AvoidSynchronizedAtMethodLevel	8	0.000000	0.000000	0.000000	0.000000
✓	ConsecutiveLiteralAppends	1	0.000000	0.000000	0.000000	0.000000
✓	MethodReturnsInternalArray	8	0.000000	0.000000	0.000000	0.000000
✓	ReplaceHashtableWithMap	1	0.000000	0.000000	0.000000	0.000000
✓	UseIndexOfChar	48	0.000000	0.000000	0.000000	0.000000
✓	UnusedModifier	31	0.000000	0.000000	0.000000	0.000000
✓	BooleanInstantiation	47	-0.000016	-0.000273	0.000235	0.000305
✓	IntegerInstantiation	84	-0.000019	-0.000247	0.000014	0.000247
✓	IfElseStmtsMustUseBraces	117	-0.000111	-0.000456	0.000186	0.001406
✓	BigIntegerInstantiation	21	-0.000156	-0.003587	0.000974	0.001110
✓	InefficientStringBuffering	12	-0.000264	-0.002649	0.000128	0.000846
✓	UnusedPrivateMethod	2	-0.000863	-0.002729	0.001002	0.002638
✗	AvoidCatchingThrowable	2	-0.001654	-0.003307	0.000000	0.002339
✓	AddEmptyString	9	-0.001833	-0.004527	0.000677	0.002117

change in maintainability is the increased number of the code lines in the modified methods. The sensors of the maintainability model will change at a low level; that is, the number of issues and the LOC metric. These changes will affect the higher level, aggregated maintainability attributes like CodeFaultProneness and Comprehensibility and also the Maintainability. A simple example of this situation is shown in listings

3.3 and 3.4. A simple method with 5 lines could grow to 14 lines if we apply all the necessary refactorings. What is more, this kind of issue has only minor priority so there is a good chance that the beneficial change in the number of issues will have a smaller influence on the maintainability than the detrimental change caused by the increased number of lines of code.

```
public static int doQuant(int n) {
  if ( n >= 0 && n < 86) return 0;
  else if (n > 85 && n < 170) return 128;
  else return 255;
}
```

Listing 3.3. Sample code with an IfElseStmtsMustUseBraces issue. LOC: 5

```
public static int doQuant(int n) {
  if ( n >= 0 && n < 86)
  {
    return 0;
  }
  else if (n > 85 && n < 170)
  {
    return 128;
  }
  else
  {
    return 255;
  }
}
```

Listing 3.4. A sample refactoring of the code in Listing 3.3. LOC: 14

In the case of InefficientStringBuffering, the reason for the detrimental change in maintainability is also the modified number of lines of code. Listing 3.5 shows this kind of issue in a code sample that needs to be refactored. Some of the developers decided to fix this issue, as can be seen in Listing 3.6. This way, there were no new lines added to the code, and the effect of the refactoring was simple; namely, one coding issue vanished.

```
String toAppend = "blue";
StringBuffer sb = new StringBuffer();
sb.append("The sky is" + toAppend);
```

Listing 3.5. A code with InefficientStringBuffering issue

```
String toAppend = "blue";
StringBuffer sb = new StringBuffer();
sb.append("The sky is").append(toAppend);
```

Listing 3.6. A sample refactoring of the code in Listing 3.5

Other developers preferred to fix this problem, as can be seen in Listing 3.7. This way, the issue vanishes as well, but there is a side effect. At least one new code line appears in the code, which again affects the lines of code metric, hence it had a slight impact on the maintainability.

```
String toAppend = "blue";
StringBuffer sb = new StringBuffer();
sb.append("The sky is");
sb.append(toAppend);
```

Listing 3.7. Another way of refactoring Listing 3.5

Another interesting refactoring was the one where Company III refactored Avoid Duplicate Literals coding issues. This kind of issue tells us that a code fragment containing duplicate String literals can usually be improved by declaring the String as a constant field. Refactoring these issues helps to eliminate dangerous duplicated strings, which should improve stability and readability. Although this was the manual phase of the project (where the companies could not yet use the refactoring tool that we intended to develop later), we spotted an interesting commit message where Company III refactored this coding issue with the help of the Netbeans IDE. Netbeans was able to assist them in finding and extracting duplicated string literals into constant fields. The fix was simple and straightforward so we decided keep these refactorings as valuable commits, and not to filter out them from the study. The developers eliminated 454 issues in one commit which covered more than 20,000 lines of code. The quality increase of this commit is quite large; and it improved the maintainability index of the whole system by 0.055.

In some cases, the measured change in maintainability was 0. The reason for this lies in a pitfall of the maintainability model, where these minor priority issues were not taken into account by the maintainability model. Hence, when these issues were fixed, the model did not recognize the change in the number of issues. Fixing these issues required only small local changes that did not influence other maintainability attributes either, so complexity and lines of code remained unaltered, for instance. As a result, the measured change in maintainability was, apparently 0.

3.2.4 Impact of Non-Refactoring Commits

We were able to analyze the systems only in their refactoring period when developers performed some refactoring tasks on their code. As a result, we have the analysis data for each system before and after a refactoring commit was submitted to the version control system. We did not analyze other commits, so we do not have analysis data for other non-refactoring commits. However, we have some opportunities here to study the impact of non-refactoring commits and compare them to refactorings. We analyzed the revisions before refactoring commits to explore the maintainability of a system between two refactoring commits. Suppose that r_i and r_j revisions are consecutive refactoring commits of a system and $j > i$. In this case, we analyzed the revisions r_{i-1}, r_i, r_{j-1} and r_j, following the same sequence of the commits. A change in the maintainability between the revisions (r_{i-1}, r_i) and (r_{j-1},r_j) is caused by two different refactoring commits, but the change in the maintainability between (r_i,r_{j-1}) is because of several non-refactoring commits. These changes measured between two consecutive refactoring commits, (which are caused by other, non-refactoring commits) makes it possible to compare the impact of refactoring and non-refactoring commits.

Although these group together several smaller (non-refactoring) commits, we can treat them as normal development tasks. For simplicity, we will refer to these as *development commits* in the rest of this section.

It seems a reasonable assumption that refactoring commits often have a positive effect on maintainability, but this would not be true for development commits. To investigate this assumption, we counted the commits which increased/decreased or had zero effect on maintainability. We use this data to study their independence with a *Pearson's chi-square test*. The input data is presented in Table 3.11.

Table 3.11. Number of refactoring and development commits which had a negative/zero/positive impact on maintainability

Commit type	**Negative**	**Zero**	**Positive**
Refactoring	44	163	108
Development	63	167	139

We define the following null hypothesis: "*for each commit, its effect on maintainability is independent of the type of the commit (refactoring or development)*". Then the alternative hypothesis is: "*for each commit, its effect on maintainability is dependent on the type of the commit (refactoring or development)*". As a result of a chi-square test, we get a p-value of 0.2156, which is greater than the 0.05 significance level. Hence, we accept the null-hypothesis that the type of the commit and its effect on maintainability are independent.

Figures 3.13, 3.14, 3.15 show how the maintainability of the systems changed over time during the refactoring period. Revision numbers are obfuscated, but their order follows the original order of the commits. In the case of System A, developers refactored four submodules of the system, and we show these submodules separately in the diagram. The diagrams confirm that refactoring and non-refactoring results in varying changes in the maintainability. Just as we can spot refactoring/development commits which suddenly improve the measured values, we can spot their counterparts which suddenly decrease these values. It is easy to see, however, that the maintainability of the systems displayed a positive tendency in the refactoring period.

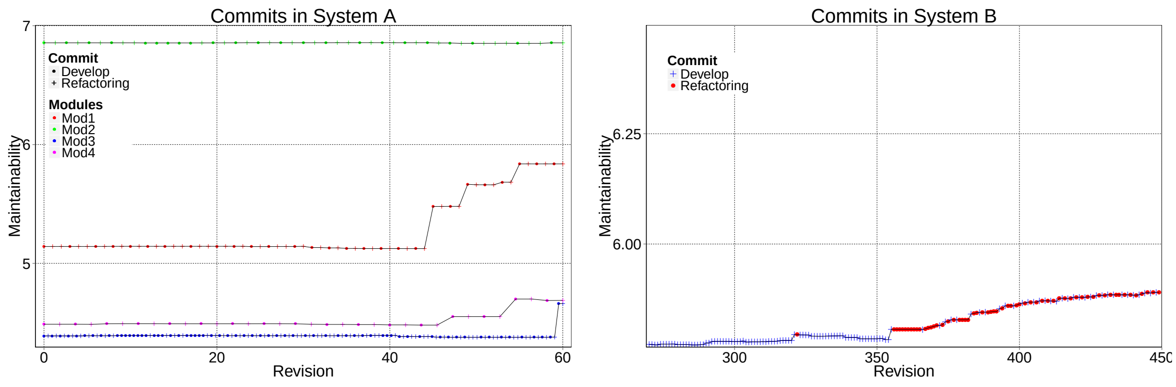


Figure 3.13. Maintainability of systems A and B during the refactoring period (revision numbers have been obfuscated, but they are in their original order)

We should also note that most of the refactorings presented in our study may be considered as small local changes and these commits are likely to have a small

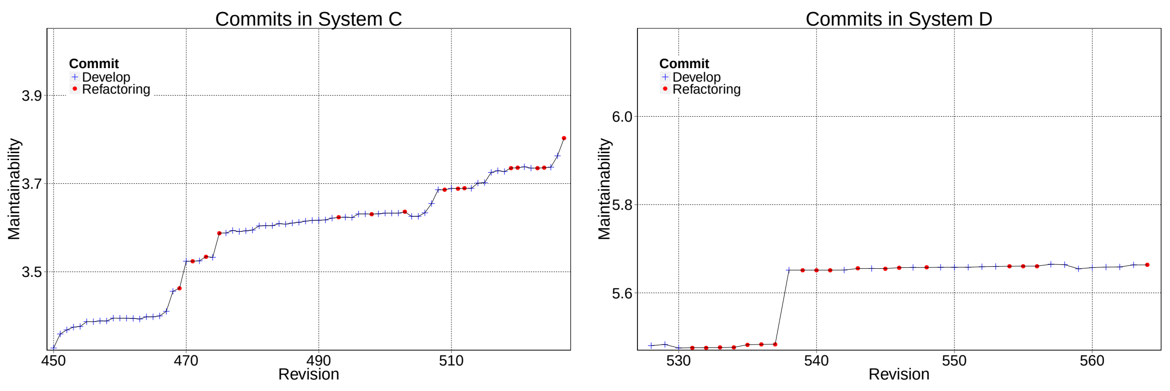


Figure 3.14. Maintainability of systems C and D during the refactoring period (revision numbers have been obfuscated, but they are in their original order)

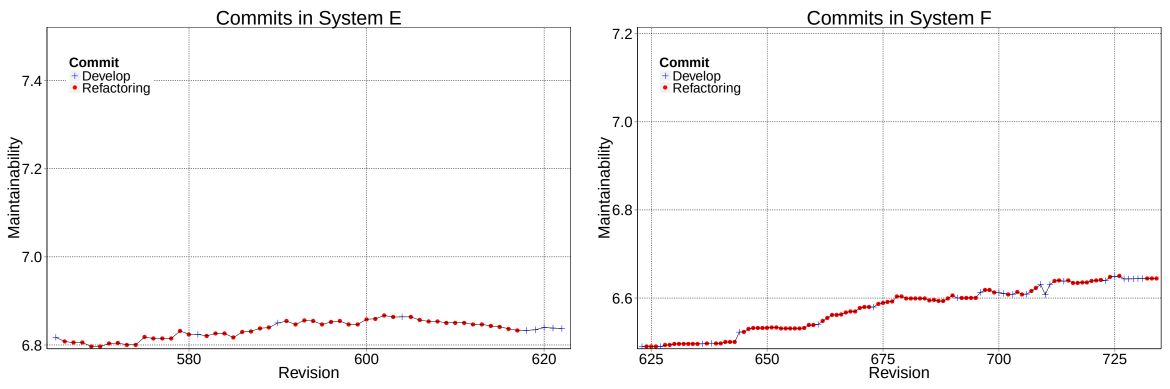


Figure 3.15. Maintainability of systems E and F during the refactoring period (revision numbers have been obfuscated, but they are in their original order)

impact on the global maintainability. So the question arises of whether the improving tendency is because developers take quality more into account in their new code, or the ColumbusQM model is more sensitive to larger code changes.

Table 3.12. The average change in maintainability of refactoring commits normalized by the change in lines of code for each system

System	**Maint. Change Avg.**	**Change in LOC Avg.**	**Maint. Change Avg. per LOC**
System A	-0.000087	30.55	0.000006
System B	0.000589	24.38	0.000099
System C	0.008362	64.73	0.000163
System D	0.000837	2.56	0.000266
System E	0.000092	6.12	-0.000504
System F	0.001441	6.28	0.000677

Besides maintainability, we measured the lines of code metric of the systems. Simply from the lines of code we can see the final difference in the newly added or deleted lines, but we cannot see the exact number of modified lines. Nevertheless, the difference in added/deleted lines is a good estimation of the size of the commit. Tables 3.12 and 3.13 show the average change in maintainability of refactoring and development commits normalized by the change in lines of code for each system. Recall that 'devel-

Table 3.13. Average change in maintainability of development commits normalized by the change in lines of code for each system

System	Maint. Change Avg.	Change in LOC Avg.	Maint. Change Avg. per LOC
System A	0.009068	-117.88	-0.000001
System B	0.000693	215.69	0.000018
System C	0.005652	19.31	0.000665
System D	0.009329	-48.00	0.000365
System E	0.001922	-5.75	0.000808
System F	0.001203	12.04	0.000088

opment commits' group together more commits. Hence, these are likely to be larger structural changes. Table 3.14 shows the Pearson's r correlation coefficients and p significance levels between the change in lines of code and the change in maintainability for all the commits of each system. These results suggest there is a strong correlation between the size of the commit and its effect on maintainability. Hence, we should acknowledge that the ColumbusQM model is more sensitive to larger code changes. Still, the smaller changes of the refactoring commits had a measurable impact on the global maintainability as well. Notice also that for some systems the correlations are negative (also when we consider them all together). Moreover, in the case of System D, they indicate a perfect negative linear relationship between variables. The change in the lines of code may be negative (when they delete lines). Hence, this means that sometimes when they remove more lines, they improve the maintainability more notably. Indeed, in the case of System D, they had five commits (out of 36) where in the 'largest' commit they removed 906 lines and obtained their best maintainability improvement of 0.1679 (see the online appendix for details).

Table 3.14. Pearson's r correlation coefficient and p significance levels between the change in lines of code and the change in maintainability

System	r	p
System A	-0.5	<0.01
System B	0.48	<0.01
System C	0.17	0.127
System D	-0.99	<0.01
System E	0.18	0.171
System F	-0.07	0.432
All	-0.42	<0.01

In spite of this, when Bakota et al. evaluated the ColumbusQM model [20] on industrial software systems, they found that "the changes in the results of the model reflect the development activities, i.e. during development the quality decreases, during maintenance the quality increases." Here, we studied a refactoring period and in contrast to Bakotat et al. we found that normal development commits usually improved the quality. This acknowledges that developers tended to take quality more into account in their new code. They also mention this to us later at the end of the project.

3.2.5 Discussion of Motivating Research Questions

Is it possible to recognize the change in maintainability caused by a single refactoring operation with a probabilistic quality model based on code metrics, coding issues and code clones?

We applied the ColumbusQM maintainability model to measure changes in the maintainability of large-scale industrial systems before/after refactoring commits. Our measurements revealed that the maintainability changes induced by refactoring operations can be seen in most of the cases. One particular change usually caused only a small change, which is quite natural considering that we analyzed 2.5 million lines of code altogether, and a particular refactoring operation usually affects only a small part of it. However, with some refactorings (mostly those involving fixing local coding issues) the model did not display any changes in the maintainability. This was due to the fact that these refactorings were very local, meaning that the sensors of the model did not recognize any changes in the metric values. By fine-tuning the maintainability model, these cases might become detectable.

Does refactoring increase the overall maintainability of a software system?

After the refactoring period, the overall maintainability of the software systems improved and the maintainability model was able to measure this improvement in five out of the six systems. Commits which fixed more coding issues had a relatively higher impact on maintainability. Similarly, we observed in the tables that when developers fixed more metrics or antipatterns together, they induced a bigger change compared to others. Hence, a larger refactoring has a noticeable, positive impact on the maintainability, which is measurable using static analysis techniques.

Can it happen that refactoring decreases maintainability?

Our findings reveal that some refactoring operations might have a negative impact on the maintainability of the system, although its main purpose is to improve it. It is not easy to decide how to fix an issue and balance its effects as it might happen that we want to improve one maintainability attribute, but we debase others.

3.2.6 Additional Observations

Overall, based on our results and analyses, there are some additional interesting observations that deserve to be discussed further.

Developers went for the easy refactorings

Although each participating company could take their time to perform large, global refactorings on their own code, the statistics tell us that they did not decide to do so. They went for the easy tasks, like the small code smells, which they could fix quickly. There might be several reasons for it, as fixing these code smells was relatively easy compared to others. Fixing a small issue which influences just the readability does not require a thorough understanding of the code so developers can readily see the problem and fix it even if it was not written by themselves. In addition, testing is easier in these cases too. Still, a larger refactoring may contain more difficulties: it requires a better knowledge and understanding of the code; it must be designed and applied more carefully; or it may happen that permission is needed to change things across components/architecture. It remains a future research question as to which choice is better in the long term in such a situation. Should we fix as many small issues as we can, or perform only a few, but large, global refactorings and restructure the code?

Developers did not refactor just to improve metrics or avoid antipatterns

Our results suggest that developers did not really want to improve the metric values or avoid certain antipatterns in their code; they simply went for the concrete problems and fixed coding issues. One reason that we must consider here is that developers may not really be aware of the meaning of metrics and antipatterns. Though we are certain that they were aware of the definition of some metrics and code smells (because we trained them for the project), they probably had no experience in recognizing and fixing problematic classes with bad cohesion or coupling values, for instance. They were not maintainability experts who were experienced in studying reports of static analyzers. This seems to tie in with the previous finding that developers went for the low-hanging fruit, and chose the easier way of improving maintainability.

Fixing more complex design flaws (e.g. antipatterns or more complex coding issues) might have a better impact on the maintainability

In Figure 3.16, we show the effect of the average impact of different refactoring types (metrics, antipatterns, coding issues) on the maintainability among all the refactoring commits, and we list the corresponding min/max/deviation values in Table 3.15. As we saw previously, developers fixed mostly coding issues, but notice that those coding issues which required a fix that modified the semantics of the code had a larger impact on maintainability, just like that for antipatterns or metrics. Taking into account how the ColumbusQM calculates maintainability, this is mainly because fixing a more complex issue (antipattern) has a bigger impact on the full code base and not just some local parts of it. Another observation here is that we see developers fixed the Duplicated Code antipattern the most often, which is the number one in Martin Fowler's dangerous bad-smell list [12].

Table 3.15. Average, minimum and maximum impact on maintainability of different refactoring types

Types	Average Change	Minimum Change	Maximum Change	Deviation of Change
Metrics	0.000995	-0.007803	0.017286	0.003854
Antipattern	0.000832	-0.007803	0.011233	0.003524
Coding Issues	0.001080	-0.003307	0.012439	0.003393
Coding Issues (SP)	0.000074	-0.009357	0.017286	0.004392

Developers learned to write better code during the refactoring period

All the systems that we studied in the refactoring period displayed an improvement in source code maintainability, even if we only take into account the revisions where

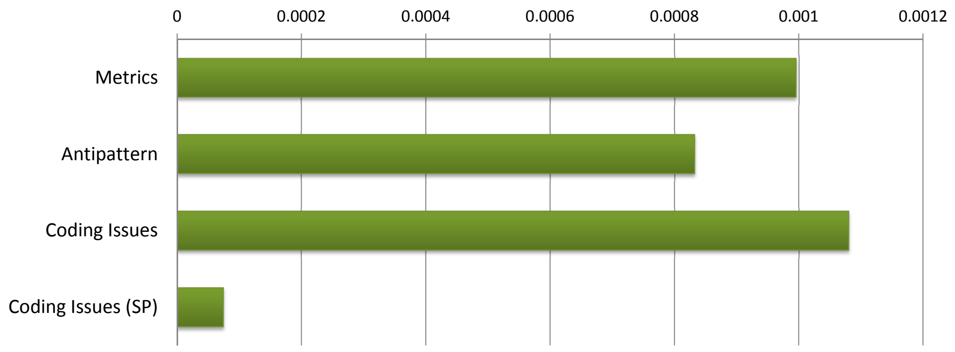


Figure 3.16. Average impact on maintainability of different refactoring types

they did not refactor the code, but just committed normal development patches. Our analysis told us that the number of newly introduced issues in the new code decreased. Indeed, developers admitted to us at the end of the project that they had learned a lot from performing a static analysis and from refactoring coding issues. They had learned how to avoid different types of potential coding issues. As a result they paid more attention to writing better code and avoiding new issues.

3.2.7 Threats to Validity

We made our observations based on hundreds of refactoring commits in six large-scale industrial systems. As in similar case studies which were not carried out in a controlled environment, there are many different threats which should be considered when we discuss the validity of our findings. Here, we give a brief overview of the most important ones.

Size of the sample set of refactoring commits investigated

The sample set was taken from a large-scale industrial environment compared to other studies, but it is still limited to the systems that we analyzed. With a larger sample set of refactorings we might have an even better basis for conclusions and a more precise view on refactorings. In the future, we intend to extend the sample set with an analysis of automatic refactorings as well.

Maintainability analysis relies only on the Columbus Quality Model and Java

The maintainability model is an important part of the analysis as it also determines what we regard as an *effect on maintainability* of refactorings. Currently, we rely on the ColumbusQM model with all of its advantages and disadvantages. On the positive side this model has been published, validated and reflects the opinion of developers [20]; however, we saw in the evaluation section that the model might overlook some aspects which would reflect some changes caused by refactorings. In particular, the model did not deal with some low priority local coding issues. The version of ColumbusQM used during the analysis relies on Java source code analysis. However, the same sensors could be applied to other object-orientated languages as well.

Refactoring suggestions and quality analysis tool used to evaluate their effect come from the same toolkit

The Columbus technology was used for both the refactoring suggestions and by the quality analysis tool. That is, the toolkit thinks that the changes made according to its own suggestions improve quality. This leads to the threat that the quality model used the same quality indicators as it suggested earlier as refactoring opportunities.

Limitations of the project

We claim that our experiment was carried out in an *in vivo* industrial context. However, this project might had unintentional effects on the study. For example, the budget for refactoring was not 'unlimited' and companies minimized the efforts that they spent on refactoring. Also, the actual state of a system, such as the size and quality of its test suite may influence the risk that a company would like to take during refactoring.

Limitations of the static analysis

We gave support to the developers in identifying coding issues with the help of a static analyzer. Naturally, this was a great help for them in identifying problematic code fragments, but it might have led the developers to just concentrate on the issues we reported. There is a risk here that by using other analyzers or by not using any at all, we might get different results.

3.3 Related Work

Since Opdyke introduced the term *refactoring* in his PhD dissertation [11] and Fowler published a catalog of refactoring 'bad smells' [12], many researchers have studied this technique to improve the maintainability of software systems. Just a few years later, Wake [29] published a workbook on the identification of 'smells', and indicated practices to recognize the most important ones and some possible ways to fix them by applying the appropriate refactoring techniques. Five years after the appearance of Fowler's book, Mens et al. [30] published a survey with over 100 related papers in the area of software refactoring.

There are several interesting topics studied today by researchers in which they examine refactoring techniques such as program comprehension [31], impact of refactoring on regression testing [32], and developers' opinions on refactoring tools [33], etc. Among the studies, there are some which investigate the positive or negative effects of refactorings on maintainability and software quality. However, there are only a few empirical studies, especially studies that were performed on large-scale industrial systems. Below, we will present an overview of research work related to our study.

3.3.1 Guidelines on how to apply refactoring methods

One reason why researchers study the relations between maintainability and refactoring is to guide developers on when and how to apply refactorings.

Sahraoui et al. [34] investigated the use of object-oriented metrics to detect potential design flaws and to suggest transformations that handle the identified problems. They relied on a quality estimation model to predict how these transformations improve the overall quality. By validating their technique on some classes of a C++ project, they showed that their approach could assist a designer/programmer by suggesting transformations.

A visualization approach was proposed by Simon et al. [35]. Their technique was based on source code metrics of classes and methods to help developers in identifying candidates for refactoring. They showed that metrics can support the identification of 'bad smells' and thus can be used as an effective and efficient way to support the decision of where to apply refactoring.

Tahvildari et al. [36, 37] investigated the use of object-oriented metrics to detect potential design flaws and suggested transformations for correcting them. They analyzed the impact of each refactoring on object-oriented metrics (complexity, cohesion and coupling).

Yu et al. [38] adopted a process-oriented modeling framework in order to analyze software qualities and to determine which software refactoring transformations are most appropriate. In a case study of a simple Fortran program, they showed that their approach was able to guide the refactoring towards high performance and code simplicity while implementing more functionalities.

Meananeatra [39] proposed the use of filtering conditions to help developers in refactoring identification and program element identification. They also proposed an approach to choose an optimal sequence of refactorings.

3.3.2 Refactoring and its effect on software defects

One way researchers attempt to assess the effects of refactorings on maintainability is to study its effects on software defects.

Ratzinger et al. [40] analyzed refactoring commits in five open-source systems written in Java and investigated via bug prediction models the relation between refactoring and software defects. They found an inverse correlation between refactorings and defects: if the number of refactoring edits increases in the preceding time period, the number of defects decreases.

Görg and Weißgerber [41, 42] detected incomplete refactorings in open-source projects and they found that incorrect refactoring edits can possibly cause bugs.

Later, Weißgerber et al. [43, 44] analyzed version histories of open-source systems and investigated whether refactorings are less error-prone than other changes. They found that in some phases of their projects a high ratio of refactorings was followed by a higher ratio of bugs. They found also phases where there was no increase at all.

3.3.3 Refactoring and its effect on code metrics

Some researchers assess the effects of refactorings on source code metrics.

Stroulia and Kapoor [45] presented their experiences with a system that followed a so-called refactoring-based development. They found that the size and coupling metrics of their system decreased after the refactoring process.

Du Bois and Mens [46] studied the effects of selected refactorings (ExtractMethod, EncapsulateField and PullUpMethod) on internal quality metrics such as the Number of Methods, Cyclomatic Complexity, Coupling Between Objects and Lack of Cohesion. Their approach is based on a formalism to describe the impact of refactorings on an AST representation of the source code, extended with cross-references. Later, Du Bois et al. [47] proposed refactoring guidelines for enhancing cohesion and coupling metrics and they got promising results by applying these transformations to an open-source project. The Ph.D. thesis of Du Bois was also about the effects of refactoring on internal and external program quality attributes [48].

3.3.4 Empirical studies on refactoring and its effects on software quality/maintainability

Empirical studies are those which are the closest to study. However, there are only a few large-scale *empirical studies* here.

Kataoka et al. [49] published a quantitative evaluation method to measure the maintainability enhancement effect of refactorings. They analyzed a single project and compared the coupling before and after the refactoring in order to evaluate the degree of maintainability enhancement. They found coupling metrics were effective for quantifying the refactoring effect and for choosing suitable refactorings. Their validation relied on a five-year-old C++ project of a single developer.

Moser et al. [50] studied the impact of refactoring on quality and productivity. They observed small teams working in similar, highly volatile domains and assessed the impact of refactoring in a 'close-to-industrial environment'. Their case study was about a Java project with 30 Java classes having 1,770 source code statements. Their findings indicated that refactoring not only increases software quality, but it also improves productivity.

Ratzinger et al. [51] observed the evolution of a 500 KLOC industrial Picture Archiving and Communication System (PACS) written in Java before and after a change coupling-driven refactoring period. They found that after the refactoring period, the code had low change coupling characteristics.

Demeyer [52] pointed out that refactoring is often blamed for performance reduction, especially in a C++ context, where the introduction of virtual function calls introduces an extra indirection via the virtual function table. He discovered, however, that C++ programs refactored this way often run faster than their non-refactored counterparts (e.g. compilers can optimize better on polymorphism than on simple if-else statements).

Stroggylos et al. [53] assessed a similar question to ours, namely whether refactoring improves software quality or not. They analyzed version control system logs (46 revision pairs) of open-source projects (Apache, Log4j, MySQL connector and Hibernate core) to detect changes marked as 'refactoring' and how software metrics were affected. They found that "*the expected and actual results often differ*", and although "*people use refactoring in order to improve the quality of their systems, the metrics indicate that this process often has the opposite results.*"

Alshayeb et al. [54] studied the effects of refactorings on different external quality attributes, namely adaptability, maintainability, understandability, reusability, and testability. They analyzed a system developed by students and two open-source systems with at most 60 classes and less than 12,000 lines of code. They investigated how C&K metrics had changed after applying refactoring techniques taken from Fowler's catalog and estimated their effects on the external quality attributes. They found that refactoring did not necessarily improve these quality attributes.

Geppert et al. [55] studied the refactoring of a large legacy business communication product where protocol logic in the registration domain was restructured. They investigated the strategies and effects of the refactoring effort on aspects of changeability and measured the outcomes. The findings of their case study revealed a significant decrease in customer-reported defects and in efforts needed to make changes.

Wilking et al. [56] investigated the effect of refactoring on maintainability and modifiability through an empirical evaluation carried out with 12 students. They tested maintainability by randomly inserting defects into the code and measuring the time needed to fix them; and they tested modifiability by adding new requirements and measuring the time and LOC metric needed to implement them. Their maintainability test displayed a slight advantage for refactoring, but regarding modifiability, the overhead of applying refactoring appeared to undermine other, positive effects.

Negara et al. [57] presented an empirical study that considered both manual and automated refactorings. They claimed that they analyzed 5,371 refactorings applied by students and professional programmers, but they did not provide further information on the systems in question.

A large-scale study, with similar findings, was carried out by Murphy-Hill et al. [58]. They applied refactorings taken from Fowler's catalog, and their data sets spanned over 13,000 developers with 240,000 tool-assisted refactorings of open-source applications. Our study is complementary, as we analyzed industrial systems instead of open-source ones and we mostly dealt with coding issues instead of refactorings from the catalog.

Kolb et al. [59] reported on the refactoring of a software component called Image Memory Handler (IMH), which was used in Ricoh's current products of office appliances. The component was implemented in C and it had about 200 KLOC. They evaluated software metrics of the product before and after a refactoring phase and found that the documentation and implementation of the component had been significantly improved, by the refactoring.

Kim et al. [60] reported on an empirical investigation of API-level refactorings. They studied API-level refactorings and bug fixes in three large open-source projects, totaling 26,523 revisions of evolution. They found an increase in the number of bug fixes after API-level refactorings, but the time taken to fix bugs was shorter after refactorings than before. In addition, they noticed that a large number of refactoring revisions included bug fixes at the same time or were related to later bug fix revisions. They also noticed frequent 'floss refactoring' mistakes (refactorings interleaved with behavior modifying edits).

In their study, Kim et al. [24] presented a study of refactoring challenges at Microsoft through a survey, interviews with professional software engineers and a quantitative analysis of version history data (of Windows 7). Among several interesting findings, their survey showed that the refactoring definition in practice seemed to differ from a rigorous academic definition of behavior-preserving program transformations and that developers perceived that refactoring involved substantial cost and risks.

3.3.5 Code smells and maintenance

Another topic close to ours is the effect of (fixing) code smells on maintenance problems.

Yamashita and Moonen [61] found that the effect of code smells on the overall maintainability is relatively small. They observed 6 developers working on 4 Java systems and only about 30% of the problems that they faced were related to files containing code smells.

In another study, Yamashita and Counsell [62] found that code smells were not good indicators for comparing the maintainability of systems differing greatly in size. They evaluated four medium-sized Java systems using code smells and compared the results against previous evaluations on the same systems based on expert judgment and C&K metrics.

In a recent study, Yamashita [63] assessed the capability of code smells to explain maintenance problems on a Java system which was examined for the presence of twelve code smells. They found a strong connection between the Interface Segregation Principle and maintenance problems.

Similarly, Hall et al. [64] found that some smells do indeed indicate fault-prone code in some circumstances, but that the effects that these smells had on faults were small. As they said, "*arbitrary refactoring is unlikely to significantly reduce fault-proneness and in some cases may increase fault-proneness*".

Ouni et al. [65] claimed that most of the existing refactoring approaches treated the code-smells to be fixed with the same importance; and they proposed a prioritization of code-smell correction tasks. Another prioritization approach was proposed by Guimaraes et al. [66] based on software metrics and architecture blueprints.

Khomh et al. [67] investigated the impact of antipatterns on classes in object-oriented systems and found that classes participating in antipatterns were more change and fault-prone than others.

Abbes et al. [68] investigated the effect of Blob and Spaghetti Code antipatterns on comprehension in 24 subjects and on three different systems developed in Java. Their results showed that the occurrence of one antipattern did not significantly make its comprehension harder, hence they recommend avoiding a combination of antipatterns via refactoring.

D'Ambros et al. [69] studied the relationship between software defects and a number of design flaws. They also found that, while some design flaws were more frequent, none of them could be considered more harmful in terms of software defects.

Chatzigeorgiou et al. [70] studied the evolution of code smells in JFlex and JFreeChart. They noticed that only a few code smells were removed from the projects and in most cases their disappearance was not the result of targeted refactoring activities, but rather a side-effect of adaptive maintenance.

Tsantalis et al. [71] examined refactorings in JUnit, HTTPCore, and HTTPClient. Among several interesting findings, they found that there was very little variation in the types of refactorings applied on test code, since most of the refactorings were about reorganization and the renaming of classes.

Recap

In contrast to the above-mentioned studies, in ours (1) we observed a *large amount of manual refactorings* (1,273 refactoring operations in 315 commits, counting also a commit with 454 operations); (2) we studied the effect of refactorings on maintainability in real-life, large-scale industrial systems with over 2.5 million total lines of code; (3) these commits fixed *different design flaws* including code smells, antipatterns and coding issues; (4) lastly, we applied a *probabilistic quality model* (ColumbusQM) which integrates different properties of the system like metrics, clones and coding issues. Our study was also carried out in a large-scale *in vivo* (industrial) environment.

3.4 Summary

The main goal of our experiments was to learn how developers refactor in an industrial context when they have the required resources (time and money) to do so. Our experiments were carried out on six large-scale industrial Java projects of different sizes and complexity. We studied refactorings on these systems, and we learned which kinds of issues developers fixed the most, and which of these refactorings were best according to certain system attributes. We investigated the effects of refactoring commits on source code maintainability using maintainability measurements based on the ColumbusQM maintainability model [20].

We found that developers tried to optimize their refactoring process to improve the quality of these systems and that they preferred to fix concrete coding issues rather than fix code smells indicated by metrics or automatic smell detectors. We claim that the outcome of one refactoring on the global maintainability of the software product is hard to predict; moreover, it might sometimes have a detrimental effect. However, a whole refactoring process can have a significant beneficial effect on the maintainability, which is measurable using a maintainability model. The reason for this is not only because the developers improve the maintainability of their software, but also because they will learn from the process and pay more attention to writing better maintainable new code.

*"If you want to make an apple pie from scratch,
you must first create the universe."*

— Carl Sagan

4 Challenges and Benefits of Automated Refactoring

To decrease software maintenance cost, software development companies generally use static source code analysis techniques. Static analysis tools are capable of finding potential bugs, anti-patterns, coding rule violations, and they can enforce coding style standards. Although there are several available static analyzers to choose from, they only support issue detection. The elimination of the issues is still performed manually by developers.

This is not a coincidence. Every developer knows that refactoring is not always easy. Developers need to identify the piece of code that should be improved and decide how to rewrite it. Furthermore, refactoring can also be risky; that is, the modified code needs to be re-tested, so developers can see if they broke something. Many IDEs offer a range of refactorings to support so-called automatic refactoring, but tools can really able to automatically refactor code smells are still under research.

Previously, we gained insights into how developers handle hand-written refactoring tasks and in what way it affected the maintainability of the source code. Based on the results of the manual refactoring phase of the Refactoring Project, here we will design a toolset that supports the automatic elimination of coding issues in Java. We will provide this refactoring tool as an aid to developers. We shall investigate the quality-changing effects of tool-assisted refactorings, while observing what kind of changes the usage of the tool causes in everyday work of developers.

4.1 An Automatic Refactoring Framework for Java

Tools which support automatic refactorings often assume that programmers already know how to refactor and they have a knowledge of the catalog of refactorings [12], but this is usually an unreasonable assumption. As Pinto et al. found in their study where they examine questions of refactoring tools on Stack Overflow, programmers are usually not able to identify refactoring opportunities, because of a lack of knowledge in refactoring, or a lack of understanding of the legacy code. They also claim that "*refactoring recommendations is one of the features that most of Stack Overflow users desire (13% of them)*" [33]. In another recent study, Fontana et al. compare the capabilities of refactoring tools to remove code smells and they found only one tool (*JDeodorant*) which was able to support code smell detection and then suggested which refactoring to apply to remove the detected smells [72]. Of course, most current tools lack this required feature to identify refactoring opportunities and to recommend problem-specific corrections which could even be automatically performed by the tools (or semi-automatically including some interactions with the developers).

In this section, we will introduce *FaultBuster*, an automatic code smell refactoring toolset which was designed with the following goals in mind:

- to assist developers in identifying code smells that should be refactored,
- to provide problem specific, automatic refactoring algorithms to correct the identified code smells,
- to seamlessly integrate easily with the development processes via plugins of popular IDEs (Eclipse, NetBeans, IntelliJ) so developers can initiate, review, and apply refactorings in their favorite environments.

4.1.1 Overview

Next, we will provide an overview of the structure of FaultBuster and give a short introduction to its features.

I Problem Context

The potential users of FaultBuster are members of a development team, potentially a developer or perhaps a quality specialist or a lead developer. Our aim was to help them by supporting '*continuous refactoring*' where developers prefer to make small improvements on a regular basis instead of just adding new features over a long period and restructuring the whole code base only when real problems arise.

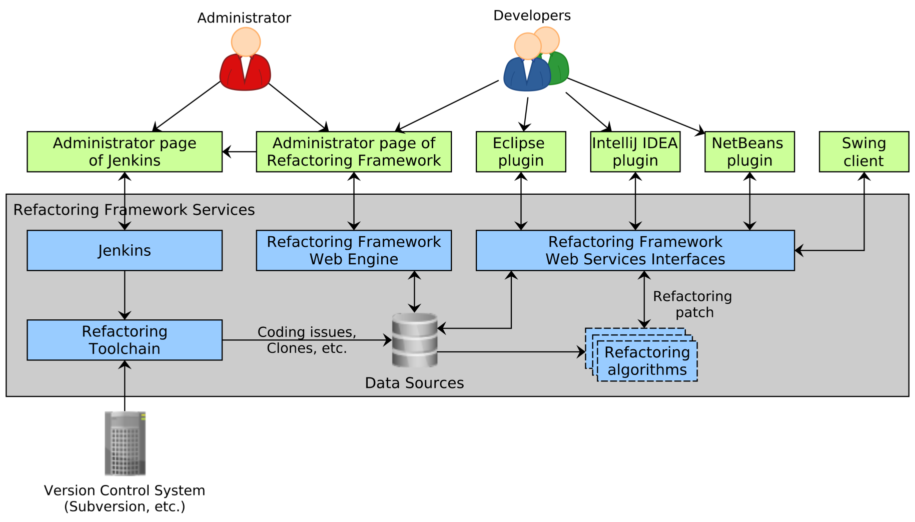

Administrator
Developers
Administrator page of Jenkins
Administrator page of Refactoring Framework
Eclipse plugin
IntelliJ IDEA plugin
NetBeans plugin
Swing client
Refactoring Framework Services
Jenkins
Refactoring Framework Web Engine
Refactoring Framework Web Services Interfaces
Refactoring patch
Refactoring Toolchain
Coding issues, Clones, etc.
Data Sources
Refactoring algorithms
Version Control System (Subversion, etc.)

Figure 4.1. Overview of the architecture of FaultBuster

For this purpose, FaultBuster was designed to periodically analyze the system in question, report problematic code fragments and provide support to fix the identified problems through automatic transformations (refactorings).

We notice here that most of the transformations supported by FaultBuster can be viewed as classic refactorings which do not alter the external behavior at all, just improve the internal structure of the source code. However, some of them may not fit into the classic definition. As mentioned in Section 2.3, some of the transformations fix potential bugs, which may indeed alter the behavior of the program, but not the behavior which the developer originally intended to implement. Hence, for some transformations, it means that we deviate slightly from the strict definition by allowing changes to the code that fix possible bugs, but do not alter the behavior of the code in any other way. For simplicity, we will call refactorings all the transformations of FaultBuster.

A sample refactoring of a coding rule violation (*Position Literals First In Comparisons*) can be seen in Listing 4.1. This code works perfectly until we invoke the 'printTest' method with a null reference that would result in a *Null Pointer Exception* (because of line 3). To avoid this, we have to compare the String literal with the variable, not the variable with the literal (see Listing 4.2). This and similar refactorings are simple, but one can avoid critical or even blocker errors by using them appropriately.

```
public class MyClass{
 public static void printTest(String a){
  if(a.equals("Test")) {
   System.out.println("This is a test!");
  }
 }
 public static void main(String[] args) {
  String a = "Test";
  printTest(a);
  a = null;
  printTest(a); // What happens?
 }
}
```

Listing 4.1. A sample 'Position Literals First In Comparisons' issue

```
public class MyClass{
 public static void printTest(String a){
  if("Test".equals(a)) {
   System.out.println("This is a test!");
  }
 }
 public static void main(String[] args) {
  String a = "Test";
  printTest(a);
  a = null;
  printTest(a); // What happens?
 }
}
```

Listing 4.2. Sample refactoring of Listing 4.1

II Architecture

Figure 4.1 provides an overview of the architecture of FaultBuster. The toolset consists of a core component called *Refactoring Framework*, three IDE plugins to communicate with the framework, and a standalone Java Swing client (desktop application).

Refactoring Framework This component is the heart of FaultBuster as its main task is to control the whole refactoring process. The framework handles the continuous quality measurements of the source code, the identification of critical parts from the viewpoint of refactoring, the restructuring of these parts, the measurement of quality improvement and the support of regression tests to verify the invariance after applying the refactorings.

In order to do so, the framework:

- Controls the analysis process and stores the results in a central database: it periodically checks out the source code of the system from a version control system (Subversion, CVS, Mercurial, Git), executes static analyzers (Java analyzer, rule checker, code smell detector, etc.) and uploads the results into the database.
- Provides an interface through web services to query the results of the analyses and to execute automatic refactoring algorithms for selected problems. After executing the algorithms on the server side, the framework generates a patch (diff file) and sends it back to the client.
- The analysis toolchain is controlled and can be configured through Jenkins.
- Has refactoring algorithms and the main settings of the framework are configurable through a web engine of the framework.

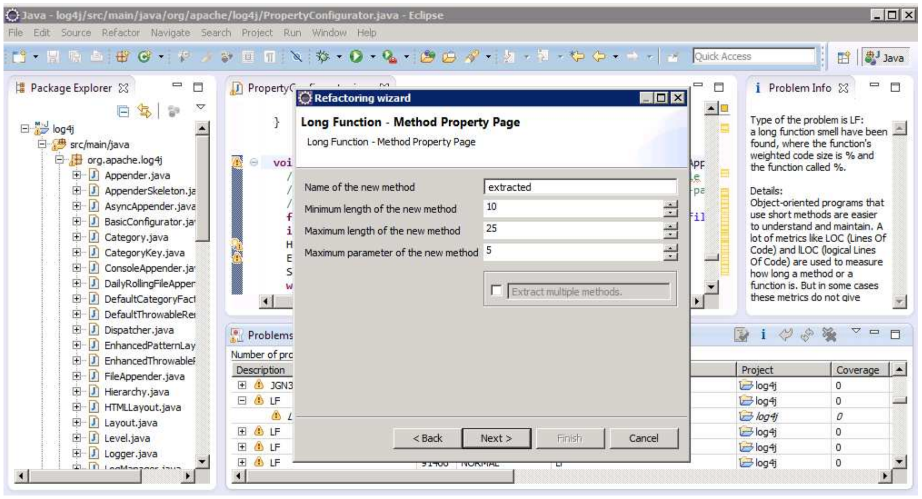
The framework was designed to be independent of the programming language. Although the current implementation only supports the Java language, it can support new languages and can be easily extended with additional refactorings. Several modules have been integrated in the implementation of the task: well-known tools supporting development procedures, like version control systems, project management tools, development environments, tools supporting tests, tools measuring and qualifying source code and automatic algorithms that implement refactorings.

IDE plugins We implemented plugins for today's most popular development environments for Java (Eclipse, NetBeans, IntelliJ IDEA) and integrated them with the framework. The goal of these plugins is to bring the refactoring activities to be implemented closer to the developers.

A plugin gets a list of problems in the source code from the framework, processes the results, and shows the critical points which detrimentally influence software quality to the user. A developer can then select one or more problems from this list and ask for solution(s) from the framework, which can then be visualized and (after confirmation) applied to the code by the plugin. Lastly, the developer can make some minor changes to it (e.g. commenting) and commit the final patch to the version control system.

When we designed the plugins, the main idea was to integrate the features offered by the framework as much as we could into the development environment. For example, we implemented standard features such as *context assist* in Eclipse. So it was a main concern that developers could work in the environment that they were used to and access the new features in a standard way.

Figure 4.2 shows a screenshot of the Eclipse plugin with our own wizard to set parameters of an algorithm which fixes a Long Function issue. Figure 4.3 shows the visualization of a patch after the execution of the algorithm.


Java - log4j/src/main/java/org/apache/log4j/PropertyConfigurator.java - Eclipse
Refactoring wizard
Package Explorer
Problem Info
< Back
Next >
Finish
Cancel

Long Function - Method Property Page

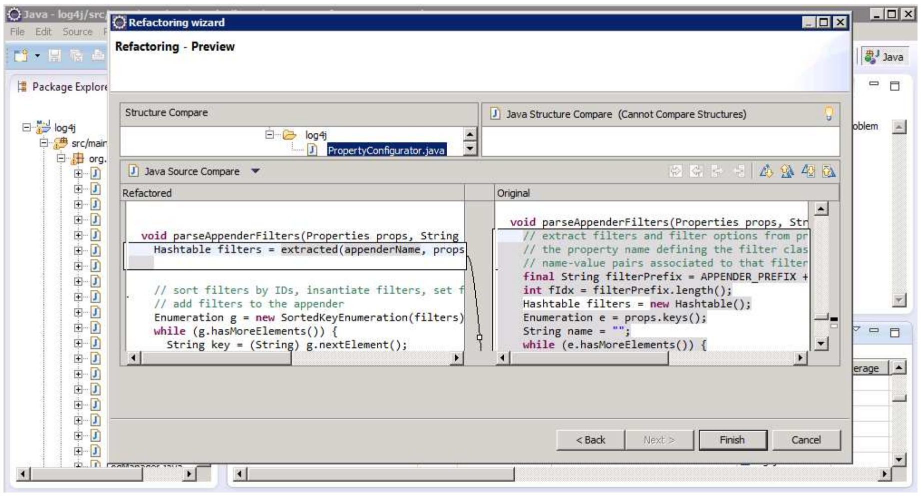
Long Function - Method Property Page

Name of the new method: extracted
Minimum length of the new method: 10
Maximum length of the new method: 25
Maximum parameter of the new method: 5
[ ] Extract multiple methods.

Type of the problem is LF:
a long function smell have been found, where the function's weighted code size is % and the function called %.

Details:
Object-oriented programs that use short methods are easier to understand and maintain. A lot of metrics like LOC (Lines Of Code) and lLOC (logical Lines Of Code) are used to measure how long a method or a function is. But in some cases these metrics do not give

Figure 4.2. Eclipse plugin – Screenshot of a Refactoring wizard with the configuration step of a refactoring algorithm for the Long Function smell



Refactored:

```
void parseAppenderFilters(Properties props, String
  Hashtable filters = extracted(appenderName, props

  // sort filters by IDs, insantiate filters, set f
  // add filters to the appender
  Enumeration g = new SortedKeyEnumeration(filters)
  while (g.hasMoreElements()) {
    String key = (String) g.nextElement();
```

Original:

```
void parseAppenderFilters(Properties props, Str
  // extract filters and filter options from pr
  // the property name defining the filter clas
  // name-value pairs associated to that filter
  final String filterPrefix = APPENDER_PREFIX +
  int fIdx = filterPrefix.length();
  Hashtable filters = new Hashtable();
  Enumeration e = props.keys();
  String name = "";
  while (e.hasMoreElements()) {
```

Figure 4.3. Eclipse plugin – Difference view of a patch after refactoring a Long Function smell

Standalone Swing Client Besides the IDE plugins, we implemented a standalone desktop client to communicate with the Refactoring Framework. At the beginning this client had only testing purposes, but later it implemented all the necessary features of the whole system, so it became a useful standalone tool of FaultBuster. The client is able to browse the reports on problematic code fragments in the system, select problems for refactoring, and invoke the refactoring algorithms, just like IDE plugins are able to do.

Administrator Pages The framework has two graphical user interfaces to configure its settings. Analysis tasks are controlled by Jenkins to periodically check out the source code and to execute the static analyzers. These tasks can be configured through the admin page of Jenkins. The rest of the framework can be configured through its own admin pages. Here, it is possible to configure user profiles and set some global parameters of the refactoring algorithms. In addition, this UI can be used to examine log messages and statistics of the framework.

Refactoring Algorithms We implemented automatic refactoring algorithms to fix common code smells and bad programming practices. The input of such an algorithm is a coding issue (with its kind and position information) and the abstract semantic graph (ASG) of the source code generated by the SourceMeter tool (see Section 4.1.2). The output of an algorithm is a patch (unified diff file) that will fix the selected problem.

FaultBuster implements algorithms that can solve 40 different kinds of coding issues (see Table 4.1) in Java. Most of these algorithms solve common programming problems like 'empty catch block', 'avoid print stack trace', 'boolean instantiation,' while some of them implement heuristics to fix bad code smells such as a long function, overly complex methods or code duplications.

Some algorithms can interact with the developer because they can be parametrized. For instance, in the case of a 'method naming convention' issue it is possible to ask the developer to give a new name for the badly named method. Still, many algorithms do not need extra information, e.g. the case of a 'local variable could be final' issue, the final keyword can be simply inserted into the declaration of the variable automatically.

It is also possible to select more occurrences of the same problem type and fix them in one go by invoking a so-called batch refactoring task. In this case, the Refactoring Framework will execute the refactoring algorithms and will generate a patch containing the fixes for all the selected issues. The only limit here is the boundary of the analysis, so it is possible to select problems from any classes, packages or projects, they just have to be analyzed beforehand by the framework.

III Extended Functionality

The core framework was implemented in Java as a Tomcat Web Application and it serves the IDE plugins through web services. Refactoring algorithms were implemented in Java using the Columbus ASG API of SourceMeter. Thanks to the Tomcat environment the toolset is platform-independent and it runs on all the supported platforms of SourceMeter (Windows and Linux).

Refactoring Wizards The client applications of FaultBuster are only soft clients, all functionality residing on the server side. This includes the refactoring algorithms as well. This allows the framework to extend its support of refactoring algorithms. Anytime an algorithm gets added or gets updated, just the server needs to be upgraded; and the clients will instantly support the new features.

Ticketing System Because of the server-client architecture of FaultBuster many users are able to connect to the server at the same time. To prevent concurrent modifications we introduced a state of the coding issues, which could be any of the following:

- **Open**: A detected coding issue. Available for fixing.
- **Under refactoring**: The coding issue is currently under refactoring by someone else.
- **Untested**: The coding issue has been refactored but it is still untested.
- **Completed**: The coding issue has been refactored and tested.
- **Committed**: The coding issue has been fixed and committed to the version control system.
- **Rejected**: It is not a real coding issue or the suggested fix was declined.

Table 4.1. Refactoring algorithms in FaultBuster

Local	AddEmptyString
	ArrayIsStoredDirectly
	AvoidReassigningParameters
	BooleanInstantiation
	EmptyIfStmt
	LocalVariableCouldBeFinal
	PositionLiteralsFirstInComparisons
	UnnecessaryConstructor
	UnnecessaryLocalBeforeReturn
	UnusedImports
	UnusedLocalVariable
	UnusedPrivateField
	UnusedPrivateMethod
	UselessParentheses
Naming	BooleanGetMethodName
	MethodNamingConventions
	MethodWithSameNameAsEnclosingClass
	ShortMethodName
	SuspiciousHashcodeMethodName
Interactive	AvoidInstanceofChecksInCatchClause
	AvoidPrintStackTrace
	AvoidThrowingNullPointerException
	AvoidThrowingRawExceptionTypes
	EmptyCatchBlock
	LooseCoupling
	PreserveStackTrace
	ReplaceHashtableWithMap
	ReplaceVectorWithList
	SimpleDateFormatNeedsLocale
	SwitchStmtsShouldHaveDefault
	UseArrayListInsteadOfVector
	UseEqualsToCompareStrings
	UseLocaleWithCaseConversions
	UseStringBufferForStringAppends
Heuristical	Clone Class (experimental)
	CyclomaticComplexity
	ExcessiveMethodLength
	LongFunction
	NPathComplexity
	TooManyMethods

The state of the coding issues were set by the client software automatically. This way, developers saw which coding issues were already under refactoring and they did not fix the same issues twice.

4.1.2 Under the Hood: Automating the Refactoring Process

Now that we have a better understanding how FaultBuster is designed we should take a look under the hood and investigate how it actually works underneath. Throughout the lifetime of the project we faced many challenges where we found interesting problems and solutions to these problems which we share in the followings.

Premise To perform refactoring operations, we created a mapping between the textual output of the static analyzer and the structural representation of the source code. This task required us to create an algorithm that takes as input a textual source code position (i.e. start and end line) and type information (i.e. for loop) of the problematic code segment and executes a search on the syntax tree to locate the related source code element in the tree. To make reverse searching possible, we use a *spatial database*. The database is created by transforming the source code into *geometric space*. Here, line numbers and column positions from the AST are used to define areas. These areas are used in *R-tree*s, where area based searching is possible.

General Process Before going into detail, we should have an overview of the general refactoring process. As Fowler defines it, refactoring is *"a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"* [12]. Based on this definition if we model the code as a graph – which every compiler does – a refactoring is a (behavior preserving) transformation on a graph. More specifically, it can be viewed as a transformation on the *abstract syntax tree (AST)*. Executing such transformation requires three components:

- An AST as a representation of the source code.
- A transformation algorithm.
- Starting points (AST nodes) called the *"origin"* where the transformation algorithm begins.

First, we parse the source code with a parser which builds an AST. Second, we create a transformation algorithm that will make modifications on the AST (i.e. pull up a method from one class to its ancestor). Next, we pick a node on the AST as the origin where the transformation algorithm will start working (i.e. selecting the method to pull up). After the transformation has been made, we get a modified (refactored) AST and the refactoring is complete.

Automated Process Because our main goal was to create automated refactorings, we extended the process with a few additional steps. These steps in the algorithm allow us to interact with developers and to make the transformations automatically. Our process works as follows:

Process 1

1. *We create an AST representation of the source code.*
2. *We carry out a static analysis on the source code to find problematic code parts, i.e. coding issues, rule violations and metric warnings. We list these issues as suggestions to help the user find candidates for refactoring.*
3. *The user selects one of the issues as the target of the refactoring process.*
4. *Based on the type of the issue chosen by the user, a refactoring algorithm is selected that is capable of fixing the given type of issue.*
5. *Based on the selected issue a proper origin node is chosen from the AST.*
6. *The algorithm makes the transformation and modifies the AST, while keeping track of what modifications it made. A modified AST is then created.*
7. *We generate source code from the modified AST.*
8. *The newly refactored source code is shown to the user where he or she can test the code and decide whether to accept or reject the refactoring.*

Next, we will present these steps in greater detail and discuss our results.

4.1.3 Process Details

I Building AST

To build an AST from the source code we will use the SourceMeter [23] tool. SourceMeter uses OpenJDK [73] as a backend to parse the code and build an *abstract semantic graph (ASG)*. ASG is an extended version of the AST with cross-edges and much more [25]. This additional information was crucial in the automation process. Because it allowed us to create flawless transformations which otherwise would not be possible. *Thus Step 1 of Process 1 is covered.*

II Finding Refactoring Suggestions

Choosing which part of the source code to refactor is quite hard. To improve the maintainability of the code, one can either start optimizing for metric values or try to eliminate anti-patterns by introducing design patterns into the code. Any of them might be a good solution. However, we will choose coding issues as the main target of our automated refactorings because of the experiences we gathered in Chapter 3. To identify coding issues we shall choose the well-known PMD static source code analyzer. It is a widely used tool among developers, especially for checking Java rule violations. Because all of the participating project members had a Java code base, it was the optimal choice to integrate it into the framework.

The output of PMD worked well in identifying coding issues and even in presenting some of these to developers as refactoring suggestions. Now let us examine the sample in Listing 4.3. In this simple example, PMD finds 9 rule violations (with default settings). It finds issues such as missing package declaration, missing comments, short variable names, magic numbers, and missing braces. Even in this simple sample of code, there are many issues that can be fixed with computer assistance.

To extract the issues we will use the XML output of PMD. This file contains a list of violations for each file with name, description, priority, and position information. An example violation is shown in Listing 4.4. This is one taken from the list of issues

```
1 public class Example {
2     public static int limiter(int x) {
3         if (x > 10)
4             return 10;
5         return x;
6     }
7 }
```

Listing 4.3. Example code

```
<file name="Example.java">
    ...
    <violation beginline="3" endline="4" begincolumn="3" endcolumn="13"
        rule="IfStmtsMustUseBraces" ruleset="Braces" class="Example"
       method="limiter" priority="3">
    Avoid using if statements without curly braces
    </violation>
    ...
</file>
```

Listing 4.4. PMD's XML report

we got as output after running PMD on Listing 4.3. It clearly states that we should use curly braces in the `if` statement in Line 3.

After presenting these kinds of issues to developers, they are usually able to select one for refactoring. *Thus Step 2 and Step 3 of Process 1 are covered.*

III Selecting the *Right* Transformation for the Job

After the user has selected an issue, the next step is to find the right transformation. Many researchers work on solutions to automate this process using machine learning techniques [74–76]. However, came up with a much simpler solution. We created several general refactoring transformations, i.e. for moving, adding, deleting, and swapping source code elements. We created a mapping between PMD violations and the transformations. For example, to fix the curly braces issue, we mapped it to an insertion transformation where a *block* element will be injected below the `if` statement (See the illustration in Figure 4.4). Each mapping defines different parameters based on the type of issue. And the transformation in the former example requires an `if` statement as a parameter. *Thus Step 4 of Process 1 is covered.*

IV Selecting a Proper Origin

After the user has selected an issue to fix and we choose the right transformation, the next step in the process is to pick an origin point on the AST, which we can give as a parameter to the transformation algorithm. In other words, one has to perform a search on the AST to find an element that matches both the description provided by the PMD report and the type of the parameter the transformation algorithm requires.

The report provides only a few key points for a violation, i.e. a begin line and

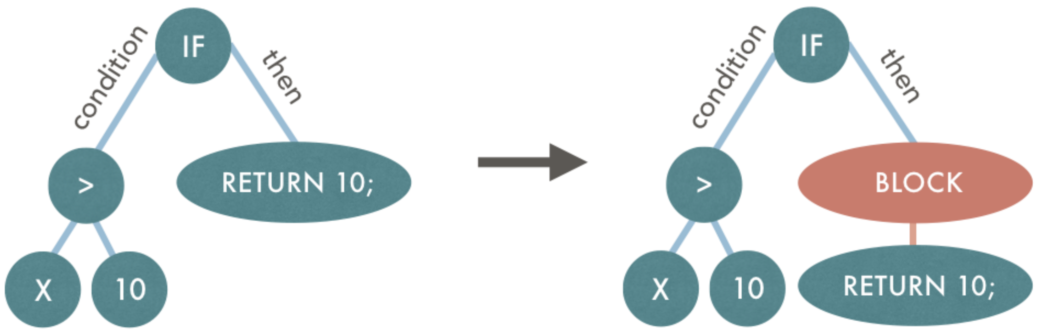


Figure 4.4. Simplified illustration of a refactoring on the AST of Listing 4.3.

```java
public class Example {
    public static int limiter(int x) {
        if (x > 10)
            return 10;
        return x;
    }
}
```

Listing 4.5. PMD highlight

column, end line and column, class, and method. Also, the source file is available in the `file` tag. If we look at the example in Listing 4.4 and the results in the example in Listing 4.3, we get the problematic code part *highlighted*. The highlighted part in Listing 4.5 shows the particular `if` statement that requires braces. Although this highlighted segment is a good visual aid for the developer to help find where the violation is, it is problematic for the computer to find nodes in the AST based on little more than position information.

One way to address the problem is to store position information for each source code element in the AST during the parsing process. Fortunately, SourceMeter does this already. Now that we know the positions on the AST, we can attempt to match the violation location information to the ones we have on the AST. It may come as no surprise that a simple equality match did not work. PMD and SourceMeter have different parsers and therefore it is quite unlikely they will have the same position information

```java
public class Example {
    public static int limiter(int x) {
        if (x > 10)
            return 10;
        return x;
    }
}
```

Listing 4.6. SourceMeter highlight

for each and every source code element. Listing 4.6 shows the position SourceMeter has identified for the `if` statement in question. Looking at both highlighted cases tells us that a simple approximation will not suffice to get a match. To handle the problem, we took a different direction which we call *reverse AST-search*.

Reverse AST-Search This notion was born when we decided to take a different direction and start looking at the source code elements, not as just data or nodes in a tree. In the text editor, they look like little areas or patches. Since they all have begin and end lines and columns, they can be viewed as coordinates on a map. This led us to the idea of transforming the source code into a *geometric space*.

We took line numbers and column positions from the AST and used them to define areas. These areas form rectangles where the corner points are the begin and end positions of the source elements. The rectangular areas are then used to build a *spatial database*, where area-based queries are possible.

i) Spatial Databases and R-trees A spatial database [77] is a database that is optimized to store and query data that represents objects defined in a geometric space. Common database systems use indexes to quickly look up values and the way that most databases index data is not optimal for spatial queries. Instead, spatial databases use a *spatial index* to speed up database operations. To create spatial index data, we decided to use *R-trees* [78].

An R-tree is a data structure where the key idea is to group together information based on spatial data and index these groups by using their minimum bounding rectangles[1]. Next, these groups are bound together at the next level of the tree by their minimum bounding rectangles, and so on. This way, a query cannot intersect any of the objects contained because all the objects within a bounding rectangle occur together. The input of a search is a rectangle called a *query box*. Every rectangle in a node (starting from the root node) is checked to see whether it overlaps with the search rectangle or not. If it does, the same thing happens with its corresponding child nodes. The search goes on recursively until all matching nodes get visited. Meanwhile, when a leaf node is found and it overlaps with the query box it is added to the result set.

R-tree applications cover a wide spectrum, ranging from spatial and temporal to image and video databases. In industry, it is used where multi-dimensional data needs to be indexed. For example, a common application is in digital maps where R-trees are used to link geographical coordinates to POIs [79].

ii) Building Spatial Index for the AST To create a spatial database for the source code, we used the *si* method in Algorithm 1. The method requires C, an AST element with position information. This might be a root node or a class node, say. When the algorithm commences, it creates an R-tree for storing the spatial index *(Alg. 1, Line 1)* and begins to traverse the descendants of the C element *(Alg. 1, Line 2)*. Note that every kind of traversal is acceptable since the position of the elements do not depend on each other.

For each source code element c, the algorithm takes into account their position P; namely, start line, start column, end line, end column *(Alg. 1, Line 3)*. Next, using these positions rectangles are created. To be precise, it creates one, two or three

[1]The "R" in R-tree is for rectangle.

Algorithm 1 Building Spatial Index for the AST.

```
Funct si(C)
Require: C is an AST element
 1: Let $\mathcal{I}$ be a new R-Tree
 2: for all $c \in descendants(C)$ do
 3:    Let $P_{start_{line}}, P_{start_{col}}, P_{end_{line}}, P_{end_{col}}$ be the position coordinates of $c$
 4:    if $P_{start_{line}} = P_{end_{line}}$ then
 5:       Add rectangle $\{P_{start_{line}}, P_{start_{col}}, P_{end_{line}}, P_{end_{col}}\}$ to $\mathcal{I}(c)$
 6:    else
 7:       Add rectangle $\{P_{start_{line}}, P_{start_{col}}, P_{start_{line}}, \infty\}$ to $\mathcal{I}(c)$
 8:       Add rectangle $\{P_{end_{line}}, 0, P_{end_{line}}, P_{end_{col}}\}$ to $\mathcal{I}(c)$
 9:       if $P_{end_{line}} - P_{start_{line}} > 1$ then
10:          Add rectangle $\{P_{start_{line}} + 1, 0, P_{end_{line}} - 1, \infty\}$ to $\mathcal{I}(c)$
11:       end if
12:    end if
13: end for
14: return $\mathcal{I}$
```

rectangles, depending on the length of the current source element. If the element position is confined to a single line, one rectangle is created *(Alg. 1, Line 5)*, which is a line on the 2D plane. In the case of multiple lines, we have a multiline element, which is an element that starts at a line in a column, and it ends in another line in a column. All positions between these two positions are part of the element. For example, in Listing 4.6 the `if` statement is a two-line element and the highlight indicates the positions belonging to the statement. To handle this, we create two rectangles. The first line has no end column *(Alg. 1, Line 7)*, and the second begins at zero *(Alg. 1, Line 8)*. If there more lines between them, we create a rectangle that covers all the space between the two lines *(Alg. 1, Line 10)*.

Each time a rectangle is created it is added to the R-Tree with a binding to the AST element. This way when the spatial query function starts running, we will get AST elements instead of rectangles. Once all the descendant source code elements get visited, we can return the resulting R-tree, and the spatial index is ready.

When we tested the algorithm in the example in Listing 4.3, we got the search space shown in Listing 4.7. Note that every node that has multiple lines are separated into more rectangles, like the `if` statement.

iii) The Search Algorithm Once we have built our spatial index, we can use it to locate a node in the AST based on position information. We created a method that uses inputs such as the ASG from SourceMeter, the parameter type of the transformation, and the violation position from the PMD report in order to search the geometric space. The **R**everse **AST**-search **A**lgorithm (*rasta* for short) is listed in Algorithm 2 below.

The purpose of the algorithm is to find the source code element that is highlighted in the PMD report. The function begins by creating a list of the source code element candidates. Next, it builds a spatial index with the C AST parameter *(Alg. 2, Line 2)*. The newly constructed index is used in the next step to query the candidates. As mentioned earlier, the spatial database requires a rectangle, called the query box as

```
Rect (1, 1), (1, INF) = Class
Rect (2, 0), (6, INF) = Class
Rect (7, 0), (7, 2) = Class
Rect (2, 2), (2, INF) = Method
Rect (3, 0), (5, INF) = Method
Rect (6, 0), (6, 3) = Method
Rect (2, 16), (2, 19) = PrimitiveTypeExpression
Rect (2, 28), (2, 33) = Parameter
Rect (2, 28), (2, 31) = PrimitiveTypeExpression
Rect (2, 35), (2, INF) = Block
Rect (3, 0), (5, INF) = Block
Rect (6, 0), (6, 3) = Block
Rect (3, 3), (3, INF) = If
Rect (4, 0), (4, 14) = If
Rect (3, 6), (3, 14) = ParenthesizedExpression
Rect (3, 7), (3, 13) = InfixExpression
Rect (3, 7), (3, 8) = Identifier
Rect (3, 11), (3, 13) = IntegerLiteral
Rect (4, 4), (4, 14) = Return
Rect (4, 11), (4, 13) = IntegerLiteral
Rect (5, 3), (5, 12) = Return
Rect (5, 10), (5, 11) = Identifier
```

Listing 4.7. Search space for the example in Listing 4.3 with the rectangles and the type of their referred source code element

Algorithm 2 The reverse AST-search algorithm.

```
Funct rasta($C, P, t$)
Require: $C$ is an AST element
Require: $P$ is a position
Require: $t$ is a type
1: Set the candidate list $\mathcal{R} := \{\}$
2: Compute $si(C)$
3: Let $S$ be the set of all AST nodes whose have intersecting rectangles with $P$
4: for all $s \in \mathcal{S}$ do
5:     if $type(s)$ is $t$ then
6:         Store $s$ in $\mathcal{R}$
7:     end if
8: end for
9: return $\mathcal{R}$
```

a search parameter. The query box in our case is the "highlight" from the PMD's output. We use this box to ask the R-tree which previously added rectangles intersect with the parameter. The R-tree returns with a set of AST nodes whose rectangles satisfied the query *(Alg. 2, Line 3)*. As an example, Listing 4.7 shows which source code elements (highlighted lines) remain after the query has been performed with the `Rect(3,3)(4,13)` box as the parameter.

Even with a small sample like Listing 4.7, the resulting set of the query can be quite big. To narrow the result set we use the third parameter, namely the type of the input parameter of the refactoring transformation, to filter the results *(Alg. 2, Line 5)*. The filtering is achieved by going through the result set and by inserting only those source code elements onto the candidate list whose type matches (actually, whose type is compatible with) the input type. After the filtering, the candidate list is returned as the result of the function.

Going on the example in Listing 4.7, we have to filter the result set with the input parameter type of the refactoring transformation. In Section III we identified this source code element type as an `if` statement. Even from a quick glance, we can see that there are two rectangles where their type is an `if` statement. Since both of the two rectangles refer to the same `if` element, the algorithm terminates. We have found an origin where the refactoring transformation can begin its operation.

iv) Heuristics As we saw previously, the best case scenario is when the algorithm ends up having only a single element in the list of candidates. This happens when the result set has only one source code element with the type of parameter. In this case, it is evident which element was highlighted as the source of the violation. Nevertheless, there are times when the candidate list has multiple source code elements of the same type. In such cases, we have to select the proper element as the origin, otherwise the refactoring will be executed wrongly.

To remove ambiguity we decided to use a "distance" measure to find the best candidate. We defined the distance as the number of characters between the position of the source code element and the highlight. More specifically, on one hand, it is a metric of the number of characters between the start position (line and column) of the source code element and the start position (line and column) of the highlight. On the other hand, it is the number of characters between the end position (line and column) of the element and the end position (line and column) of the highlight. The distance is just the sum of the two. To calculate this value we use the original source file where the exact number of characters could be measured.

This method allows us to select the proper source code element from the list of candidates in almost every case. Still, there is a certain mathematical probability that the calculated distances will be equal for all candidates. However, because the chance of this event is astronomically small (it never even happened once in our exhaustive testing period, see Section 4.2), we chose to notify the user with a message that the refactoring failed because of ambiguity.

v) Alternative Method The algorithm above uses two-dimensional information to back-propagate code smells to AST elements and handles code as lines and columns. However, source code can be viewed as a linear sequence of characters as well. Here, a simple one-dimensional data structure and interval operations could replace the fairly complicated two-dimensional approach. Despite this mechanism being seemingly simpler, it would require different input data. Both the given inputs – the AST and PMD – work with two-dimensional data. This would mean that we would either require the source file as input or the AST and PMD must provide one-dimensional data.

- The latter requirement would be needed from both tools to replace position information with char-sequence index or store it as additional data. This new data would cause an increase both in the processing time and storage space for the tools for information that probably no one else will ever use. Still, if char-sequence index data is available as input, the one-dimensional approach is preferable.

- The former requirement would introduce an additional parameter to the algorithm; namely, the original source file. In the case where source code is given it is possible to handle the text as one-dimensional data and map the source code elements to char-sequence indexes. However, this approach has several drawbacks. First, every search would need to read the source file, which is an i/o intensive task. Next, the mapping of two dimensional data to character sequence indexes would have to consider whitespace. For example, when reading a tab character from the file, the algorithm has to know it is 2, 4, 8 (or other) characters long. Both parsers could have mixed tab size settings, which would make the mapping difficult. This would also affect the two-dimensional approach, but since a line just contains only a few tabs it easier to match the source code elements in a line than in the entire file.

The reverse AST-search algorithm works only with an AST and PMD's report as input. These tools and inputs are treated as third-party from the algorithm's point-of-view. Since these inputs contain two-dimensional data and source code is not available, the one-dimensional approach would not suffice. Nevertheless, from the point-of-view of the refactoring process, the source file is given; but we still choose the two-dimensional approach because it provides more accurate matches in real-life scenarios.

vi) Summary The reverse AST-search algorithm enabled us to select the proper origin, which is a source code element in the AST that is the input of the refactoring transformation later on. *Thus Step 5 of Process 1 is covered.*

```java
public class Example {
    public static int limiter(int x) {
        if (x > 10) {
            return 10;
        }
        return x;
    }
}
```

Listing 4.8. Refactored code

V Executing the Refactoring

Now that we have covered each preceding step, we have all the components and we are ready to perform the refactoring. As mentioned earlier in Section III, the refactoring algorithms are defined as smaller, multiple generic transformations. The type of the PMD violation determines which transformation(s) will be executed. There are some complex cases where a simple transformation will not suffice and fixing them will require multiple operations.

In order to fix the *missing curly braces* (the issue in Listing 4.3), the transformation inserts a *block* statement into the *then* clause of the `if` statement and rewires every former member of the *then* clause to make a member of the *block* statement. See Figure 4.4 as an illustration of a process like this.

Once the refactoring transformation completes its operations, a new, modified, issue-free AST is created. *Thus Step 6 of Process 1 is covered.*

VI Generating Source Code and Creating Diffs

In the previous step, we completed the refactoring at the model level. Even though the refactoring process came to an end, there is one more thing to do. Because our main goal is to assist developers, the next task is to translate the AST back to source code where they can readily interpret the changes.

Generating Source Code The source code generation is realized by systematically going through the AST and writing code to a text file according to the underlining source code element. For example, if we start at a file, if there is a package declaration we write the package keyword following the name of the package and a semicolon to close the statement. This is followed by import statements and so on. The generation goes on until every source code element is visited and the code is fully reconstructed from the AST.

In the case of the example in Listing 4.3, after the refactoring transformation we get the code shown in Listing 4.8. As expected, both curly braces appeared and therefore the `return` statement got a *block* around it. As a consequence, the PMD rule violation got fixed, and our code maintainability improved. *Thus Step 7 of Process 1 is covered.*

Keeping Track of Modifications Previously, we showed how to fully reconstruct the code from the AST. However, generating the whole code base is unnecessary. It is sufficient to recreate only those code segments where the changes occurred. There are, however, multiple ways to reduce the amount of generated code. An easy solution would be to create only those files which were affected by the refactoring. Our only concern, in this case, was that our code generation cannot reproduce exactly (100%) equivalent source code. This happens because though SourceMeter stores source code element positions and even comments, it does not store data concerning whitespaces and indentation. Despite this, we created the source code generator in such a way that it "pretty prints" the code, but what is considered "pretty" is subjective. For example, in Listing 4.8 the beginning bracket is positioned after the method declaration, but someone may prefer it to be on the next line. On one hand, it is possible to make this configurable. On the other hand, there are other remaining issues, such as whether there are two spaces between the `public` and the `static` keywords or whether they should be written in separate lines.

To reduce the former anomalies, we sought to minimize generation even within files themselves. Our approach keeps track of which nodes are modified and at what level when the refactoring operation is running. We mark those nodes that get, for instance, inserted, deleted, or swapped. Furthermore, we mark those nodes which we visited during the operation but did not modify them. For example, in Listing 4.8 we put a *block* statement into the *then* clause of the `if` statement and rewired the former content (the `return`) of the *then* clause as a statement of the *block* node. We marked this *block* statement as inserted, and the `return` as unmodified. The latter was required because marking a node is a recursive operation which will mark the entire subtree of that node as well. By marking the `return` statement as unmodified we will leave this subtree untouched and bring about efficiency benefits.

Keeping track of modifications allowed the generator to only modify those places where it was necessary. Only the new or modified source code elements get generated, and every other part of the source code gets copied from the original source file. In the example, this works in the following way. The generator starts traversing the refactored AST from root to bottom, in a preorder strategy. When it finds unmodified nodes, it just copies the source code from the original file into the refactored file. This is based on the position information stored in the AST. This goes on until it finds a modified node in this case, an inserted *block* statement. Next, it generates the *block* statement. More precisely, it generates only the starting bracket, because there are still unvisited descendants of the *block* node in the AST. When the traversal goes to the next child, it finds an unmodified node again. It does so the same way as before, it copies the code from the original source code, but this time it will insert the copied code with an increased indent because the generator keeps track of the fact that we are now in a *block* statement. After every descendant has been visited and copied, we return to the leave-visit for the *block* statement. The generator inserts the closing bracket into the right place, and a visit continues. Since every other node is unmodified, everything else is copied, and the generation is complete.

Generating just the required code parts created nearly the same code as the original, with most indentation and whitespaces in the right place. This was an important request from developers because interviews showed that they did not want to bother with fixing the indentation (see Section 4.3.3).

Creating Diffs As soon as the generation process ended, it became possible to present the refactored code to the developers. However, reviewing entire unannotated files is not a welcomed idea by developers. Since this is the output of an automated pro-

```
--- Example.java (original)
+++ Example.java (refactored)
@@ -1,7 +1,8 @@
 public class Example {
     public static int limiter(int x) {
-        if (x > 10)
+        if (x > 10) {
             return 10;
+        }
         return x;
     }
 }
```

Listing 4.9. Output diff file

cess, users would probably like to check what changes the automation process will apply.

To meet the need of the users, we will only show their a patch (unified diff file) as output. This patch file contains the differences between the refactored source file and the original one. This enables the developer to review what changes the automation process made on the code. Besides this, it allows the user to make a decision at the end of the process of whether to accept or reject the suggested refactoring. If it is the former, the user can apply the diff on the original source code, and this will transform it into the refactored code. *Thus Step 8 of Process 1 is covered.*

For example, Listing 4.9 lists the diff file for the refactoring of the example in Listing 4.3. Note that this shows which lines are marked for deletion (starting with a "-") and which ones are marked as added (starting with a "+").

4.1.4 Discussion

I Performance

Our refactoring tool was implemented in Java. One of the requirements for our tool was for it to be responsive from a user perspective. This required that we to optimize each step for speed. The most important optimization we preformed was with the Reverse AST-search algorithm.

Building the spatial database for the whole system was an unnecessary overhead. To reduce the search space, we built the spatial index based on the issue the user had selected. The rule violation has information about which source file it is in. Only these file elements were added to the R-tree and it greatly reduced the search space. As a comparison, building the entire search space on a PC[2] for the log4j[3] project took 221 ms and used 46 MB of memory, while building only one file took 53 ms and memory used was less than a kilobyte.

Further optimizations helped to speed up the process as well. For example, the the filtering step moved a few steps ahead in the order of the execution of the Reverse AST-search algorithm. Filtering was applied while building the search space. Only those AST nodes were added to the R-tree where the type of node matched the type

[2]Intel i7 3.40 GHz with 8 GB ram.

[3]`http://logging.apache.org/log4j/1.2`

of refactoring transformation parameter. This way, executing a single search operation takes less than a millisecond.

The above improvements besides some other tweaks made our tool quick and this appealed to developers. We did not make detailed measurements of the tool's performance in our studies, but in general the tool performed well.

II Threats to Validity

We have identified a few validity threats that might affect the the internal and external validity of our results. Here, we discuss the validity of our findings.

Usage of Java We have only considered Java as the target of our actions. Some of the other languages may require different approach. Nevertheless, our process is readily adaptable to most text-based programming languages.

Application of a Third-Party Tool We provided support to the developers in identifying coding issues with a third-party static analyzer, namely PMD. Naturally, this was a great help in identifying problematic code fragments, but it might have introduced many unnecessary steps during the refactoring process. There is a risk here that by using other analyzers or by using our own, we might skip the AST-search part. For example, if we were to develop our own issue finder tool (see Section 5), we could directly report the AST node where the problem is located. However, our process makes our refactoring tool independent of a single third-party static analyzer. The way, it is constructed makes it capable of switching to another analyzer with only minor modification.

4.2 Evaluating the Connection between Automatic Refactorings and Maintainability

By definition, the intention of developers with refactoring is to improve comprehensibility, maintainability, hence the overall quality of the source code. However, there is a disagreement in the literature as to whether it truly improves quality or not. In Section 3.2 we investigated this phenomenon and found that in most of the cases refactoring improved the overall maintainability of the systems in most cases.

Here, we investigate how automatic refactorings change maintainability. We asked developers to do refactorings on their systems with the previously introduced Refactoring Framework called Faultbuster. To study this situation, we address the following questions:

- Does automatic refactoring increase the overall maintainability of a software system?
- What is the impact of different automatic refactoring types on software maintainability?
- What is the impact of different automatic refactoring types on the code metrics used in the maintainability model?

4.2.1 Methodology

We gathered our research data in a similar way how we did with the manual refactorings (see Section 3.2.1). However, there are some differences. Figure 4.5 provides a brief overview of the automatic refactoring phase of the project.

We started by identifying possible targets for refactoring by analyzing their systems (*Step 1*). During the measurement period, the framework supported the refactoring of 21 different coding issues, so the companies were asked to fix issues from this list[4].

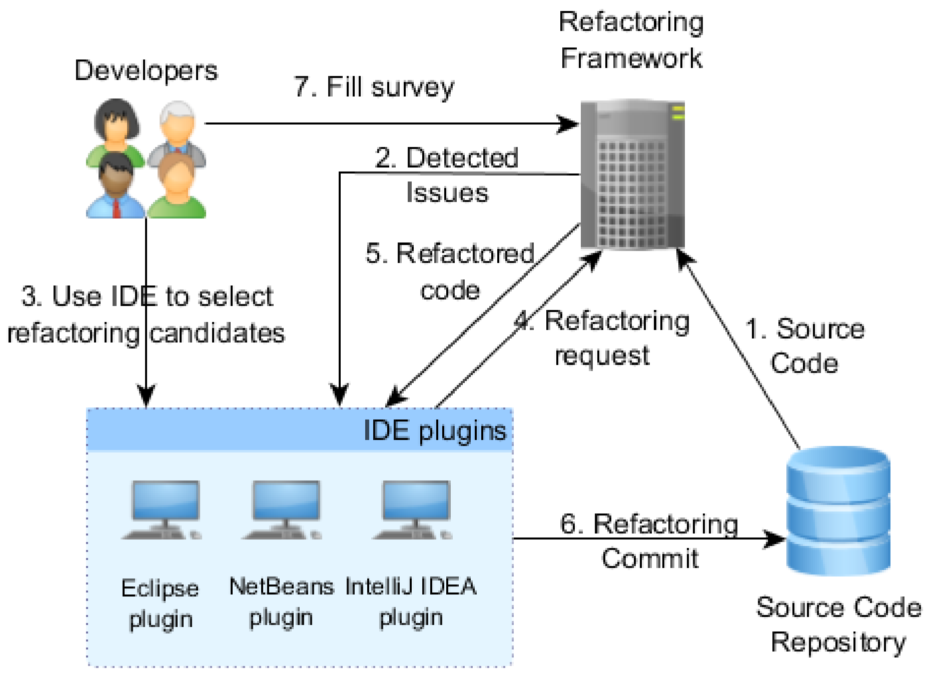


Figure 4.5. Overview of the refactoring process

It was a project requirement for the developers to refactor their own code, hence improve its maintainability, but – just like before – they were free to select how they went through it. So was the choice of the developers as to what kind of coding issues they should fix with the help of the framework. The process of fixing a coding issue was to apply the appropriate refactoring operation offered by the framework through the developers' standard development environment. The FaultBuster plugins were able to load from the framework and to present in the IDEs all the detected coding issues that the developers were able to refactor with the help of the tool (*Step 2*). To apply a refactoring operation, the developers selected an issue from the code and called the refactoring service through the IDE plugins (*Step 3*). After gathering all the required information from the plugin, a request was sent to the Refactoring Framework to perform the refactoring step on the code (*Step 4*).

After a refactoring operation was carried out on the ASG, the framework re-generated the transformed source code. The generated patch was sent back to the IDE plugins in which the developers were able to preview the modifications (with the help of the built-in diff viewers of the IDEs) before they applied it (*Step 5*). Of course, the developers had the opportunity to discard the changes if they were not satisfied

[4]After developers started using the tool regularly, they asked us to support more and more coding issues, and in the end, FaultBuster supported 40 issue types (as described in Section 4.1).

with the resulting refactored code. In that case, no changes were made in the code base. Note, that the framework allowed fixing multiple issues at once, but this type of batch refactorings had to be of the same type (for example, the framework was able to fix hundreds of *PositionLiteralsFirstInComparisons* issues in one patch, but mixing issues was not supported). If the presented patch got accepted, the developers applied them on the current code base and performed a commit to upload the refactored code into the source code repository (*Step 6*).

Besides applying concrete refactorings, the project required that the companies fill out a survey (which we collected with the IDE plugins) after each refactoring and give an explanation on what and why they refactored during their work session (*Step 7*). The survey contained revision-related information as well, so we could relate one refactoring to a revision in the version control systems.

After this refactoring phase, we analyzed the marked revisions and investigated the change in the maintainability of the systems caused by refactoring commits. Figure 4.6 gives an overview of this analysis. As before, it was not a requirement from the developers that they commit only refactorings to the version control system, or that they create a separate branch for this purpose. What was a requirement though is that a commit containing refactoring operations could not contain other code modifications. Hence, for each system we could identify the revisions (r_{t_1}, ..., r_{t_i}, ..., r_{t_n}) that were reported in the surveys collected by the Refactoring Framework after refactoring commits, and we analyzed all these revisions with the revisions prior to them. As a result, we chose for a system the set of revisions $r_{t_1-1}, r_{t_1}, ..., r_{t_i-1}, r_{t_i}, ..., r_{t_n-1}, r_{t_n}$ where r_{t_i} is a refactoring commit and r_{t_i-1} is the revision prior to this commit.

We performed the analysis of these revisions with *QualityGate SourceAudit* described in Section 2.2. If a commit contained more than one refactoring of the same type – because the framework supported a way of bulk fixing the issues – we calculated the average amount of maintainability changes of a refactoring type by dividing the maintainability change brought about by the whole commit by the number of actual refactorings contained in it. Everywhere in this chapter, if we deal with maintainability change caused by a refactoring type, we use the average values of these changes. This is of course a small threat to validity, as there is no guarantee that all the fixed issues in various places in the code will affect the maintainability in the same way. However, all the refactorings were performed by an automatic framework which resulted in very similar (though due to the possible manual steps not necessarily the same) fixes for the issues, therefore the chances that these refactorings had different impacts on maintainability is minimal. Besides analyzing the maintainability of the above revisions, we gathered data from the version control system as well, such as diffs and log messages.

4.2.2 Results

Following the process described in Section 4.2.1, the companies performed a large number of automatic refactorings on their own code base using the Refactoring Framework developed within the project. They uploaded almost 4,000 refactorings to the source code repositories with more than 1,000 commits altogether (see Table 4.2). We analyzed 4 projects of 4 different companies and collected data according to the method depicted in Figure 4.6. That is, we calculated all the maintainability changes brought about by refactoring commits and aggregated the data at various levels. As the maintainability model used takes the number of coding issues into account (see Figure 2.2)

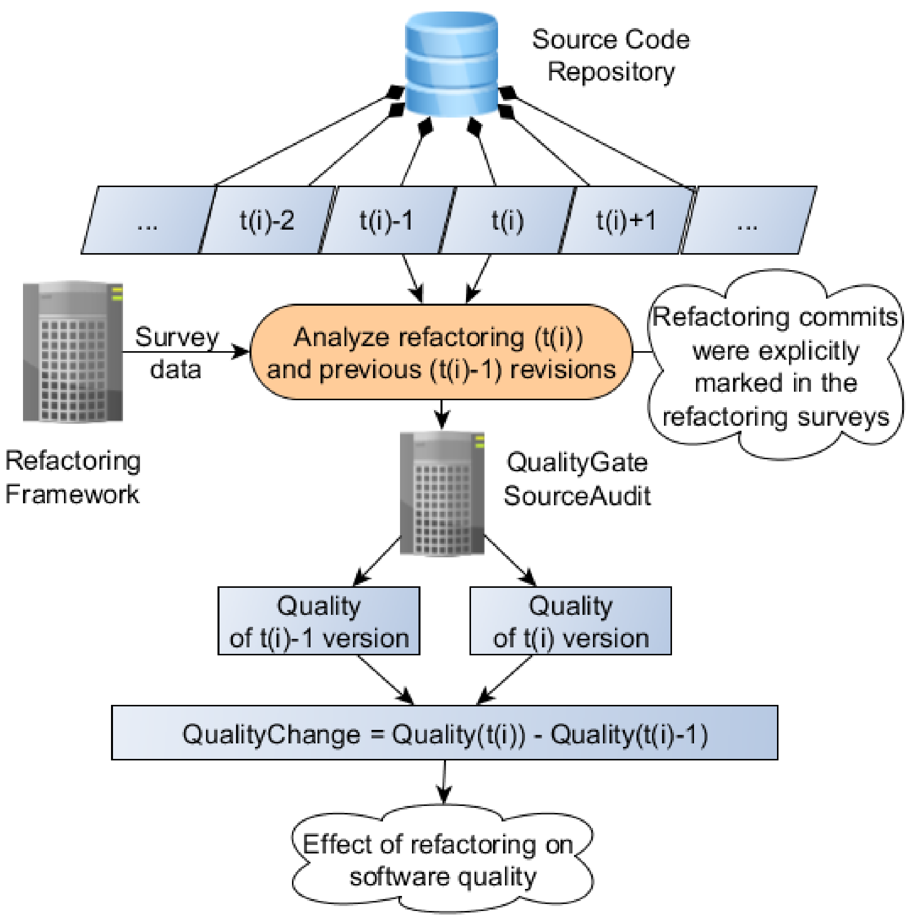

Source Code Repository
...
t(i)-2
t(i)-1
t(i)
t(i)+1
...
Survey data
Analyze refactoring (t(i)) and previous (t(i)-1) revisions
Refactoring commits were explicitly marked in the refactoring surveys
Refactoring Framework
QualityGate SourceAudit
Quality of t(i)-1 version
Quality of t(i) version
QualityChange = Quality(t(i)) - Quality(t(i)-1)
Effect of refactoring on software quality

Figure 4.6. Overview of the analysis process

and the Refactoring Framework supports the refactoring of such coding issues, one might induce that it is trivial that upon refactoring the maintainability of code will increase. Nonetheless, fixing an issue might cause code changes that lead to e.g. changes in code clones, new coding issues, or changes in metrics. So it is far from trivial to predict the complex effect of refactorings on code maintainability. What is more, the task that the Refactoring Framework includes some semi-automatic steps, thus developers are able to configure the same refactoring operations somewhat differently. For example, fixing an *EmptyCatchBlock* issue begins with three options, namely *a*) add logger; *b*) use *printStackTrace()*; and *c*) leave a comment, where selecting one option may introduce new options (e.g.: comment text and logger kind).

Table 4.2. Selected projects

Company	Project	kLOC	Analyzed revisions	Refactoring commits	Refactorings
Company I	Project A	1,119	299	217	1,444
Company II	Project B	962	868	449	1,306
Company III	Project C	206	1,313	316	404
Company IV	Project D	780	200	66	682
	Total	**3,067**	**2,680**	**1,048**	**3,836**

First, we show how the sum of all refactoring related maintainability changes turned out to be for the various projects. Next, we will dig a bit deeper into the data to find out what the average impact of the individual refactoring types is on software maintainability. Then, we will go even one step further to explore the effect of refactoring types on software maintainability at the level of code metrics.

I Effect of Automatic Code Refactoring on Software Maintainability

The data presented in Table 4.3 can bring us closer to find the answer to our first question. The rows of the table contain an overview of the quality properties of 4 systems of 4 companies participating in the automatic refactoring phase of the project. The Coding Issues column shows the overall number of issues that were fixed by (semi)automatic refactoring in a particular system. The Maintainability Before and Maintainability After columns contain the maintainability values of the systems before and after the automatic refactoring phase, calculated as described in Section 2.2 (0 is the worst value, 10 is the best). The total improvement (*Total Impr.*) column reflects the difference between the maintainability values before and after the automatic refactoring phase, hence if this value is negative, the overall maintainability of the system has decreased during the refactoring phase, while a positive difference means a maintainability improvement. Note that the companies were allowed to perform any kind of code modifications during this phase, not just refactorings, so this value represents the combined effect of all the code changes on the system maintainability. The next column, refactoring improvement (*Ref. Impr.*), is the code improvement achieved solely by refactorings. This is calculated as the sum of the maintainability changes caused by commits containing refactoring operations only (i.e. sum of the maintainability differences between refactoring and prior commits). The last column (*Ref. Impr. %*) is simply the ratio of the refactoring and total improvement values. Its intuitive meaning would be the amount of code improvement caused by refactoring commits. Bigger values than 100% may occur, which mean that the effect of refactorings is higher than the overall effect of all the code changes; however, this effect might be positive and negative as well.

In total, in 3 out of 4 cases the overall system maintainability values increased during the refactoring phase. In these 3 projects, the net effect of refactoring commits was also positive, meaning that the automatic refactoring phase increased the maintainability of the code. The only exception is *Project A*, where both the overall system maintainability and the net effect of refactoring commits were detrimental. But even in this case only a fraction (i.e. 80%) of the maintainability decrease was caused by the refactoring commits. This finding is more or less in line with the results of the manual refactoring phase of the project where we found that in most cases refactoring improved the overall maintainability of the systems with only a few minor exceptions. In the case of tool-aided refactoring, this finding also holds true for 3 projects out of 4.

Table 4.3. Quality changes of the selected projects

Company – Project	**Coding Issues**	**Maint. Before**	**Maint. After**	**Total Impr.**	**Ref. Impr.**	**Ref. Impr. %**
Comp I – Proj A	1,444	4.449238	4.411970	-0.037268	-0.029822	80
Comp II – Proj B	1,306	6.039320	6.072320	0.032999	0.032999	100
Comp III – Proj C	404	4.132307	4.258933	0.126627	0.144507	114
Comp IV – Proj D	682	6.158691	6.161626	0.002935	0.003142	107

Table 4.4. Quality changes caused by refactoring coding issues

Coding Issue Type	Project A			Project B			Project C			Project D		
	#	avg	ratio	#	avg	ratio	#	avg	ratio	#	avg	ratio
AddEmptyString	0	0	0	150	0	1 ↑	1	0	1 ↑	270	0	1 ↑
ArrayIsStoredDirectly	0	0	0	33	0.000027	1.31 ↑	2	0.000184	3.15 ↑	0	0	0
AvoidPrintStackTrace	0	0	0	17	0	1 ↑	3	0.000061	1.71 ↑	0	0	0
AvoidReassigningParameters	0	0	0	30	0.000010	1.12 ↑	174	0.000007	1.08 ↑	10	0	1 ↑
AvoidThrowingNullPointerException	0	0	0	10	0	1 ↑	3	0.000540	7.32 ↑	0	0	0
AvoidThrowingRawExceptionTypes	0	0	0	3	0.000062	1.63 ↑	0	0	0	6	0	1 ↑
BooleanGetMethodName	0	0	0	125	0	1 ↑	1	0	1 ↑	9	0	1 ↑
EmptyCatchBlock	0	0	0	20	0.000058	1.68 ↑	14	0.000550	7.43 ↑	30	0.000044	1.51 ↑
EmptyIfStmt	0	0	0	32	0.000012	1.14 ↑	5	0.000074	1.87 ↑	5	0	1 ↑
MethodNamingConventions	0	0	0	2	0	1 ↑	21	0.000004	1.05 ↑	9	0	1 ↑
PositionLiteralsFirstInComparisons	409	0.000114	2.33 ↑	26	0.000054	1.42 ↑	5	0.000432	6.05 ↑	9	0.000060	1.70 ↑
PreserveStackTrace	0	0	0	90	0.000003	1.04 ↑	24	**-0.000001**	0.99 ↑	8	0	1 ↑
SimpleDateFormatNeedsLocale	0	0	0	141	0.000001	1.02 ↑	17	0	1 ↑	58	0	1 ↑
SwitchStmtsShouldHaveDefault	0	0	0	170	0.000016	1.19 ↑	47	0.000020	1.23 ↑	23	0.000010	1.12 ↑
UnnecessaryConstructor	1,035	**-0.000077**	0.10 ↑	41	**-0.000023**	0.73 ↑	7	**-0.000170**	**0.99** ↓	40	**-0.000042**	0.51 ↑
UnnecessaryLocalBeforeReturn	0	0	0	149	0	1 ↑	13	0	1 ↑	135	**-0.000002**	0.98 ↑
UnusedLocalVariable	0	0	0	32	0.000012	1.14 ↑	30	0.000050	1.58 ↑	0	0	0
UnusedPrivateField	0	0	0	36	0.000005	1.06 ↑	0	0	0	4	0	1 ↑
UnusedPrivateMethod	0	0	0	29	0.000007	1.08 ↑	6	0.000285	4.33 ↑	5	0	1 ↑
UseLocaleWithCaseConversions	0	0	0	57	0.000108	2.26 ↑	0	0	0	44	0.000034	1.40 ↑
UseStringBufferForStringAppends	0	0	0	113	0.000008	1.09 ↑	30	0.000099	2.16 ↑	17	0.000008	1.09 ↑

The seemingly negative results of *Project A* could be explained by a very project specific factor. The system which suffers from maintainability decrease belongs to a company where developers performed only two different types of refactorings, namely *PositionLiteralsFirstInComparisons* and *UnnecessaryConstructor*. Their motivation might have been that these refactorings required only local changes (i.e. they were low hanging fruits), therefore they were easier to manage and test the code after the modification. However, the effect of this limited set of refactoring types is completely different from a more balanced set of refactorings (see Table 4.4).

The results of the other three companies support this hypothesis, as they performed a much wider range of refactoring tasks, and the maintainability of their systems increased in all cases. In the case of *Project C* and *Project D*, it is even true that the refactoring commits caused a larger increase in the maintainability than the overall increase at the end of the phase, which means that code modifications other than refactorings decreased the maintainability. In the *UnnecessaryConstructor* line of Table 4.4, we can see that all the values are negative, meaning that this type of refactoring caused a maintainability decrease in each and every system. Taking into consideration the fact that out of the two types of refactoring performed by Company I, *UnnecessaryConstructor* was the absolute dominant by its number, it is now clear that the overall decrease in the maintainability of their system can be credited to this single type of refactoring. It is an interesting question of why removing an *UnnecessaryConstructor* decreases the maintainability, which we elaborate on in Section II.

To summarize, we observe that the overall effect of the automatic refactoring phase tends to be positive, and the small bias is caused by the dominant number of a single type of refactoring (i.e. *UnnecessaryConstructor*) in *Project A*.

II Impact of Automatic Refactoring Types on Software Maintainability

During the automatic refactoring period, developers fixed different kinds of coding issues, which had different effects on software maintainability. In Table 4.4 we show for each system the number of fixed coding issues (column '#') and its average maintainability change (column 'avg') credited to the various kinds of coding issue types the developers fixed (semi)automatically. As the maintainability change of a single commit measured on the scale of 0 to 10 is extremely small, we also added a column to the table (column 'ratio') that reflects the number of times this change was bigger or smaller compared to an average maintainability change caused by a non-refactoring commit. We refer to this number as *nonRefactAvg* in the following, and its value is 0.00005. The $\uparrow$ means that the actual change is bigger than the average maintainability change of the non-refactoring commits, while $\downarrow$ means a worse effect than the average. Please note that the average maintainability change of the non-refactoring commits is negative, so a maintainability decrease may still be marked with $\uparrow$ (meaning that the actual maintainability degradation is smaller than that of an average commit). For example, a ratio of 1.68 $\uparrow$ means that the actual maintainability improvement is greater than the average non-refactoring commit by 1.68 times of the absolute value of the average change:

$$avg = nonRefactAvg + ratio * |nonRefactAvg|$$

This is why a neutral change value (i.e. 0) is marked with 1 $\uparrow$, as 0 is better than the average maintainability change of non-refactoring commits, which is negative.

We can readily see that Company I fixed only 2 types of coding issues in *Project*

A, as we already pointed it out in the previous section. The other companies fixed a wider range coding issues, 21 types altogether. The results indicate that in 55% of the cases refactoring increased the overall maintainability of the system, while it decreased the maintainability in only 10% of the cases (shown in bold). In 35% of the cases it did not cause any noticeable difference in maintainability measured by the model (i.e. the model was insensitive to the change). If we compare the results with the average maintainability changes of non-refactoring commits, we can see that only one value caused a larger maintainability decrease than an average non-refactoring commit. So even in those few cases where a refactoring type caused a maintainability decrease, it was much smaller than an average maintainability degradation introduced by a commit containing no refactorings. Also, the largest maintainability increases caused by some refactoring types are more than 7 times bigger than the average decrease caused by non-refactoring commits.

Looking closer at the results, we can see that fixing the *UnnecessaryConstructor* coding issue decreased the maintainability in each case. This issue occurs when a constructor is not necessary; i.e., when there is only one constructor, it is public, has an empty body, and takes no arguments. The automatic refactoring algorithm simply deleted these constructors. Intuitively, the maintainability of the source code should have been increased because we deleted unnecessary code and decreased the lines of code metric as well. However, ColumbusQM is not directly affected by the system size as it could lead to false conclusions like larger systems are necessarily harder to maintain, so the code reduction itself would not justify a maintainability increase anyway. Instead of the mere sizes of the systems, the maintainability model relies on the distribution of the method lengths. In this particular case the method length distribution is shifted towards the direction of longer methods as a lot of “good quality” code/methods got deleted. The removed constructors consisted of just a few lines, had no coding issues, had small complexity and they did not refer to other classes. Therefore, a maintainability decrease occur upon deleting such good quality methods due to the shift in the distribution of metric values like length, complexity or number of parameters of the remaining methods.

There are two other issues where the maintainability of a system decreased for some of the projects. One issue is the *UnnecessaryLocalBeforeReturn* that caused a decrease in maintainability for *Project D*. In this case the automatic refactoring algorithm simply inlined the value of the local variable into the return statement (which resulted in a line deletion as well). This should have increased the maintainability because it reduces the method length and removes a coding issue from the source code. However, it did not change the maintainability or it even decreased it (albeit the decrease was very small compared to other changes). Investigation of this phenomena revealed that a single change in lines of code or in the number of minor (low-priority) rule violations is so small that it has no noticeable effect. What is more, in some cases fixing these issues introduced code clones as well (the only difference between two methods was the unnecessary local variable) which immediately decreased the measured maintainability.

The other issue causing a maintainability decrease is *PreserveStackTrace* in *Project C*. The typical fix of this issue is to add the root exception as a second parameter to the constructor of the newly thrown exception. However, Company III could not apply this strategy as their own exception classes did not override this two parameter constructor. So instead of the usual fix, they instantiated a new exception in a local variable, called its *initCause()* method with the root exception and threw the new exception. Besides

Table 4.5. Ratios of quality changes on individual metrics level

Coding Issue	**Pri.**	#	**AD**		**CBO**		**CC**		**CD**		**CLOC**		**LLOC**		**McCC**		**NII**		**NLE**		**NOA**		**PAR**		**P1**		**P2**		**P3**		**Maint.**	
			+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
AddEmptyString	P3	421	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.06	0	0	0
ArrayIsStoredDirectly	P2	35	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0.97**	0	0	0	**0.97**	0
AvoidPrintStackTrace	P2	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.05	0	0	0	0.05	0
AvoidReassigningParameters	P3	214	0	0	0	0	0	0.11	0	0	0	0	0.02	0	0	0	0	0	0.12	0	0	0	0	0	0	0	0	0	0.16	0	0.29	0.02
AvoidThrowingNullPointerExcept.	P1	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.23	0	0	0	0	0	0.23	0
AvoidThrowingRawExceptionTypes	P2	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.11	0	0	0	0.11	0
BooleanGetMethodName	P3	135	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.07	0	0	0
EmptyCatchBlock	P1	64	0	0	0	0	0.05	0.06	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0.91**	0	0	0.13	0	0	**0.91**	0.02
EmptyIfStmt	P2	42	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.07	0	0	0	0.1	0.02
MethodNamingConventions	P3	32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.09	0	0.03	0
PositionLiteralsFirstInComparisons	P1	449	0	0	0	0	0.02	0.02	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0.88**	0	0	0	0	0	**0.88**	0
PreserveStackTrace	P2	122	0	0	0	0	0.01	0.03	0	0	0	0	0	0.01	0	0	0	0.02	0	0	0	0	0	0	0	0	0.25	0	0	0	0.25	0.06
SimpleDateFormatNeedsLocale	P3	216	0	0	0	0	0.04	0.01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.15	0	0.1	0.01
SwitchStmtsShouldHaveDefault	P2	240	0	0	0	0	0.01	0.01	0	0	0	0	0	0.01	0	0	0	0	0	0	0	0	0	0	0.04	0	0.35	0	0	0	0.4	0.01
UnnecessaryConstructor	P3	1,123	0.13	0.02	0	0	0	0.08	0	0	0.04	0.01	0	0.14	0	0.16	0.01	0.21	0	0.14	0	0	0	0.22	0	0.08	0	0.13	0.19	0	0.04	**0.73**
UnnecessaryLocalBeforeReturn	P3	297	0	0	0	0	0	0.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.12	0	0	0.1
UnusedLocalVariable	P2	62	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.15	0	0	0	0.15	0
UnusedPrivateField	P2	40	0	0	0.03	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.03	0
UnusedPrivateMethod	P2	40	0	0	0.03	0	0	0.05	0	0	0	0	0	0	0	0	0	0	0.05	0	0	0	0.08	0	0	0	0.05	0	0	0	0.1	0.05
UseLocaleWithCaseConversions	P1	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0.86**	0	0	0	0	0	**0.86**	0
UseStringBufferForStringAppends	P2	160	0	0	0	0	0	0	0	0	0	0	0.01	0	0	0	0	0	0	0	0	0	0	0	0	0	0.25	0	0	0.01	0.26	0

the additional lines, the fix also introduced a new incoming call to the *initCause()* method of the exception objects. All these decrease the maintainability, which slightly outweigh the positive effect of removing a coding issue.

All in all, the results indicate that despite the seemingly counter-intuitive effects of fixing some issues, refactoring different types of coding issues usually increase code maintainability.

III Impact of Automatic Code Refactoring on Code Metrics

Table 4.5 shows all the sensor nodes (internal quality properties) of the ColumbusQM ADG (see Figure 2.2), and the overall maintainability of a system as well. Sensor nodes represent *goodness* values of source code metrics. In the table we list two ratios for each sensor node. A ratio is the number of coding issue fixes when the refactoring caused a positive (column '*+*') or negative (column '*–*') change to the goodness value of the current sensor, divided by the number of all refactorings (positive, negative, and zero change). The values larger than 0.5 are highlighted in bold. The table also shows the priority (column *Pri.*) for each coding issue according to a scale between 1-3, and describes how dangerous an issue is (P1 – critical, P2 – major, P3 – minor).

The goal of the project was to increase the maintainability of the software systems. The column '*Maint. +*' shows the ratio of how many times a refactoring increased the overall maintainability of a system. For example, 0.86 means that *UseLocaleWithCaseConversions* fixes had a positive impact on maintainability in 86% of the cases. The column '*Maint. –*' shows the ratio of how many times a refactoring decreased the overall maintainability of a system. Looking at the same line again, we see that the value is 0, which means that fixing this type of issues did not decrease the maintainability. The remaining 14% did not affect the maintainability in either a positive or negative way.

Looking at these values, we can see that fixing coding issues generally increases the overall maintainability. However, there are a few issue types which did not change the maintainability at all, or they even decreased it. Increases happened mostly because of the expected behavior of the maintainability model, namely, decreasing the number of coding issues in the source code improves maintainability and stability, hence the quality. This behavior can be observed mainly in the *P1*, *P2*, *P3* columns (the numbers of coding issues with different priorities, respectively). For example, *ArrayIsStoredDirectly* did not change any other sensors, just the number of *P2* coding issues and this increased the maintainability in each case. Still, this pattern cannot be applied to every row in the table. For example *AddEmptyString*, *BooleanGetMethodName* coding issues increased the goodness of *P3* sensor in 6-7% of the cases but we cannot see any increase in maintainability. This is because the positive effect of *P3* sensor was so small that it increased the overall maintainability to such a small amount that it is lost due to rounding errors.

An interesting observation can be made on the *EmptyCatchBlock* where besides the 91% improvement of *P1*, one can see a 13% decrease in the *P2* sensor. A closer look into this case told us that in some automatic *EmptyCatchBlock* refactorings developers choose to solve the issue with "put an e.printStackTrace() call into the catch block" option for the refactoring algorithm which resolved the *EmptyCatchBlock* but introduced a new *AvoidPrintStackTrace* issue at the same time.

Another compelling case is the *AvoidReassigningParameters* issue which has a definitive improvement in the logical lines of code (LLOC) and nesting level (NLE) sensors. Fixing this reassignment involved removing some code parts that reduced the code lines and sometimes the complexity (i.e. the maximal nesting level) of the projects. Besides reducing the number of coding issues, these improvements caused a maintainability increase in 29% of the cases. However, in 2% of the cases, we observed a maintainability decrease. This is because in 11% of the cases the removal of some code parts resulted in new code clones (CC), hence two or more code parts differed only in the removed statements. So the effect of this refactoring is not easy to predict, but in the majority of the cases we observed a maintainability increase.

UnusedPrivateField increased the goodness of the CBO sensor in 3% of the cases but it did not affect any of the other sensors. This happened mostly because of the small number of fixes and also because it sometimes introduced *UnusedImports* coding issues as well.

Section II explained why *UnnecessaryLocalBeforeReturn* coding issue decreased maintainability. Table 4.5 shows that the introduction of code clones (CC) had a bigger effect on maintainability than the fixes of the *P3* issue. Similarly, *UnnecessaryConstructor* is also referred to in Section II and its precise effects can be seen in Table 4.5. Almost every sensor is affected by this coding issue fix, but this is mainly because of the large number of refactorings.

In summary, we observe that fixing coding issues by automatic refactorings does not have a significant impact on metrics in most of the cases, mainly because the changes are local. However, some fixes have an effect on metrics one would not think of at first glance.

4.2.3 Threats to Validity

Even in a case study which was carried out in a controlled environment, there are many different threats which should be considered when we discuss the validity of our observations. Here, we give a brief overview of the most important ones.

Heterogeneity of the commits

As we were interested in the effect of particular refactoring types on software maintainability, we filtered out those commits that contained different types of refactorings. Although the number of such commits was relatively low, it is obviously a loss of information. Additionally, when a commit contained multiple refactoring operations of the same type, we had to use the average of the maintainability changes to estimate the effect of an individual refactoring operation. This is also a threat to validity, as the same refactorings may have a different impact on the same system. However, its likelihood is minimal, as all the refactorings have been carried out (semi)automatically, and this resulted in very similar type of modifications in the code.

Maintainability analysis relies only on the ColumbusQM maintainability model

The maintainability model is an important part of the analysis as it also determines what we consider as an "effect on maintainability" of refactorings. Currently we rely on ColumbusQM with all of its advantages and disadvantages. On the positive side this model has been published, validated and reflects the opinion of developers [20];

however, we saw that the model might miss some aspects that would reflect some changes caused by refactorings.

Limitations of the project

We claim that our experiment was carried out in an *in vivo* industrial context. However, this project might had unintentional effects on the study. For example, as in the manual phase, the budget for refactoring was not "unlimited" and some companies were looking for fixes that required the smallest amount of extra effort. A good example of this is Company I, who really just performed two such types of refactorings.

Limitations of the supported refactoring types

The supported automatic refactorings focus on fixing 21 different coding issues. It is only a fraction of the possible and widely used set of refactoring operations, therefore our overall conclusions are limited to these type of refactorings. However, most of these refactorings are simple yet powerful tools for improving the code structure agreed by all the companies involved in the project.

4.3 Analysis of Developers' Opinions on Refactoring Automation

There are several challenges which should be kept in mind during the design and development phases of a refactoring tool, and one is that developers have several expectations that are quite hard to satisfy. To address this, during the manual phase we asked developers what they thought about refactoring automation. Developers provided us with several recommendations in the manual phase (How did they refactor? Do they think that it is possible to automate their steps? If yes, how would they automate them?). Then, they gave us feedback on the resulting implementations (We asked them how they used it and how much it helped them in their everyday work.). Besides our experiences, we also examined their opinions.

Here, we present and summarize the opinions of the developers and the several challenges we faced on how to automate refactoring transformations.

4.3.1 What developers think about refactoring automation?

Throughout the manual refactoring and the automatic refactoring phases, we asked developers to fill out surveys for the refactoring operations they had carried out. For each refactoring commit, they had to fill out a survey that contained questions targeting the initial identification steps, and they also had to explain why, how, and what they modified in their code. There were around 40 developers involved in this phase of the project (5-10 per company). The questions related to our study were the following:

- *How difficult would it be to automate your* ***manual refactoring*** *for the issue?* (1 - very easy, 5 - very hard) + explanation
- *How much did the* ***automated refactoring*** *help in your task?* (1 - no help at all, 5 - great help) + explanation

I Manual refactorings

During the manual refactoring phase of the project, developers refactored their codebase manually, and they filled out a survey for each refactoring. We had an online Trac system for this purpose, and whenever they opened a ticket for an issue, they had to explain why they found it problematic, and answer some more questions. Similarly, we asked them some further questions when they closed the ticket after they had finished the refactoring.

Among these questions, they had to rate on a scale of 1 to 5 (1 - very easy, 5 - very hard) how difficult it would be to automate the manual refactoring. Along with this number, they had to justify their answer.

Table 4.6. Developers' feedback on how hard it would be to automate refactoring operations

Num. of Replies	Avg	Med	Dev	Num. of Types
430	2.06	1	1.23	61

The developers completed the survey for 430 tickets, as can be seen in Table 4.6. Our results tell us that developers gave responses for 61 different kinds of coding issues (actually we showed them around 220 different kinds of coding issues). Figure 4.7 shows the histogram of their replies. As can be seen, most of the refactorings were rated with smaller values, which means that they were optimistic about the automation: they thought that most of the coding issues could be easily fixed through automated transformations. However, they also identified some cases where they thought that the automation would be hard to realize. Notice also that Table 4.6 also supports this observation, as the average value is around 2 and the median is 1. Table 4.7 lists the coding issues and the level of difficulty of their automation based on feedback of the developers.

Table 4.7. How difficult the refactoring automation of coding issues is according to developers

Aut.	Coding Issues
Very Hard (5)	AvoidInstanceofChecksInCatchClause, ExceptionAsFlowControl, EmptyIfStmt
Hard (4)	SwitchStmtsShouldHaveDefault, AvoidCatchingThrowable, MethodReturnsInternalArray
Medium (3)	UseStringBufferForStringAppends, AvoidSynchronizedAtMethodLevel, SignatureDeclareThrowsException, AvoidCatchingNPE, AbstractClassWithoutAbstractMethod, ConsecutiveLiteralAppends, LooseCoupling, NonThreadSafeSingleton, ReplaceHashtableWithMap, SystemPrintln, UnusedFormalParameter, UseLocaleWithCaseConversions, UnsynchronizedStaticDateFormatter
Easy (2)	EmptyCatchBlock, OverrideBothEqualsAndHashcode, PreserveStackTrace, UnnecessaryLocalBeforeReturn, AtLeastOneConstructor, UnusedPrivateField, UnusedPrivateMethod, AvoidThrowingRawExceptionTypes, UnusedLocalVariable, AvoidDuplicateLiterals, AvoidDeeplyNestedIfStmts, AddEmptyString, AvoidFieldNameMatchingTypeName, ArrayIsStoredDirectly, AbstractNaming, ImmutableField, OnlyOneReturn, UnnecessaryConstructor, UnnecessaryWrapperObjectCreation
Very Easy (1)	AvoidPrintStackTrace, UnusedImports, UseIndexOfChar, InefficientStringBuffering, IntegerInstantiation, MethodArgumentCouldBeFinal, CyclomaticComplexity, BooleanInstantiation, BigIntegerInstantiation, BeanMembersShouldSerialize, CollapsibleIfStatements, CompareObjectsWithEquals, IfElseStmtsMustUseBraces, LocalVariableCouldBeFinal, SimplifyConditional, ShortVariable, UncommentedEmptyMethod, UnnecessaryFinalModifier, UnusedModifier, UnnecessaryReturn, VariableNamingConventions

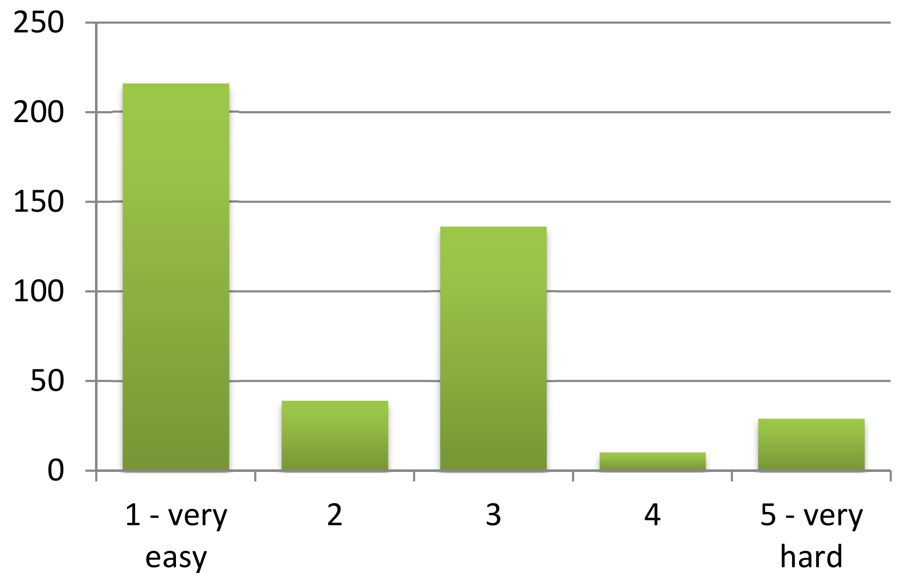


Figure 4.7. Histogram of the answers given for "*How difficult would it be to automate your manual refactoring for the issue?*"

II Automated refactorings

Based on the observations of the manual period, we started to develop the automated refactoring tool. When we began the development of the tool, we selected the refactoring transformations for implementation based on the earlier feedback of the developers. Also, we asked the companies to provide us with a list of coding issues (with priorities) that they wanted to fix in the automated phase so that we could concentrate on the most desired ones. After gathering the lists, we ranked each coding issue by the values the companies provided us. Then we created a ranked list of the coding issues that most of the companies wanted at the top, and the coding issues that nobody wanted at the end. Interestingly, the resulting list contained many issues that were no longer considered during the manual phase, most probably because the companies fixed all occurrences of some issue types so they were not interested in the automation of these.

We started implementing the refactoring algorithms based on this ordered list. We developed automatic refactoring solutions for 42 different coding issues. The supported list of coding issues consisted of 22 different issue types that were considered during the manual period plus 20 new ones.

During the automatic refactoring stage, we asked the developers to once again document their refactorings. This time, we incorporated the survey into our tool that asked them to fill it out after each refactoring transformation. This way, we gathered over 1,700 answers for 30 coding issue types (see Table 4.8).

Table 4.8. Total help factor survey

Num. of Replies	Avg	Med	Dev	Num. of Types
1,726	3.05	3	1.23	31

In the automatic phase, we asked developers about how much the automated refactoring solution assisted them in their refactoring task. They had to give a value between 1 and 5 here as well. A 5 meant that the automation helped a lot, while a 1 meant that

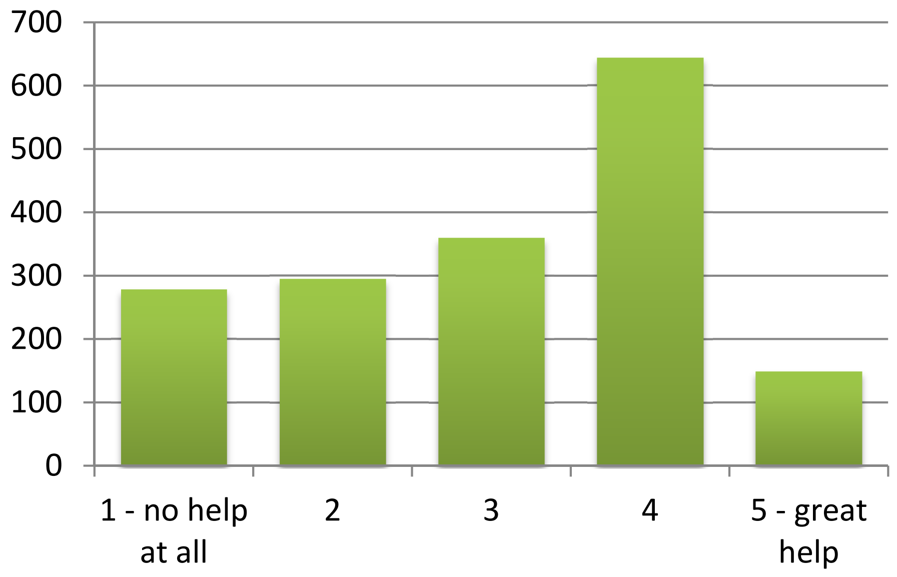


Figure 4.8. Histogram of the answers given for the question *"How much did the automated refactoring help in your task?"*

it did not help at all (or it even made the situation worse). As we see in Table 4.8, the average of the replies was around three. In Figure 4.8, the distribution of the responses can be seen in a histogram. This tells us that developers were generally satisfied with the automated refactoring solutions, and they gave a score of 4 in many cases.

Actually, if we consider all the transformations where the given value is greater than 1 (these are the transformations that the developers said were a help), we find that all the refactorings made the tasks of developers easier or faster, except for two cases. This can be seen in Figure 4.9, where we can see the degree of help for each kind of coding issue. The points stand for the average help of a refactoring solution and the bars around them indicate the standard deviation.

Every refactoring algorithm for a coding issue got a value above 1, except the *LooseCoupling* and the *MethodNamingConventions* coding issues. In their explanations, the developers said they found that fixing one issue had a communication overhead that sometimes made it easier for them to refactor the code manually in the IDE instead. However, this overhead might be negligible if they fix more issues together. For example, the refactoring solution for *MethodNamingConventions* issue suggests a better name for a method (e.g. if a method name starts with an uppercase letter it recommends the same name beginning with a lowercase letter). After the developer accepted the refactoring suggestion, they had to wait until our tool applied the modification. This could take a few seconds because of the server-client architecture.

Upon examining Figure 4.9 again, we realized that when we consider not only the average but also the standard deviation for each coding issue, we can classify the following 5 refactoring types as 'sometimes bad': *LongFunction*, *CyclomaticComplexity*, *UseStringBufferForStringAppends*, *UselessParentheses*, *TooManyMethods*.

The developers explained these as follows. The *UselessParentheses* issue fell into the same category as the former two; it is faster to do it manually in some cases. The *LongFunction* and *CyclomaticComplexity* issue fixing refactoring solutions used an extract method refactoring algorithm where the algorithm applied a heuristic to find parts of the code that can be extracted to satisfy the requirement by the issue, to reduce the length of the method or to reduce the complexity of it. The main problem

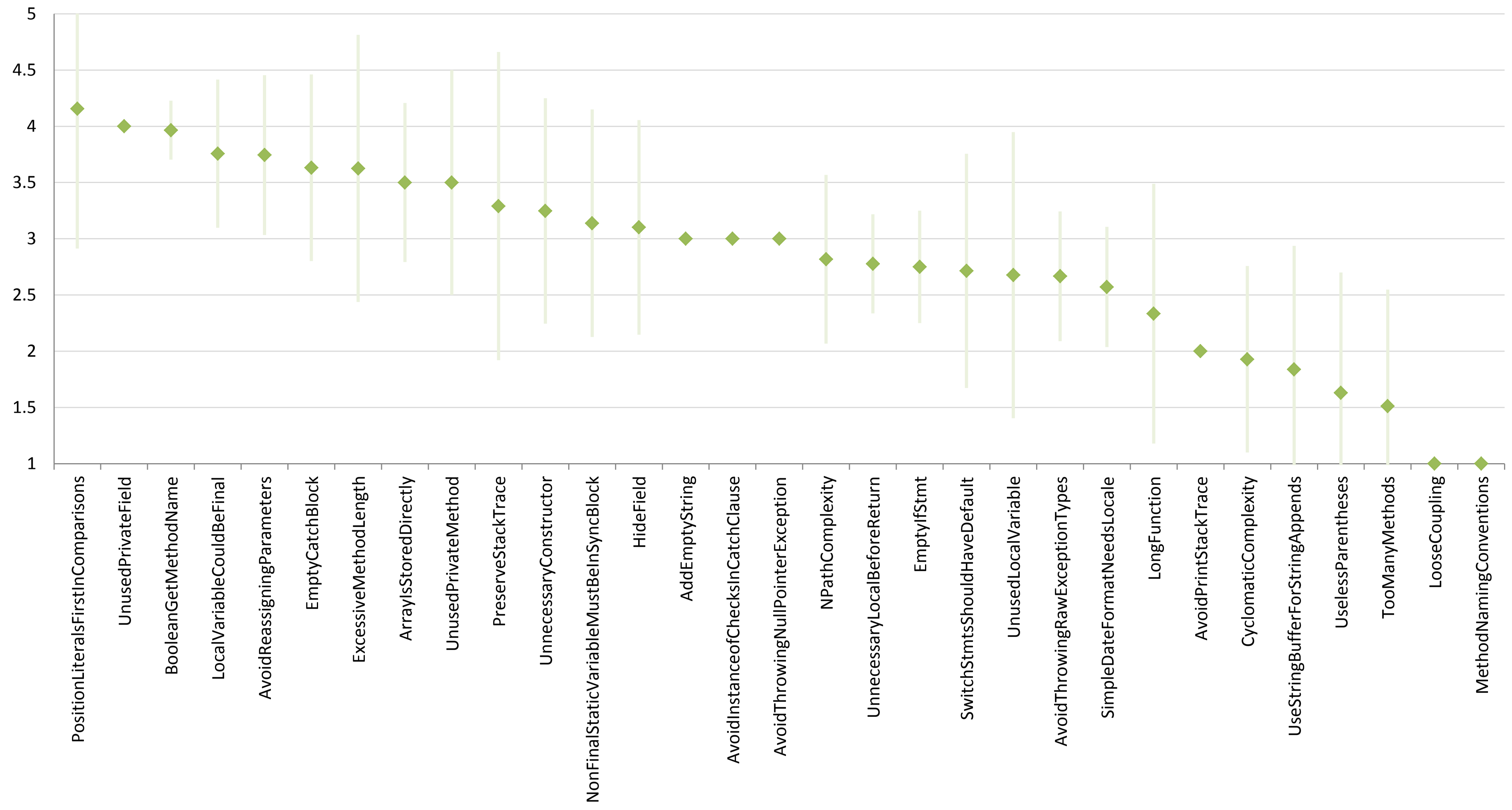


Figure 4.9. How much the automated refactoring solution assisted the developers (5 - great help, 1 - no help at all)

with this algorithm was that it was hard for developers to fathom how it worked. They simply preferred to do it manually instead of using the tools. The *TooManyMethods* issue suffered from the same problem, but in this case the underlying algorithm was 'extract class'. Developers' notes on the *UseStringBufferForStringAppends* issue show that although they were satisfied with the semantic aspect of the algorithm, many formatting problems arose.

4.3.2 Did automation increase developers productivity?

Previously, we found that the automatic tool helped in the everyday work of developers in most cases. We also found in some cases that time was a relevant factor in considering a refactoring a help or not. Here, we will examine whether tool-assistance increases developer productivity.

In the questionnaire we drew up during the automatic period we asked developers how much time it took to finish a refactoring with tool-assistance; and how much time it would have taken doing the refactoring manually. We got replies to this question for approximately 7,800 refactorings by the end of the project. Figure 4.10 represents the survey data. Here, we observe how many times faster a tool-assisted refactoring is compared to a manual one on average. The light-gray bar below the number *1* shows when the manual and automatic refactoring takes roughly the same time. When a tool-assisted refactoring is slower than doing it manually it is on the left-hand side of the bar, otherwise it is on the right-hand side. Colors also represent slower refactorings with red, faster ones with green, and when they take about the same time with orange bars.

We found that automated solutions in average are 1.7 times faster than manual ones. However, as Figure 4.10 shows how the difference for particular coding issue types vary. For example, there are six cases where manual methods are faster. A closer look on the slower issues reveals that it ties up with the observations discussed in the previous section. Simpler fixes are faster to make in the IDEs manually because of the server-client architecture of FaultBuster. However, the results also indicate that there are cases where the automatic technique was 3-5 times faster. One of the reasons why such an increase in speed was achieved is because these coding issues are refactored in batches. For example, the *UnusedConstructor* coding issue was 90% of the times executed in a batch together with 10 issues of the same type.

4.3.3 Lessions Learned

Thanks to the Refactoring Project, during the development of FaultBuster, we had chance to get immediate feedback from potential users of the tool in all stages of its development (starting from the design phases to the last testing phases of the project).

During the design phase of the tool we consulted regularly with the developers of the participating companies concerning the refactoring transformations which they wanted to be available in the final product. Throughout the initial meetings it became clear that they wanted ready solutions for their actual problems, particularly for those which were easily understandable for the developers and by solving them, they could gain the most in terms of increasing the maintainability of their products. However, they did not really provide us with a concrete list of the issues that they wanted us to deal with. In addition, most of the developers said that before the project they had not used any refactoring tools except the ones provided by their IDEs. Therefore,

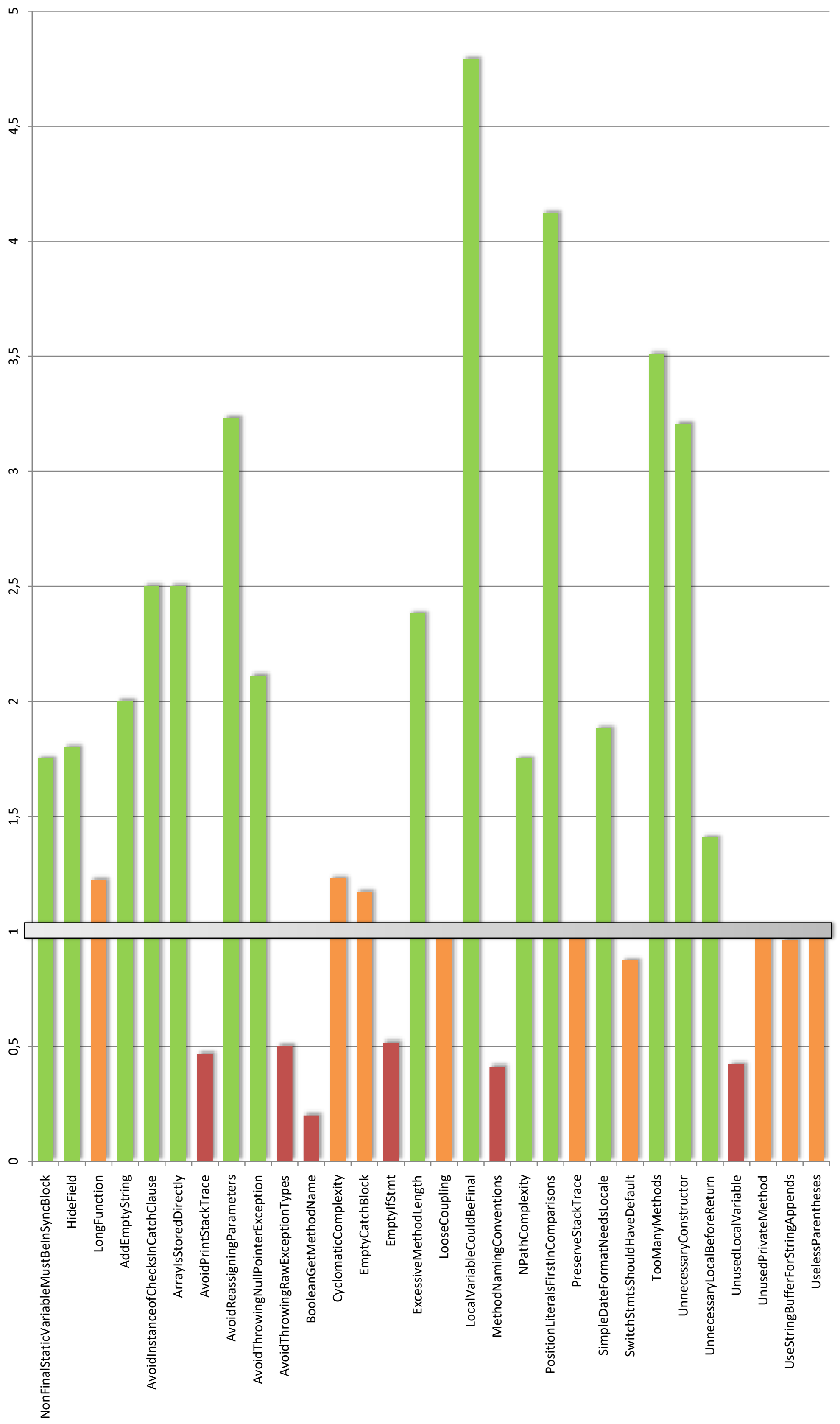


Figure 4.10. How much faster was it to do a refactoring with tool-assistance on average

we started implementing transformation algorithms to fix coding rule violations which were very common in their projects. Soon, when we provided the companies with the first prototype versions for testing, they started to send us lots of other issue types and refactoring algorithms that they wanted to be supported in the new releases. Among the desired refactorings, there were some more complex ones too like eliminating long or complex methods. In the end, we also implemented an algorithm that eliminates clones (code duplications) from the source code.

At the end of the project, we can indeed say, that FaultBuster performed well and was tested exhaustively by the companies. The companies participating in the project performed around 6,000 refactorings altogether which fixed over 11,000 coding issues. Interviews with the developers showed that they found the everyday work with the tool really helpful in many situations and they intended to keep using it in the future.

We gathered a lot of experience on how to design and implement such a tool, and also on the final usability of FaultBuster, which we will briefly summarize below.

I Challenges in how to automate refactoring transformations

Precise syntax tree Without a doubt, a key consideration of refactoring transformations is to *have a precise representation of the source code.* One can model the source code as an *abstract syntax tree* (AST) to perform different (graph) transformations on it. Transformations can be just as good as the underlying representation is, so we found it necessary to have an accurate and complete AST. As an illustration, consider a rename method refactoring. Here, we do not simply change the method name, but *a*) we have to check that the new name does not conflict with other method names in the same scope (e.g. parent and child classes); *b*) we have to check for disambiguation in other classes where the method is invoked; *c*) and then, when it passes the former two checks we are allowed to rename the method and all of its invocations to the new name. To do this, we have to analyze all the dependencies, and potentially include external ones as well.

Regenerate (only the) modified code After the transformations on the AST, we have to apply the changes to the source code. To do this, we have to (re)generate the source code from the AST (at least, and preferably only for the modified code parts). It is also advisable not to introduce unnecessary changes to the other parts of the code.

Code formatting The process of code generation requires some indentation and code formatting as well. It is usually hard for the users to specify formatting rules, and hence it is also hard to regenerate a code formatted exactly as the user would like to see it. This was one of the most difficult challenges we could not fully overcome, and this caused the most dissatisfaction among developers. However, based on our experiences, developers mostly accepted this limitation if they found the refactoring to be semantically correct and they had to reformat the code only a bit manually (e.g., they could easily do this in the IDE automatically).

Patch generation As the last step after the transformation, we generated a *diff* (difference file or patch) between the old source code and the new one. Then we sent this diff file to the IDE where the developers could decide to accept or reject the modification.

Code clone elimination Another interesting experience was that the developers eagerly wanted to eliminate code duplications (code clones). By the end of the project, we developed an experimental algorithm that was able to refactor code clones via extract method and extract class refactoring transformations. Note that automated code clone detection is a hot research topic as well [80], especially code clone elimination. It is quite a challenge to come up with a good solution for this issue.

II What makes a refactoring operation good or bad?

Precise problem detection Developers only wanted recommendations made for real faults or optimization opportunities, and they wanted to avoid false positives. Looking at false recommendations takes time, and it does not bring any benefit to the project. Besides false positive issues, they also wished to avoid true negative issues. As a common use case, they said they wanted to remove all the occurrences of a certain type of issue. Reporting only some occurrences would give them a false sense of security.

Understandability of the transformations Refactorings with a good and easy-to-comprehend description were more popular among the developers. Unlike those refactoring solutions that required more parameters or were harder to understand, developers rarely used these and gave worse scores in the survey.

Performance It was important to carry out the modifications quickly, or at least quicker than could be done manually.

Batch refactorings One way to improve efficiency is by supporting the refactoring of several issues at the same time. With the automated tool, developers were able to fix many issues of the same type all at once (we called this *batch* refactoring). This batch-refactoring process made refactoring tasks a lot faster. For example, they were able to fix all occurrences of *UnusedConstructor* issues with a press of a button. This option was beloved by developers, and batch refactorings got better scores in the survey. However, we did not allow batch refactoring of all the issues. We had to implement some restrictions in this process because we observed that developers tended to accept these refactorings without checking the result of the automated refactoring operations. This was flattering because it meant that they trusted the algorithm and its results. Nonetheless, we did not want them to blindly accept the refactorings. Therefore, we only allowed the refactoring of one type of issue at a time, and we only allowed it for some simpler refactorings. This way we guaranteed that they had to check complex refactorings (e.g. extract class) and ensured that simpler ones ran faster.

Comment handling Comments are integral parts of the source code, and sometimes they are closely related to source code elements. From these transformation, developers expected that they would also be able to handle these situations. For instance, a refactoring that removes an unused constructor should remove the comment before the constructor as well. Similarly, in some cases they asked us to generate simple comments.

4.4 Related Work

Related Tools

Since the introduction of the term 'refactoring' [11], many researchers studied it [30] as a technique, e.g., to improve source code quality [48, 81], and many tools were implemented providing different features to assist developers in refactoring tasks. FaultBuster is a refactoring tool to detect and remove code smells, i.e., in this section, we shall give an overview of tools which have similar capabilities, or are related to FaultBuster through some specific features.

In a recent study, Fontana et al. examined refactoring tools to remove code smells [72]. They evaluated the following tools: Eclipse [82], IntelliJ IDEA [83], JDeodorant [84], and RefactorIT [85]. In the case of JDeodorant, they say that this "*is the only software currently available able to provide code smell detection and then to suggest which refactoring to apply to remove the detected smells.*" To evaluate the other refactoring tools, they relied on the code smell identification of iPlasma [86] and inCode [87].

In an earlier study, Pérez et al. also identified smell detection and automatic corrections as an open challenge for the community, and proposed an automated bad smell correction technique based on the generation of refactoring plans [88].

As regards detecting bad smells, there are many static analyzers available to automatically identify programming flaws, like the products of Klocwork Inc. [89] or Coverity Inc. [90] These tools are sometimes able to identify serious programming flaws (e.g. buffer overflow or memory leak problems) that might lead to critical or blocker problems in the system. There are open source or free solutions as well, such as PMD [27], FindBugs [91], CheckStyle [92] for Java, and the Code Analysis features and FxCop in Visual Studio [93]. These tools usually implement algorithms to detect programming flaws, but fixing the identified issues remains the task of the developers.

The DMS Software Reengineering Toolkit [94] product of Semantic Designs Inc. has a 'program transformation engine' which allows the tool to perform code generation and optimization, and makes it able to remove duplicated code (with CloneDR).

There are many IDEs available with automatic refactoring capabilities and they support typical code restructurings (e.g. renaming variables, classes) and some common refactorings from the Fowler catalog. For instance, IntelliJ IDEA was one of the first IDEs to implement these techniques and it is able to support many languages (e.g. PHP, JavaScript, Python), not just Java, which it was originally designed for. Eclipse and NetBeans also implement similar algorithms. However, neither of these IDEs support the automatic refactoring of programming flaws. And there are many plugins available to extend their refactoring capabilities, such as ReSharper [95] and CodeRush [96] for .NET.

Compared to these tools, the above all lack the feature of scanning the code and suggesting which refactorings to perform, which is one of the main strengths of FaultBuster. JDeodorant, as an Eclipse plug-in, is the only tool that has a similar capability, as it is able to identify four kinds of bad smells (namely 'Feature Envy', 'State Checking', 'Long Method' and 'God Class'), and refactor them by using a combination of 5 automatic refactoring algorithms. FaultBuster is more general in a way, as it allows the refactoring of coding issues (see Table 4.1) and it has plug-in support for IntelliJ and NetBeans too (besides Eclipse).

Another good feature of FaultBuster is its ability to effectively perform a large set of refactorings (i.e. batch refactorings) together on a large code base. The lack of tools available that can handle a massive Java code base and provide a large class of useful refactorings also motivated the development of refactoring tools like Refaster [97] from Google [98].

Related Studies

Since the term 'refactoring' was introduced [11, 12], many researchers have studied its role in software development. Some studies estimate that about 70–80% of all structural changes in the code are due to refactorings [99, 100], which clearly indicates its importance in software evolution. Mens et al. published a survey to provide an extensive overview of research work in the area of software refactoring [30] and cited over 100 studies. However, the popularity of the topic has been increasing recently.

Automation techniques can support the regular task of refactoring and they are intensively studied by researchers. Ge et al. implemented the BeneFactor tool which detects developers' manual refactoring and reminds them that automation is available, then it completes the refactoring automatically [101, 102]. Vakilian et al. proposed a compositional paradigm for refactoring (automate individual steps and let programmers manually compose the steps into a complex change) and implemented a tool to support it. Henkel et al. implemented a framework which captures and replays refactoring actions [103]. Jensen et al. used genetic programming for automated refactoring and the introduction of design patterns [104]. Also, there are many approaches available to support specific refactoring techniques, such as extract method [105, 106], refactoring to design patterns [84] and clone refactoring [107].

There seems to be, however, disagreement among researchers as to whether refactoring truly improves software maintainability or not. Stroulia and Kapoor [45] investigated how metrics were affected and found that size and coupling metrics of their system decreased after the refactoring process. Du Bois and Mens [46] studied the effects of refactoring on internal quality metrics based on a formalism to describe the impact of a representative number of refactorings on an AST representation of the source code. Du Bois wrote his dissertation after studying the effects of refactoring on internal and external program quality attributes [48] and earlier Du Bois et al. [47] proposed refactoring guidelines for enhancing cohesion and coupling metrics; they got promising results by applying these transformations to an open-source project. Kataoka et al. [49] provided a quantitative evaluation method to measure the maintainability enhancement effect of refactorings. Yu et al. [38] adapted a modeling framework in order to analyze software qualities to determine which software refactoring transformations were the most appropriate. Moser et al. [50] studied the impact on quality and productivity as they observed small teams working in similar, highly volatile domains and assessed the impact of refactoring in a close to industrial environment. Their results indicate that refactoring not only increases software quality, but also improves productivity. One of the few industrial case studies that investigated the typical use and benefits of refactorings was carried out by Kim et al. [108] at Microsoft. Their survey revealed that the refactoring definition in practice was not confined to a rigorous definition of semantics-preserving code transformations and that developers perceived that refactoring involves substantial cost and risks. They found that the top 5 percent of preferentially refactored modules in Windows 7 experience a greater reduction in the number of inter-module dependencies and several complexity measures but they increase the size of more than the remaining 95 percent. This indicates that measuring the impact of refactoring requires multi-dimensional assessment.

A large-scale study was carried out by Murphy-Hill et al. [58] where they studied manual refactorings from Fowler's catalog, and their data set spans over 13,000 developers with 240,000 tool-assisted refactorings of open-source applications. Similarly, Negara et al. [57] presented an empirical study that considered both manual and automated refactorings. They reported that they analyzed 5,371 refactorings applied by students and professional programmers, but they did not provide more information about the systems in question.

Most of the above studies were performed on either several small projects and/or open-source systems, which is one important difference compared to our study, as we examined a *large amount of automatic refactorings on proprietary software.* Another difference is that we used the ColumbusQM to objectively measure changes in the maintainability, while earlier studies relied just on internal code metrics. It allows us to compare different refactorings and draw conclusions which might help developers in planning refactoring tasks or inspire research projects.

4.5 Summary

In this chapter, we summarized our experiences of the automatic refactoring period of the Refactoring Project. We sought to develop automated refactorings and for this purpose we designed FaultBuster, an automated refactoring framework. We presented an automated process for refactoring coding issues. We used the output of a third-party static analyzer to find refactoring suggestions. We created an algorithm that is capable of locating a source code element in an AST based on textual position information. The algorithm transforms the source code into a searchable geometric space by building a spatial database.

We had to take into account several expectations of the developers when we designed and implemented the automatic refactoring tools. Among several challenges of the implementation, we identified some quite important ones, such as performance, indentation, formatting, understandability, precise problem detection, and the necessity of a precise syntax tree. Some of these have a strong influence on the usability of a refactoring tool, hence they should be considered early in the design phase. We made an exhaustive evaluation, which confirmed that our approach can be adapted to a real-life scenario, and it provides viable results.

We made interesting observations about the opinions of the developers who utilized our tools. The results showed that they found most of the manual refactorings of coding issues easily implementable via automatic transformations. Also, when we implemented these transformations and observed the automated solutions, we found that almost all refactoring types helped them to improve their code.

Employing the QualityGate SourceAudit tool, we analyzed the maintainability changes caused by the different refactoring tasks. Our analysis revealed that out of the supported coding issue fixes, all but one type of refactoring operation had a consistent and traceable positive impact on the software systems in the majority of cases. Here, 3 out of the 4 companies involved achieved a more maintainable system at the end of the refactoring phase. We observed however that the first company preferred low-cost modifications, therefore they performed only two types of refactorings from which removing unnecessary constructors had a controversial effect on maintainability. Another observation was that it was sometimes counter productive to just blindly apply the automatic refactorings without taking a closer look at the proposed code modification. It happened several times that the automatic refactoring tool asked for user input to be able to select the best refactoring option, but developers used the default settings because this was easier. Some of these refactorings then introduced new coding issues, or failed to effectively remove the original issue. So human factor is still important, but the companies were able to achieve a measurable increase in maintainability just by applying automatic refactorings.

Last but not least, this study shed light on some important aspects of measuring software maintainability. Some of the unexpected effects of refactorings (like the detrimental effect of removing unnecessary constructors on maintainability) are caused by the special features of the applied maintainability model.

The fact that developers tested the tool on their own products provided a real-world test environment. Thanks to this context, the implementation of the toolset was driven by real, industrial motivation and all the features and refactoring algorithms were designed to fulfill the requirements of the participating companies. We implemented refactoring algorithms for 40 different coding issues, mostly for common programming flaws. By the end of the project the companies refactored their systems with over 5 million lines of code in total and fixed over 11,000 coding issues. FaultBuster gave a complex and complete solution that allowed them to improve the quality of their products and to incorporate continuous refactoring into their development processes.

"We cannot solve our problems with the same thinking we used when we created them."

— Albert Einstein

5 Applications of Model-Queries in Anti-Pattern Detection

In the detection of coding anti-patterns, the starting point of the refactoring process is to provide developers with problematic points in the source code. Developers then decide how to handle the issues they found. During the Refactoring Project, first developers investigated the list of reported anti-patterns and manually addressed the problems. Based on these experiences, the actual needs of partners were evaluated, and a refactoring framework was implemented with support for anti-pattern detection and guided automated refactoring with IDE integration.

In FaultBuster we used a third-party coding rule violation detection tool called PMD (see Section 4.1.2). PMD is a widely used, open-source static analyzer tool in the Java community. It has been integrated into SonarQube [109] and has its own Eclipse plugin. At the time of the development this seemed like a good solution with many benefits. Then, it quickly became clear that achieving our goals using PMD has some drawbacks. First, as we mentioned in Section 4.1.3, we had to implement the Reverse AST-search Algorithm to find the problematic source code elements in the AST. Second, on several occasions PMD did not provide precise problem highlights, as in Listing 4.5. Third, the reports of developers indicated (see Section 4.3.3) that in many cases the suggested coding issues are not real problems (false positives) and that there are several instances where it lacks the power to identify real ones (true negatives).

In this chapter, we focus on the detection of coding anti-patterns. In order to create a superior detection tool, we will investigate the costs and benefits of using the popular industrial Eclipse Modeling Framework (EMF) as an underlying representation of program models processed by four different general-purpose model query techniques based on native Java code, OCL evaluation and (incremental) graph pattern matching. We will provide an in-depth comparison of these techniques on the source code of 28 Java projects using anti-pattern queries taken from refactoring operations in different usage profiles. Our results reveal that general purpose model queries can outperform hand-coded queries by 2-3 orders of magnitude, with the trade-off of an increase in memory consumption and model load time of up to an order of magnitude.

5.1 Motivation

In the Refactoring Project, the original plan was to use the Columbus ASG of SourceMeter as the program representation together with its API to implement queries, since the API offers program modification functionality for implementing refactorings as well. However, queries for finding anti-patterns and the actual modifications can be separated. Our work builds on this separation to investigate the performance of various query solutions. Our aim was to include generic, model-based solutions in the comparison. Generic solutions offer flexibility and additional features like change notification support in the EMF and reusable tools and algorithms, like supporting high-level declarative query definitions [110, 111]. Such features could reduce the effort needed to define refactorings as well.

In the following, we investigate two viable options for developing queries for refactorings: (1) execute queries and transformations by developing Java code working directly on the ASG; and (2) create the EMF representation of the ASG and use EMF models with generic model-based tools. Several years ago, we found that typical modeling tools were able to handle only middle-sized program graphs [112]. We now re-exam this question and assess whether model-based generic solutions have evolved to compete with hand-coded Java-based solutions. We seek answers to questions like: *What are the main factors that affect the performance of anti-pattern detection (like the representation of program models, their handling and traversing)? What size of programs can be handled (in terms of memory and runtime) with various solutions? Does incremental query execution lead to a better performance?*

We should add that while we present our study on program queries in a refactoring context, our results can be applied in other areas as well. For instance, program queries are applied in several scenarios in maintenance and evolution from design pattern detection to impact analysis; furthermore, we think that real-life case studies are first-class drivers of improvement of model-driven tools and approaches.

5.2 Technological Overview

In this section, we first give a brief overview on program queries. First, we will show how to represent Java programs as an ASG or EMF model, then present the graph pattern formalism and use it to capture various anti-patterns.

5.2.1 Introduction to Program Queries

Program queries play a central role in various software maintenance and evolution tasks. Refactoring, an example of such tasks, seeks to change the source code of a program without altering its behavior in order to improve its readability, maintainability, or to detect and eliminate coding anti-patterns. After identifying the location of the problem in the source code, the refactoring algorithm applies predefined operations to fix the issue. In practice, the identification step is frequently defined by program queries, while the manipulation step is captured by program transformations.

Advanced refactoring and reverse engineering tools (like the Columbus framework [25]) first construct an Abstract Semantic Graph (ASG) as a model from the source code of the program, which enhances the traditional Abstract Syntax Tree with semantic edges for method calls, inheritance, type resolution, etc. In order to handle large programs, the ASG is typically stored in a highly optimized in-memory representation. Moreover, program queries are captured as hand-coded programs traversing the ASG driven by a visitor pattern, which may require a lot of development and maintenance effort.

Models used in model-driven engineering (MDE) are uniformly stored and manipulated in accordance with a metamodeling framework, such as the Eclipse Modeling Framework (EMF), which offers advanced tooling features. Essentially, EMF automatically generates a Java API, model manipulation code, notifications for model changes, persistence layer in XMI, and simple editors and viewers (and many more) from a domain metamodel, which can significantly speed up the development of EMF-compliant domain-specific tools.

EMF models are frequently post-processed by advanced model query techniques based on graph pattern matching that exploits different strategies such as local search [113] and incremental evaluation [114]. Some of these approaches can be scaled up for large models with millions of elements in forward engineering scenarios, but up to now, no systematic investigation has been carried out to demonstrate if they can be efficiently applied as a program query technology. If this is the case, then advanced tooling offered by the EMF could be readily used by refactoring and program comprehension tools without drawbacks.

5.2.2 Managing Models of Java Programs

I Abstract Semantic Graph for Java

The Java analyzer of the Columbus reverse engineering framework is found in SourceMeter and it is used to get program models from the source code (similarly as for the C++ language [25, 115]). The ASG contains all information that is in a typical AST extended with semantic edges (e.g., call edges, type resolution, overrides). It is designed primarily for reverse engineering purposes [116, 117] and it conforms to our Java metamodel.

In order to keep the models of large programs in memory, the ASG implementation is heavily optimized for low memory consumption, e.g., handling all model elements and String values in a central store to avoid storing duplicate values. However, these optimizations are hidden behind an API interface.

In order to assist the processing aspect of the model (e.g., executing a program query), the ASG API supports visitor-based traversal [118]. These visitors can be used to process each element on-the-fly during traversal, without manually coding the (usually preorder) traversal algorithm.

Example 1 *To illustrate the use of the ASG, we present a short Java code snippet and its model representation in Figure 5.1. The code consists of a public method called* `equals` *with a single parameter, together with a call of this method using a Java variable* `srcVar`*. The corresponding ASG representation is depicted in Figure 5.1b, omitting type information and boolean attribute values such as the final flags for readability.*

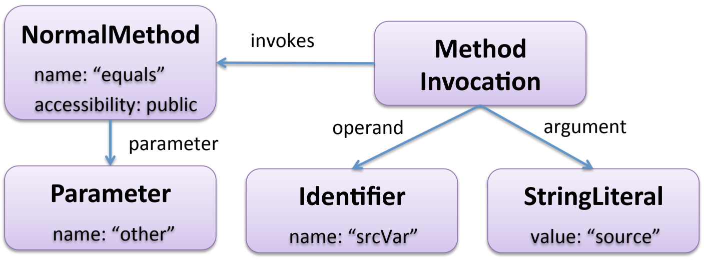
The method is represented by a `NormalMethod` node that has the name `equals` and `public` accessibility attribute. The method parameter is represented by a `Parameter` node with the name attribute `other`, and it is connected with the method using a `parameter` reference.

The call of this method is represented by a `MethodInvocation` node that is connected to the method node by an `invokes` reference. The variable the method is executed

```
public boolean equals(Object other) {...}

...
// Code inside another method
// The variable 'srcVar' is defined locally
srcVar.equals("source");
...
```

(a) Java Code Snippet



(b) ASG Representation

Figure 5.1. ASG Representation of Java Code

on is represented by an `Identifier` node via an `operand` reference. Lastly, an `argument` reference connects a `StringLiteral` node describing the `"source"` value.

II Java Application Models in EMF

Metamodeling in the EMF Metamodeling is a fundamental part of modeling language design as it allows the structural definition (e.g., abstract syntax) of modeling languages. The EMF provides a Java-based representation of models with various features, such as notification, persistence, and generic, reflective model handling. These common persistence and reflective model handling capabilities enable the development of generic (search) algorithms that can be executed on any given EMF-based instance model, regardless of its metamodel.

The model handling code is generated from a metamodel defined in the *Ecore* metamodeling language together with higher level features such as editors. The generator work-flow is highly customizable, e.g., allowing the definition of additional methods.

The main elements of the Ecore metamodeling language are the following: `EClass` elements define the types of objects; `EAttribute` extend EClasses with attribute values while `EReference` objects present directed relations between EClasses.

Example 2 *As an illustration, we present a small subset of the Java ASG metamodel realized in the Ecore language in Figure 5.2 that focuses on method invocations depicted in Figure 5.1. The metamodel was designed to provide an equivalent representation of the ASG of the Columbus framework in the EMF, both at the model level and the generated Java API. The entire metamodel consists of* 142 *EClasses with* 46 *EAttributes and* 102 *EReferences.*

The `NormalMethod` and `Parameter` EClasses are both elements of the metamodel that can be referenced from Java code by name. This is represented by generalization relations (either direct or indirect) between them and the `NamedDeclaration` EClass. This way, both inherit all the EAttributes of the `NamedDeclaration`, such as the `name` and the `accessibility` controlling the visibility of the declaration.

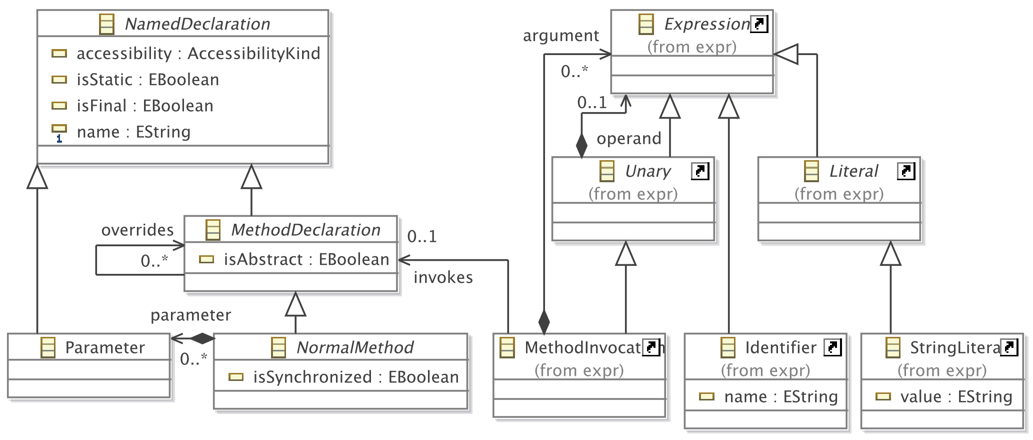


Figure 5.2. A Subset of the Ecore Model of the Java ASG

Similarly, the EClasses `MethodInvocation`, `Identifier` and `StringLiteral` are part of the `Expression` elements of Java. Instead of attribute definitions, the `MethodInvocation` is linked to other EClasses using three EReferences: (1) the EReference `invokes` points to the referred `MethodDeclaration`; (2) the `argument` selects a list of expressions to be used as the arguments of the called methods, and (3) the inherited `operand` EReference selects an expression representing the object the method is called on.

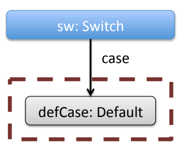
Notes on Columbus Compatibility The Java implementation of the Java ASG of the Columbus Framework and the generated code from the EMF metamodel use similar interfaces. This makes it possible to create a combined implementation that supports the advanced features of the EMF, such as the change notification support or reflective model access, and it remains compatible with the existing analysis algorithms of the Columbus Framework by generating an EMF implementation from the Java interface specification.

However, there are also some key differences between the two interfaces that should be addressed. The most important difference lies in multi-valued reference semantics, where the EMF disallows having two model elements connected multiple times using the same reference type, while the Columbus ASG occasionally relies on such features. To maintain compatibility, the EMF implementation is extended with proxy objects, which ensure the uniqueness of references. The implementation hides the presence of these proxies from the ASG interface while the EMF-based tools can navigate through them.

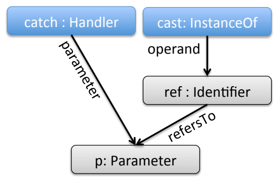
Other minor changes range from different method naming conventions for boolean attributes to defining additional methods to traverse multi-valued references. All of them are handled by generating the standard EMF implementation together with the Columbus compatibility methods.

5.2.3 Definition of Model Queries using Graph Patterns

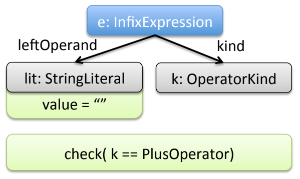
Graph patterns [110] are a declarative, graph-like formalism representing a condition (or constraint) to be matched against instance model graphs. This formalism is usable for various purposes in model-driven development, such as defining model transformation rules and defining general purpose model queries including model validation constraints.



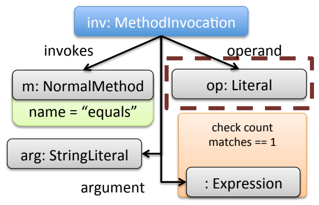
(a) Switch w/o Default



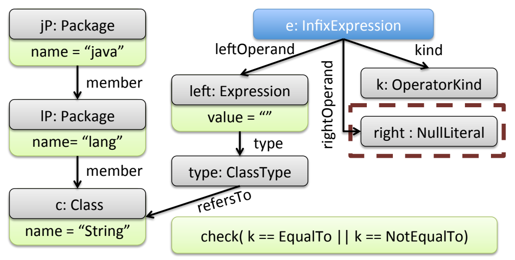
(b) Catch Problem



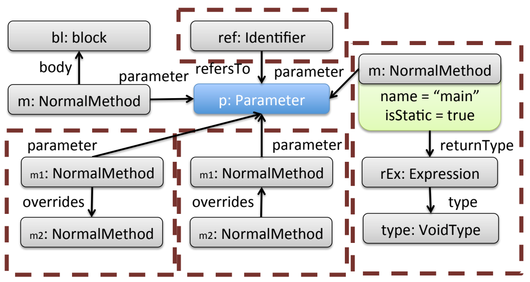
(c) Concatenation to Empty String



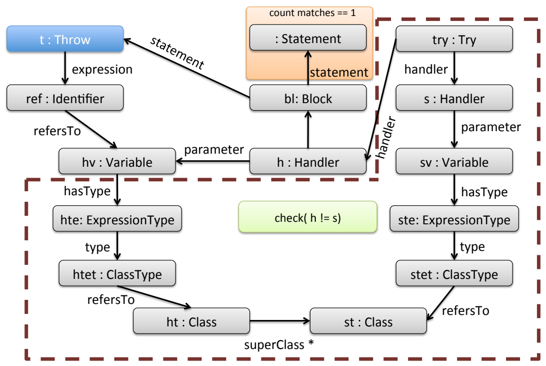
(d) String Literal as Compare Parameter



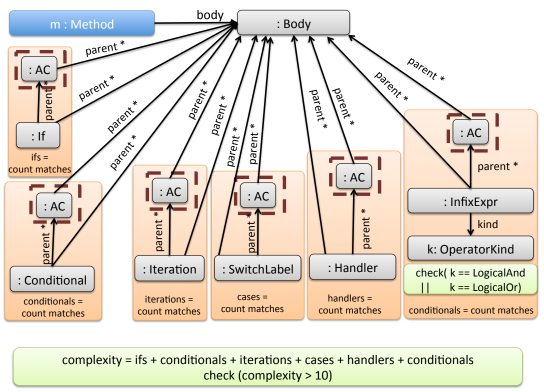
(e) String Compare without Equals Method



(f) Unused Parameter



(g) Avoid Rethrow



(h) Cyclomatic Complexity

Figure 5.3. Graph Pattern Representation of the Search Queries

Here, we give only a brief overview of the concepts, and for more detailed, formal definitions see [119].

A graph pattern consists of *structural constraints* prescribing the interconnection between the nodes and edges of a given type and *expressions* to define *attribute constraints*. These constraints can be illustrated by a graph where the nodes are classes from the metamodel, while the edges prescribe the required connections of the selected types between them.

Pattern parameters are a subset of nodes and attributes interfacing the model elements interesting from the perspective of the pattern user. A *match* of a pattern is a tuple of pattern parameters that fulfills all the following conditions: (1) it has the same structure as the pattern; (2) it satisfies all structural and attribute constraints; and (3) it does not satisfy any NAC.

Complex patterns may reuse other patterns by applying different types of *pattern composition constraints*. A *(positive) pattern call* identifies a subpattern (or called pattern) that is used as an additional set of constraints to meet, while *negative application* *conditions* (NAC) describes the cases where the original pattern is *not* valid. Next, *match set counting* constraints are used to calculate the number of matches a called pattern has, and use them as a variable in attribute constraints. Pattern composition constraints can be illustrated as a subgraph of the graph pattern.

When evaluating the results of a graph pattern, any subset of the parameters can be bound to model elements or attribute values that the pattern matcher will handle as additional constraints. This allows reuse of the same pattern in different scenarios, such as checking whether a set of model elements fulfills a pattern, and it lists all matches of the model.

Example 3 *Figure 5.3 captures all the search problems from section 5.3 as graph patterns. Here, we will only discuss the String Literal as Compare Parameter problem (5.3d) in detail, and all other patterns can be interpreted in a similar way.*

The pattern consists of five nodes called `inv`, `m`, `op` and `arg`, representing the model elements of the types `MethodInvocation`, `NormalMethod`, `Literal`, `Expression` and `StringLiteral`, respectively. The distinguishing (blue) formatting for the node `inv` means that it is the parameter of the pattern.

In addition to the type constraints, node `m` shall also fulfill an attribute constraint ("equals") on its name attribute. The edges between the nodes `inv` and `m` (and for `arg`) represent a typed reference between the corresponding model elements. However, as the node `op` is included in a NAC block (depicted by a dotted red box), the edge `operand` means that either no operand should be given or the operand must not point to a `Literal` typed node.

Finally, to ensure that the invoked method has only a single parameter, the number of arguments are counted. The highlighted part of the pattern formulates a subpattern consisting of the arguments of the `MethodInvocation`, and the number of these subpattern matches is checked to be 1*. This kind of checking could also be expressed using a NAC block describing a different parameter, but the use of match counting is easier to read.*

After matching this pattern to the model from Figure 5.1, the result will be a set containing a single element, namely the `MethodInvocation` instance.

5.3 Experiment Setup

In the first round of experiments we selected six types of anti-patterns based on the feedback of project partners and formalized them as model queries. The diversity of the problems was among the most important selection criteria, resulting in queries that varied both in complexity and programming language context ranging from simple traverse-and-check queries to complex navigation queries potentially with negative conditions. Here, we briefly and informally describe these refactoring problems and the related queries used in our case study.

Switch without Default Missing `default` case has to be added to the `switch`. *Related query:* We traverse the whole graph to find Switch nodes without a default case.

Catch Problem In a catch block there is an `instanceof` check, for the type of the catch parameter. Instead of the `instanceof` check a new catch block has to be added for the checked type and the body of the conditional has to be moved there. *Related* *query:* We search for identifiers on the left hand side of the `instanceOf` operator and check whether they point to the parameters of the containing catch block.

Concatenation to Empty String When a new `String` is created starting with a number, usually an empty `String` is added from the left of the number to force the `int` to `String` conversion, because there is no `int + String` operator in Java. A much better solution is to convert the number using the `String.valueOf()` method first. *Related query:* We search for empty string literals, and check the type of the containing expression. If the container expression is an infix expression, then we also make sure that the string is located at the left hand side of the expression and the kind of the infix operator is the String concatenation ("+").

String Literal as Compare Parameter When a `String` variable is compared with a `String` literal using the `equals()` method, it is unsafe to have the variable on the left hand side. Changing the order makes the code safe (by avoiding null pointer exception) even if the String variable to be compared is `null`. *Related query:* We search for all method invocations with the name "equals". Afterwards, we check that their single parameter is a string literal.

String Compare without Equals Method This refactoring was mentioned earlier. *Related query:* We search for the `==` operator and check whether the left hand side operand is of type `java.lang.String`. We have to check the right hand side operand as well: in case of `null` we cannot use the method call. In fact, it is not necessary because in this case the comparison operator is the right choice.

Unused Parameter When unused parameters remain in the parameter list they usually can be removed from the source code itself. *Related query:* We search for the places in the method body where parameters are used. However, there are specific cases when removing a parameter that is not used in the method body results in errors, such as (1) when the method has no body (interface or abstract method); (2) when the method is overridden by or overrides other methods; and (3) in `public static void main` methods.

After the first round of our experiments described in [120], it turned out that all antipatterns could be effectively evaluated by our selection of tools. In order to find the limits of the approaches, we selected two additional, more complex antipatterns that required additional capabilities.

Avoid Rethrowing Exception The catch block is unnecessary if the exception handling code only re-throws the caught exception without further actions. We look for a thrown exception in the catch block and check whether the thrown exception is the same (or descendant) as the caught one. However, simply rethrowing the exception is valid, if a specific exception is to be handled externally, while a more generic exception handler block is responsible for managing a superclass of the caught exception. This antipattern requires a transitive closure calculation for the inheritance hierarchy as a new feature.

Cyclomatic Complexity Cyclomatic complexity measures the number of linearly independent paths through a program's source code, usually calculated for a function as the number of decision points +1. A highly complex code (e.g. assessed using the cyclomatic complexity metric) tends to be difficult to test and maintain and it tends to have more defects. The pattern requires counting various types of program elements within a method body. This calculation relies on counting model elements together with simple arithmetic operations and extensive traversal around the containment hierarchy. To have the same validation format, we shall list the methods with cyclomatic complexity higher than 10.

5.4 Program Queries Approaches

Now, we give a brief overview of the possible approaches for implementing anti-pattern detection as program queries. First, a visitor-based search approach is described, followed by two different graph-pattern based approaches (both supported by the EMF-IncQuery), and then we will use the OCL language to describe the query problems.

5.4.1 Manual Search Code

The ASG representation allows one to traverse the Java program models using the visitor [118] design pattern that can form the basis of the search operations.

Visitor-based searches are easy to implement and maintain if the traversed relations are based on containment references, and require no custom setup before execution. However, as the order of the traversal is determined outside the visitor, non-containment references are required to be traversed manually, typically with nested loops. Alternatively, traversed model elements and references can be indexed, and in a post-processing step these indexes can be evaluated for efficient query execution. In both cases, significant programming effort is needed to achieve efficient execution.

Example 4 *The results of the String Literal as Compare Parameter (5.3d) pattern can be calculated by collecting all MethodInvocation instances from the model, and then executing three local checks whether the invoked method is called `equals`, if it has an argument with a type of `StringLiteral`, and if it is not invoked on a `Literal` operand.*

Figure 5.4 presents (a simplified) Java implementation of the visitor. A single `visit` method is used as a start for traversing all `MethodInvocation` instances from the model, and checking the attributes and references of the invocation. It is possible to delegate the checks to different `visit` methods, but then the visitor has to track and combine the status of the distributed checks to present the results that are difficult to implement in a sound and efficient way.

The ASG does not initially contain reverse edges in the model. It provides an API to generate these extra edges in a second pass after loading the model, but this requires extra time and memory. As the subject queries in this study could be implemented without these extra resources, to keep the memory footprint low, we prefer not to generate them.

```java
public class CompareParameterVisitor extends Visitor {

    //A set to store results
    private Set<MethodInvocation> invocations
        = new HashSet<MethodInvocation>();

    @Override
    public void visit(MethodInvocation node) {
        super.visit(node);
        //Checking invoked method name and number of parameters
        if ("equals".equals(node.getInvokes().getName())
                && node.getArgument().size() == 1) {
            //Node argument
            Expression argument = node.getArgument(0);
            //Node operand
            Expression operand = node.getOperand();
            //Type checking for argument
            if (argument instanceof StringLiteral
                    //NAC checking for operand
                    && !(operand instanceof Literal)) {
                //Result found
                invocations.add(node);
            }
        }
    }
}
```

Figure 5.4. Visitor for the String Literal as Compare Parameter Problem

5.4.2 Graph Pattern Matching with Local Search Algorithms

Local search based pattern matching (LS) is commonly used in graph transformation tools [121–123], which commences the match process from a single node and extends it in a step-by-step fashion with the neighboring nodes and edges following a *search plan*. From a single pattern specification multiple search plans can be calculated [113], hence the pattern matching process starts with a plan selection based on the input parameter binding and model-specific metrics.

A search plan consists of a totally ordered list of *extend* and *check* operations. An *extend* operation binds a new element in the calculated match (e.g., by matching the target node along an edge), while *check* operations are used to validate the constraints between the already bounded pattern elements (e.g., attribute constraints or whether an edge runs between two matched nodes). If an operation fails, the algorithm backtracks; and if all operations are executed successfully, a match is found.

Some extend operations, such as finding the possible source nodes of an edge and iterating over all elements of a certain type might be very expensive to execute during a search, but this cost can be reduced by the use of an incremental model indexer, such as the EMF-IncQuery Base[1]. This kind of indexer can be set up while loading the model, and then updating it on model changes using the notification mechanism of the EMF. If no such indexing mechanism is available (e.g., because of its memory overhead), the search planner algorithm should consider these operations with higher costs, and provide alternative plans.

Example 5 *To find all String Literals appearing as parameters of* `equals` *methods, a 7-step search plan given in Table 5.1 was used. First, all* `NormalMethod` *instances*

[1] https://wiki.eclipse.org/EMFIncQuery/UserDocumentation/API/BaseIndexer

Table 5.1. Search Plan for the String Literal Compare Pattern

Operation	Type	Notes
1: Find all `m` that `m` ⊂ `NormalMethod`	Extend	Iterate
2: Attribute test: `m.name=="equals"`	Check	
3: Find `inv` that `inv.invokes` → `m`	Extend	Backward
4: Count of `inv.argument` → `arg` is 1	Check	Called Plan
5: Find `arg` that `inv.argument` → `arg`	Extend	Forward
6: Instance test: `arg` ⊂ `StringLiteral`	Check	
7: Find `op` that `inv.operand` → `op`	Extend	Forward
8: NAC analysis: `op` ⊄ `Literal`	Check	Called plan

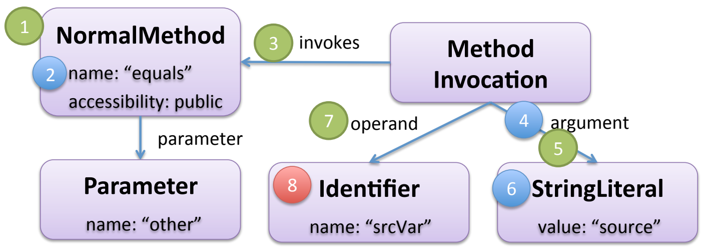


Figure 5.5. Executing the Search Plan

are iterated over to check for their name. Then a backward navigation operation is executed to find all the corresponding method invocations to check its argument and operand references. In the last step, a NAC check is executed by starting a new plan execution for the negative subplan, but this time only looking for a single solution.

Figure 5.5 shows the execution of the search plan on the simple instance model introduced previously. In the first step, the `NormalMethod` is selected, then its `name` attribute is validated, followed by the search for the `MethodInvocation`. At this point, following the `argument` reference ensured that only a single element was available, then the `StringLiteral` was found and checked. Lastly, the `operand` reference is followed, and a NAC check is executed using a different search plan.

It should be mentioned here that the search begins with listing all `NormalMethod` elements as opposed to the visitor-based implementation, which starts with the `MethodInvocations`. This was motivated by the observation that in a typical Java program there are more method invocations than method definitions, so starting this way would likely result in fewer traversed search states, while still finding the same results in the end. However, this optimization relies on having an index which allows cheap backward navigation during pattern matching for step 3 (unlike the ASG based solution where this information is not available without an extra traversal).

5.4.3 Incremental Graph Pattern Matching using the Rete algorithm

Incremental pattern matching [114, 124] is an alternative pattern matching approach that explicitly caches matches. This makes the results available at any time without an additional search, but the cache needs to be incrementally updated whenever changes are made to the model.

The Rete algorithm [125], which is well-known in rule-based systems, was efficiently adapted to several incremental pattern matchers [126–128]. The algorithm uses an extended incremental caching approach that not only indexes the basic model elements, but it also indexes *partial matches* of a graph pattern that enumerates the model element tuples that satisfy a subset of the graph pattern constraints. These caches are organized in the graph structure called a Rete network, which can be incrementally updated during model changes.

The *input nodes* of Rete networks represent the index of the underlying model elements. The *intermediate nodes* execute basic operations like filtering, projection, and join, on other Rete nodes (either input or intermediate) they are connected to, and store the results. Afterwards, the match set of the entire pattern is available as an *output (or production) node*.

When the network is initialized, the initial match set is calculated and the input nodes are set up to react to the model changes. When receiving a *change notification*, an *update token* is released on each of their outgoing edges. Upon receiving such a token, a Rete node determines how (or whether) the set of stored tuples will change, and releases update tokens on its outgoing edges. This way, the effects of an update will propagate through the network, eventually influencing the result set stored in the production nodes.

Example 6 *To illustrate a Rete-based incremental pattern matching, we first depict the Rete network of the String Literal as Compare Parameter pattern in Figure 5.6.*

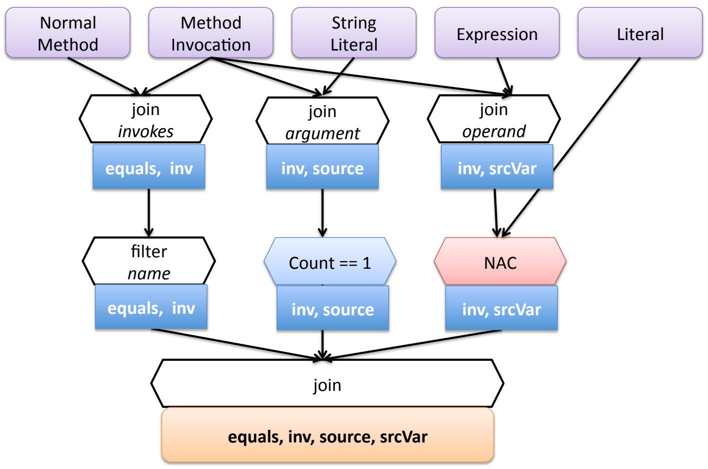


Figure 5.6. Rete Network for the String Literal Compare Pattern

The network consists of five input nodes that store the instances of the types `NormalMethod`, `MethodInvocation`, `StringLiteral`, `Expression` and `Literal`, respectively. The input nodes are coupled by join nodes that calculate the list of elements connected by `invokes`, `argument` and `operand` references, respectively. As both ends have already been enumerated in the parent nodes, both forward and backward references can be calculated efficiently. The invoked method list (output of the `invokes` join node) is filtered by the `name` attribute of Methods, while the argument lists are `filtered` for one per call. The NAC checking is executed by removing the elements with `Literal` types from the result of the `operand` join. Then, all partial matches are joined together to form the resulting matches.

```
context MethodInvocation:
def: stringLiteralAsCompareParameter : Boolean =
self.invokes.name = 'equals'
   and self.arguments -> exists(oclIsKindOf(StringLiteral))
   and self.arguments -> size() = 1
   and not self.operand.oclIsKindOf(Literal)
```

Figure 5.7. The OCL Expression of the String Literal as Compare Parameter Problem

It should be stressed that the Rete node, such as the `MethodInvocation` *in the example, can be used in multiple join operations; in such cases the final join is responsible for filtering out the unwanted duplicates (for a selected variable).*

5.4.4 Model Queries with OCL

OCL [111] is a standardized, pure functional model validation and query language for defining expressions in the context of a metamodel. The language itself is very expressive, exceeding the expressive power of first order logic by offering constructs such as collection aggregation operations (`sum()`, etc.). The rest of the section gives a basic overview of OCL expressions, and for a more detailed description of the possible elements consult the specification [111].

Variables of an OCL expression refer to instance model elements and a set of basic types including strings, various number formats and different kinds of collections. For these types, built-in operations are defined such as comparison operators and membership testing.

Furthermore, OCL expressions are compositional, allowing one to define sub-expressions in more complex expressions, including the `let` expression for defining additional variables, the `if` expression for implementing conditions and *iterator* expressions that evaluate subexpressions on all members of a collection.

Each OCL expression is valid in a *context*, described as a metamodel type. The OCL standard allows the definition of multiple context variables, but OCL implementations often just support a single one.

Example 7 *To illustrate the capabilities of OCL, Figure 5.7 formulates the String Literal as Compare Parameter problem as an OCL query. This query can be evaluated starting from a* `MethodInvocation` *context variable, which is referred to throughout the query as* `self`*.*

The query is described as the conjunction of 4 *different sub-expressions:*

1. *It is checked whether the target of the invocation has a* `name` *attribute with the value of* `'equals'`*. The type of the invoked call is not checked, as based on the metamodel it is known to be correct.*

2. *It is checked whether the list of* `arguments` *contains an element that has the type of (StringLiteral). The* `exists` *operation is one of the iterator operations that detects whether any member of the collection satisfies the condition.*

3. *It is checked whether the size of the arguments collection is exactly* 1.

4. *Lastly, the operand type is checked* not *to be* `Literal`*.*

OCL expressions can be evaluated as a search of the model, where the corresponding search plan is encoded in the expression itself. This makes the manual optimizations of the queries possible, but it requires a detailed understanding of the instance, metamodels, and the underlying OCL engine.

5.5 Measurement Context

To provide a context for our performance evaluation, next we will describe the executed measurements of this experiment. This includes a detailed evaluation of all our instance models and queries using different complexity metrics and the description of our measurement process. The selection of metrics was motivated by earlier results of [129] where the values of different metrics are compared with the execution time of different queries.

The use of metrics helps us to identify which queries/models are more difficult for the selected tools. And it also allows us to compare both the models and the queries with other available performance benchmarks.

5.5.1 Java Projects

The approaches were evaluated on a test set of 28 open-source projects. The projects are sized between $1kLOC$ and $2MLOC$, and used in various scenarios. The list of projects include the ArgoUML editor, the Apache CloudStack infrastructure manager tool, the Eclipse Platform, the Google Web Toolkit (GWT) library, the Tomcat Java application server, the SVNKit Subversion client, the online homework system WeBWorK, and the Weka data mining software, and many others. Table 5.2 contains the full list of projects and their analyzed versions (and projects where snapshots were used are marked in the table).

To compare these models, Table 5.2 shows different metrics that characterize them, including their size in terms of lines of code and in terms of number of nodes, edges and attributes of the graph representation, the number of metamodel types used and the indegree and outdegree of the graph nodes. The graph structure of all models are similar: they use about 90–100 of the types specified in the metamodel, and the average indegree and outdegree is 3. The big numbers in the maximum indegree column are related to the representation of the Java type system: a few types, such as `String` or `int` are referred to many times throughout the code.

In the remainder of the section, only the results related to the programs larger than $100kLOC$ are presented, as they still represent a wide range of Java applications, and in the case of smaller models the specific differences between the tools are much smaller (but similar to those presented here)[2].

5.5.2 Query Complexity

The antipatterns used different approaches in the various tools, resulting in different query complexity in each case. To compare them, Table 5.3 describes the complexity of queries implemented in the various tools. We have selected lists complexity measures for

[2]For a detailed test result containing all the models and raw measurement data visit our website: `http://incquery.net/publications/extended-program-query-comparison`

Table 5.2. Model Metrics

	Version	LOC	Node Count	Edge Count	Attribute Count	Type Count	Avg/Max InDegree		Avg/Max OutDegree	
ArgoUML	0.35.1 (*)	174 516	1 002 129	2 973 258	6 895 018	100	3	72 230	3	445
CloudStack	4.1.0	1 369 952	5 390 662	16 478 218	36 650 136	100	3,1	631 140	3,1	1 198
Eclipse	3.0.0	2 294 146	8 403 914	26 254 507	58 219 100	97	3,1	1 245 390	3,1	1 958
Frinika	0.5.1	64 828	429 407	1 292 961	3 065 383	99	3	54 286	3	844
GWT	2.3.0	1 078 630	3 219 239	9 986 705	22 364 819	101	3,1	392 098	3,1	1 206
Hibernate	3.5.0	773 166	2 444 419	7 563 207	16 789 330	102	3,1	193 769	3,1	522
Jackrabbit	2.8	590 420	1 765 882	5 341 431	12 145 662	100	3	271 217	3	708
Java DjVu	0.8.06	23 570	129 068	372 444	926 653	92	2,9	26 918	2,9	1 026
javax.usb	1.0.1	1 161	12 231	32 388	89 399	83	2,6	969	2,6	148
JFreechart	1.2.0	327 865	865 148	2 663 967	6 022 410	93	3,1	50 658	3,1	445
JML	1.0b3	10 159	72 598	212 544	520 599	94	2,9	4 908	2,9	221
JTransforms	2.4	38 400	295 009	945 643	2 053 900	80	3,2	117 775	3,2	217
Makumba	0.8.1.9	65 065	378 204	1 127 797	2 637 424	98	3	62 717	3	445
OpenEJB	4.5.2	575 363	1 785 660	5 428 385	12 377 185	101	3	152 624	3	540
Physhun	0.5.1	4 935	36 962	108 888	263 091	86	2,9	2 944	2,9	148
ProteinShader	0.9.0	22 651	137 416	391 322	997 679	88	2,8	9 654	2,8	445
Qwicap Guess	1.4b24	443	7 903	21 222	59 069	85	2,7	918	2,7	107
Robocode	1.5.4	28 245	204 362	599 556	1 500 298	97	2,9	17 323	2,9	445
sdedit	3.0.5	14 717	145 453	413 998	1 075 471	97	2,8	12 643	2,8	445
Stendhal	0.75.1	105 411	667 142	2 037 645	4 688 300	98	3,1	49 556	3,1	445
Struts2	1.4.0	274 092	927 163	2 849 021	6 452 090	100	3,1	95 272	3,1	620
Superversion	2.0b8	29 282	238 842	705 875	1 731 692	94	3,0	2 041	3,0	445
SVNKit	1.3.0.5847	114 189	698 753	2 203 436	4 843 209	93	3,2	57 987	3,2	272
Tomcat	8.0.0 (*)	459 579	1 338 601	4 084 668	9 302 681	102	3,1	116 637	3,1	620
WebWork	2.2.7	46 208	285 372	853 724	2 018 672	95	3	36 439	3	445
Weka	3.7.10 (*)	205 537	1 615 637	4 989 653	11 259 543	99	3,1	216 651	3,1	550
Xalan	2.7	349 681	708 445	2 093 338	4 937 831	93	3	87 447	3	445
Xins	2.2a2	21 698	164 989	472 003	1 193 822	89	2,9	15 169	2,9	445

Table 5.3. Query Complexity Metrics

	Visitor		Query						OCL
	LOC	**CC**	**Param.**	**Variables**	**Edges**	**Attr.**	**Calls**	**NEG**	**MC**
catch	78	14	4	6	3	0	1	0	9
concatenate	32	8	6	8	3	1	3	0	4
constant compare	39	10	6	11	5	0	2	2	7
no default switch	53	11	2	3	1	0	0	1	2
string compare	56	15	10	17	10	1	7	2	15
unused parameter	88	21	11	19	8	0	6	1	21
avoid rethrow	210	54	11	24	12	0	2	1	23
cyclomatic complexity	114	22	23	40	5	2	9	7	34

the different formalisms to understand how query complexity varies with the different approaches.

In the case of visitors, we calculate the lines of Java code required together with its cyclomatic complexity. The six original queries were written in less than 100 lines of code and had a cyclomatic complexity of 10–20. The two new queries were more complex both in terms of lines of code and cyclomatic complexity.

For graph patterns, we rely on metrics defined in [129]: the number of query *variables* and *parameters*, the number of *edge* and *attribute* constraints, the number of subpattern *calls* and the combined number of negative pattern calls and match counters *NEG*. It should be added that the metrics were not calculated from the graphical notation of Figure 5.3, but their implementation in the EMF-IncQuery, where different subpatterns were created to facilitate reuse both at the design level and during runtime. A subpattern call introduces new variables for the parameters of the subpattern that are the same as some parameters at their call site; this might lead to an increased number of variables compared to the number of edge and attribute constraints.

To measure the complexity of OCL queries, we used a minimum complexity (MC) metric presented in [130] that is based on calculating or estimating the number of model elements visited during the execution of its search, where multiple visits of the same element accounts as different ones. However, the metric definition relies on the model structures; and in order to have a model-independent metric, estimates need to be provided for the models.

Here, we calculate a lower bound of this metric by underestimating the number of visited model elements while mentioning that each OCL expression or operation will be evaluated with at most one model element that is related to the number of conditions to be evaluated. This way, it is possible to get a lower bound of the complexity for instance models that have at least one single result for the query.

The complexity of the queries over the different approaches behave in a similar way in almost every case except for the following three: (i) the *no default switch* case uses the simplest pattern and an OCL query, while in the case of visitors, (ii) the *concatenation* case uses the simplest visitor. (iii) Conversely, the calculation of *cyclomatic complexity* is clearly the most complex query in the graph patterns formalism and OCL, while its visitor is much simpler than the *avoid rethrow*. We think that this difference is based on the fact that the calculation of cyclomatic complexity needs only the traversal of the containment hierarchy that visitors excel in.

5.5.3 Measurement process

All the measurements were executed on a dedicated Linux-based server with 32 GB RAM running Java 7. On the server, the Java ASG of the Columbus Framework was installed along with the EMF-IncQuery (supporting graph pattern matching using both the local search and the Rete-based incremental approaches) and the Eclipse OCL [131] tool.

All the program queries were implemented as both visitors for the ASG (by a Columbus expert from the University of Szeged) and as graph patterns (by a model query expert from the Budapest University of Technology and Economics) - who was a different reviewer from the original implementer of the query. In the case of OCL expressions, we relied on our previous experience in comparing model query tools [129], where OCL experts were asked to verify the created queries. Visitors were executed on both model representations, while the graph patterns (both for local search-based and incremental queries) and the OCL queries were evaluated on the EMF representation. In order to also be able to understand use cases where multiple queries are executed together, indexes were built for all queries. In every case, the time to load the model from its serialized form and the time to execute the program query were measured together with the maximum heap size usage.

The query implementations were manually verified to return the same values for all the tools in three ways. First, (1) the specifications created were reviewed to fulfill the original, textual specifications. Then, (2) in a selection of smaller programs all instances were manually compared to return exactly the same issues. Lastly, (3) for each model, the number of issues found was reported and compared.

Every program query was executed ten times, and the standard deviation of the results was verified. Afterwards, we averaged the time and memory results without the smallest and the largest values. In order to minimize the interference between the different runs, for the execution of a model, tool and query a new JVM was created and ran in isolation. Also, all the measurements were performed with a 10 minute timeout: when loading the model, initializing and executing the query took more than the timeout, the measurement was treated as a failed one. The time to start up and shut down the JVM was not included in the measurement results.

5.6 Measurement Results

To compare the performance characteristics of the different program query techniques, next we will present the detailed performance measurement results.

5.6.1 Load Time and Memory Usage

Figure 5.4a shows the time required to load the models in seconds. As our measurements suggested that the model load time was practically independent of the query selection, we will only provide an aggregated result table. The only exception to this rule is the *cyclomatic complexity* pattern with incremental pattern matching: here we found that indexing the transitive closure of the containment hierarchy was prohibitively expensive both in terms of load time and memory usage. For this reason, we executed two sets of measurements: (1) one without initializing the *cyclomatic complexity* pattern (INC), and (2) another that also included this pattern (INC-CC).

Figure 5.8 shows the detailed load time and memory usage measurements for the Jackrabbit tool in box plots; and the diagrams for the other cases were similar. In general, the diagrams reveal that the repeated measurements of the test cases in general have very small differences, except in a few cases, and there are large differences when comparing the results of different techniques.

It can be seen that the load time is 3–4 times longer when using an EMF-based implementation over the manual Java ASG, and further increases can be seen when initializing the pattern matchers for local search and incremental queries. The two-phase load algorithm for the EMF model (EMF case), and the time to set up the indexes (local search) and partial matches (Rete) may account for these increases. As OCL does not use any specific index, no additional load overhead over the EMF visitor implementation was measured.

Table 5.4. Measurement Results

(a) Load Time (in seconds)

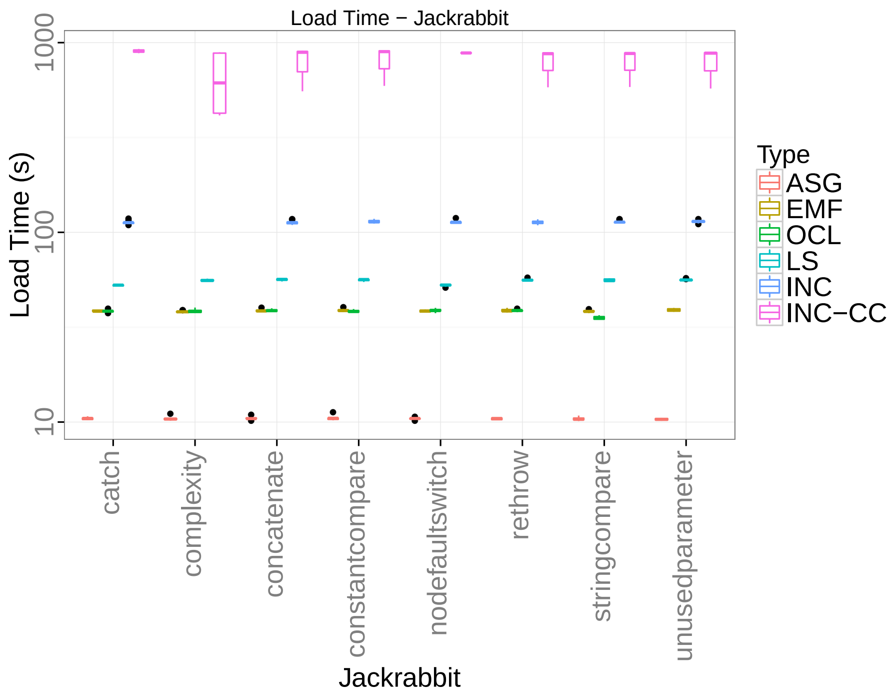
	ASG	EMF	OCL	LS	INC	INC-CC
CloudStack	27,5 ± 0,6	115 ± 3,2	115 ± 1,8	156 ± 3,0	343 ± 5,9	NA
ArgoUML	6,7 ± 0,1	25 ± 0,6	25 ± 0,5	35 ± 0,6	52 ± 1,3	312 ± 53,3
Eclipse	41,7 ± 0,7	169 ± 2,3	171 ± 2,8	238 ± 3,2	470 ± 4,1	NA
GWT	16,1 ± 0,1	80 ± 2,1	80 ± 0,5	102 ± 2,7	199 ± 2,3	NA
Hibernate	13 ± 0,2	58 ± 1,7	57 ± 1,8	83 ± 1,9	146 ± 2	NA
Jackrabbit	10,4 ± 0,2	39 ± 0,5	38 ± 0,6	55 ± 0,7	113 ± 2,3	796 ± 152
JFreeChart	5,6 ± 0,2	21 ± 0,4	21 ± 0,4	30 ± 0,5	44 ± 1,2	277 ± 7,0
OpenEJB	10,6 ± 0,2	44 ± 0,8	43 ± 0,7	60 ± 0,8	117 ± 3,1	NA
Stendhal	4,4 ± 0,1	17 ± 0,5	17 ± 0,4	23 ± 0,4	36 ± 1,2	239 ± 11,7
SVNKit	4,4 ± 0,1	18 ± 0,3	18 ± 0,4	25 ± 0,5	39 ± 14	268 ± 7,7
Struts2	5,7 ± 0,1	23 ± 0,4	23 ± 0,4	32 ± 0,6	49 ± 1,1	292 ± 8,7
Tomcat	8,3 ± 0,2	33 ± 0,6	33 ± 0,6	43 ± 0,6	69 ± 1,7	484 ± 15,8
Weka	9,4 ± 0,2	38 ± 0,7	37 ± 0,3	52 ± 0,4	111 ± 2,4	526 ± 29,5
Xalan	4,8 ± 0,1	19 ± 0,3	19 ± 0,2	25 ± 0,3	38 ± 1,1	254 ± 9,4

(b) Memory Usage (in MB)

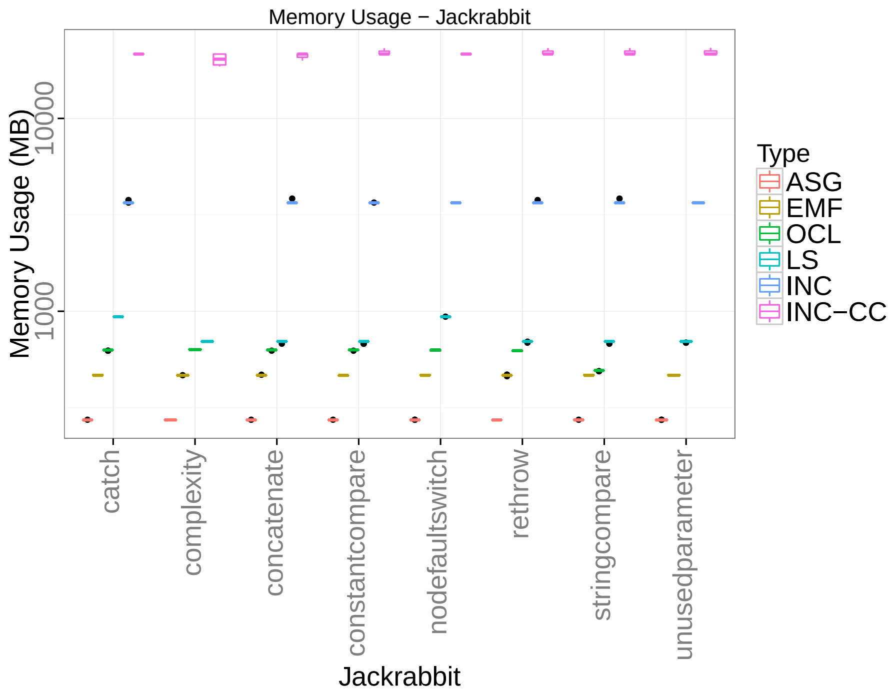
	ASG	EMF	OCL	LS	INC	INC-CC
CloudStack	2189 ± 0,47	3503 ± 1,39	3925 ± 38	4017 ± 2,7	10414 ± 58,88	NA
ArgoUML	198 ± 0,81	404 ± 0,9	461 ± 2,3	549 ± 9,1	5068 ± 42,09	11974 ± 841
Eclipse	2453 ± 0,66	4054 ± 1,87	4641 ± 3,9	4745 ± 1848	17754 ± 753,93	NA
GWT	2579 ± 0,12	1967 ± 2,49	2178 ± 2,9	3566 ± 1,3	5973 ± 32,93	NA
Hibernate	2086 ± 0,14	2524 ± 1,73	2788 ± 2,4	2995 ± 37,5	4507 ± 2,54	NA
Jackrabbit	309 ± 0,04	583 ± 4,62	651 ± 63	955 ± 9,8	3652 ± 59,45	22123 ± 1593
JFreeChart	160 ± 0,06	360 ± 2,18	429 ± 67	530 ± 82,6	4400 ± 0,34	10560 ± 273
OpenEJB	344 ± 0,26	656 ± 2,89	662 ± 82	946 ± 6,5	3889 ± 23	NA
Stendhal	109 ± 0,06	229 ± 0,51	431 ± 36	460 ± 124,2	3383 ± 68,85	7783 ± 629
SVNKit	129 ± 0,48	252 ± 3,12	401 ± 2,6	409 ± 2,8	3717 ± 4819	9835 ± 556
Struts2	159 ± 0,03	359 ± 2,71	479 ± 2,6	521 ± 2,9	4893 ± 70,27	11636 ± 180
Tomcat	246 ± 0,04	547 ± 6,05	601 ± 7,6	788 ± 66,7	6637 ± 64,05	16929 ± 2169
Weka	290 ± 0,07	616 ± 6,08	615 ± 151	695 ± 10,6	3427 ± 1	20357 ± 1377
Xalan	146 ± 0,59	260 ± 2,85	441 ± 1,7	445 ± 9	3600 ± 0,52	8259 ± 535

(c) Query Execution Time (in seconds)

		catch	cyclomatic complexity	concatenate	constant compare	no default switch	avoid rethrow	string compare	unused parameter	maximum deviation (%)
CloudStack	ASG	5,3	7,6	6,0	5,5	5,3	5,9	5,4	6,0	16%
	EMF	3,9	5,0	3,7	3,7	4,1	4,0	3,6	4,4	15%
	OCL	6,2	90,7	6,8	9,0	6,6	7,1	7,4	NA	6%
	LS	0,13	81,50	0,55	0,28	0,02	0,26	1,09	0,76	24%
	INC	0,012	NA	0,010	0,024	0,012	0,013	0,013	0,020	18%
ArgoUML	ASG	1,8	2,4	1,7	1,9	1,7	1,8	1,6	1,9	5%
	EMF	1,3	1,5	1,3	1,3	1,2	1,3	1,1	1,3	9%
	OCL	2,4	14,3	2,0	2,9	1,6	2,1	2,2	NA	11%
	LS	0,05	6,94	0,16	0,09	0,01	0,06	0,17	0,26	13%
	INC	0,012	0,011	0,010	0,013	0,012	0,012	0,012	0,012	13%
Eclipse	ASG	8,0	11,3	8,3	9,2	7,8	7,7	7,0	9,3	11%
	EMF	5,6	7,4	5,5	5,6	5,9	5,7	5,3	6,1	10%
	OCL	10,3	122,2	10,0	12,4	9,4	9,9	12,1	NA	3%
	LS	0,20	99,82	0,85	0,25	0,08	0,21	1,03	1,45	8%
	INC	0,010	NA	0,009	0,013	0,014	0,010	0,011	0,022	11%
GWT	ASG	5,2	11,2	5,4	5,1	6,9	5,9	5,4	6,4	24%
	EMF	3,0	3,9	2,8	2,8	2,9	2,9	2,8	3,2	8%
	OCL	4,7	37,5	4,6	5,8	4,0	4,6	4,6	NA	9%
	LS	0,05	29,15	0,47	0,15	0,03	0,10	0,53	0,39	4%
	INC	0,010	NA	0,009	0,012	0,012	0,011	0,011	0,013	7%
Hibernate	ASG	4,2	5,3	4,6	3,9	4,5	3,9	5,1	4,5	19%
	EMF	2,8	3,2	2,7	2,6	2,4	2,7	2,3	2,8	9%
	OCL	3,8	34,4	3,7	6,0	3,3	4,3	3,7	NA	10%
	LS	0,05	14,58	0,23	0,13	0,02	0,10	0,30	0,37	5%
	INC	0,011	NA	0,009	0,011	0,011	0,010	0,010	0,011	14%
Jackrabbit	ASG	2,8	3,6	2,8	2,8	2,7	2,8	2,6	3,0	4%
	EMF	1,8	2,3	1,8	1,8	1,8	1,8	1,6	1,9	6%
	OCL	2,9	24,1	2,9	4,1	2,6	3,3	3,2	NA	8%
	LS	0,10	50,93	0,26	0,10	0,04	0,13	0,32	0,36	36%
	INC	0,012	0,013	0,010	0,011	0,011	0,011	0,011	0,012	13%
JFreeChart	ASG	2,3	2,9	2,2	2,3	2,2	2,3	2,1	2,4	8%
	EMF	1,2	1,4	1,1	1,2	1,1	1,1	1,0	1,2	6%
	OCL	1,9	12,1	1,9	2,7	1,4	1,8	2,3	NA	6%
	LS	0,05	6,94	0,16	0,10	0,01	0,06	0,10	0,21	28%
	INC	0,009	0,010	0,010	0,012	0,012	0,010	0,012	0,012	23%
OpenEJB	ASG	2,6	3,5	2,6	2,8	2,5	2,6	2,4	2,9	3%
	EMF	1,8	2,3	1,8	1,9	1,9	1,9	1,7	2,0	9%
	OCL	2,9	20,4	2,8	3,9	2,3	3,4	3,1	NA	7%
	LS	0,12	12,40	0,25	0,11	0,02	0,17	0,36	0,36	32%
	INC	0,012	NA	0,009	0,012	0,011	0,011	0,011	0,013	16%
Stendhal	ASG	1,6	2,1	1,7	1,8	1,6	1,7	1,5	1,9	9%
	EMF	0,9	1,2	1,0	1,0	1,0	1,0	0,9	1,0	2%
	OCL	1,5	10,1	1,7	2,2	1,2	1,5	1,8	NA	3%
	LS	0,03	4,64	0,16	0,09	0,01	0,05	0,19	0,23	20%
	INC	0,010	0,011	0,010	0,013	0,012	0,010	0,012	0,012	14%
SVNKit	ASG	1,6	1,9	1,6	1,7	1,6	1,6	1,5	1,8	7%
	EMF	1,0	1,2	1,0	1,0	1,0	1,0	0,9	1,0	9%
	OCL	1,5	13,3	1,8	2,2	1,2	1,6	2,3	NA	6%
	LS	0,06	9,49	0,16	0,06	0,01	0,09	0,19	0,25	18%
	INC	0,012	0,012	0,010	0,012	0,011	0,011	0,010	0,012	16%
Struts2	ASG	2,2	2,9	2,2	2,3	2,3	2,2	2,0	2,4	7%
	EMF	1,2	1,5	1,2	1,2	1,2	1,2	1,1	1,3	4%
	OCL	2,0	12,7	2,0	2,8	1,5	1,9	2,1	NA	7%
	LS	0,05	7,09	0,15	0,08	0,02	0,07	0,23	0,26	16%
	INC	0,012	0,011	0,010	0,011	0,012	0,011	0,011	0,013	14%
Tomcat	ASG	2,3	3,0	2,4	2,4	2,4	2,4	2,2	2,6	7%
	EMF	1,5	2,0	1,5	1,5	1,5	1,5	1,3	1,6	15%
	OCL	2,6	21,3	2,5	3,3	2,1	2,7	3,1	NA	8%
	LS	0,08	13,48	0,23	0,10	0,02	0,13	0,33	0,31	16%
	INC	0,013	0,012	0,011	0,013	0,012	0,012	0,013	0,012	24%
Weka	ASG	3,2	4,3	3,2	3,3	3,1	3,2	3,0	3,4	5%
	EMF	1,7	2,2	1,7	1,7	1,7	1,7	1,5	1,8	4%
	OCL	2,8	26,2	3,1	3,6	2,3	2,8	3,2	NA	7%
	LS	0,06	21,46	0,27	0,09	0,02	0,09	0,39	0,33	23%
	INC	0,010	0,013	0,009	0,011	0,010	0,010	0,011	0,011	11%
Xalan	ASG	1,7	2,0	1,7	1,7	1,6	1,7	1,6	1,8	9%
	EMF	1,1	1,3	1,1	1,2	1,1	1,1	1,0	1,2	8%
	OCL	1,9	12,5	1,8	2,2	1,3	1,8	2,2	NA	3%
	LS	0,05	11,32	0,16	0,07	0,02	0,08	0,21	0,24	3%
	INC	0,011	0,011	0,010	0,013	0,012	0,012	0,011	0,013	11%



(a) Load Time (in seconds)



(b) Memory Usage (in MB)

Figure 5.8. Distribution of Load Time and Memory Usage of the Jackrabbit Project

A similar increase can be seen for the memory usage in Figure 5.4b. Here, the EMF representation uses about twice as much memory, while the incremental engine may require an additional 10–15 times more memory to store its partial result caches compared with the ASG. When adding the *cyclomatic complexity* pattern as well, an additional increase in memory usage is observed, resulting in a memory exhaustion for the largest models (over 500kLOC, or 1.7M graph nodes).

The smaller memory footprint of the Java ASG representation is the result of model-specific optimizations that are not applicable in generic EMF models. The additional increase for local search and Rete-based pattern matchers is mostly due to the index and partial match set sizes, respectively. Like load times, the use of OCL does not result in a change in memory usage compared with the EMF model.

The memory footprint increase of the *cyclomatic complexity* pattern is probably caused by the indexing of the transitive closure of the parent relation. As every model element has a parent and the containment hierarchy is usually deep, this transitive closure may alone become several times the size of the entire model, making it very expensive to index. Despite this, the containment hierarchy can be effectively traversed using search operations, hence the other approaches can handle this query much better.

Generally speaking, neither for load times nor memory usage was the standard deviation of the results significant compared with the other values, with the notable exception of the load time of the Jackrabbit tool with INC-CC, and the SVNKit applications memory usage with INC. The first one can be explained with garbage collection, as the memory usage was close to the 25 GB limit. For the latter, we have no clear explanation; however as we have witnessed no other fluctuations of this size, we think that it was caused by a temporary issue that occurred during our measurements.

5.6.2 Search Time

Figure 5.4c presents the search time measurements (and uses *NA* if the measurement timed out). For each model and each program query the average search time is listed first. Furthermore, in Figure 5.9, we highlighted the results of the Jackrabbit project in a box plot, where there are only minimal differences between any two different executions of the same case, similar to load and search times.

Both visitor implementations performed similarly, producing similar execution times for queries, but they increased with model size as they traverse the entire model to find the results. The time differences between the ASG and EMF visitors were mainly the results of the memory optimizations of the original ASG implementation that avoided storing the same values multiple times, but required additional indirections during the model traversal. The reverse navigation option is not used in our measurements.

The local search and Rete based solutions provide a two-to-three orders of magnitude faster query execution, achieved by replacing the model traversal by calls to a pre-populated (and incrementally updated) index. Also, the search time of incremental queries is largely independent of model size, while in the case of local search it increases more slowly than in the case of the visitor executions. As the search times for INC queries were exactly the same regardless whether the *cyclomatic complexity* query was loaded or not, their rows were merged in the table.

The execution of OCL queries include a traversal of the model together with additional search operations, making the search slower than the visitor implementations. An exception to this is the unused parameter query: in this case the search opera-

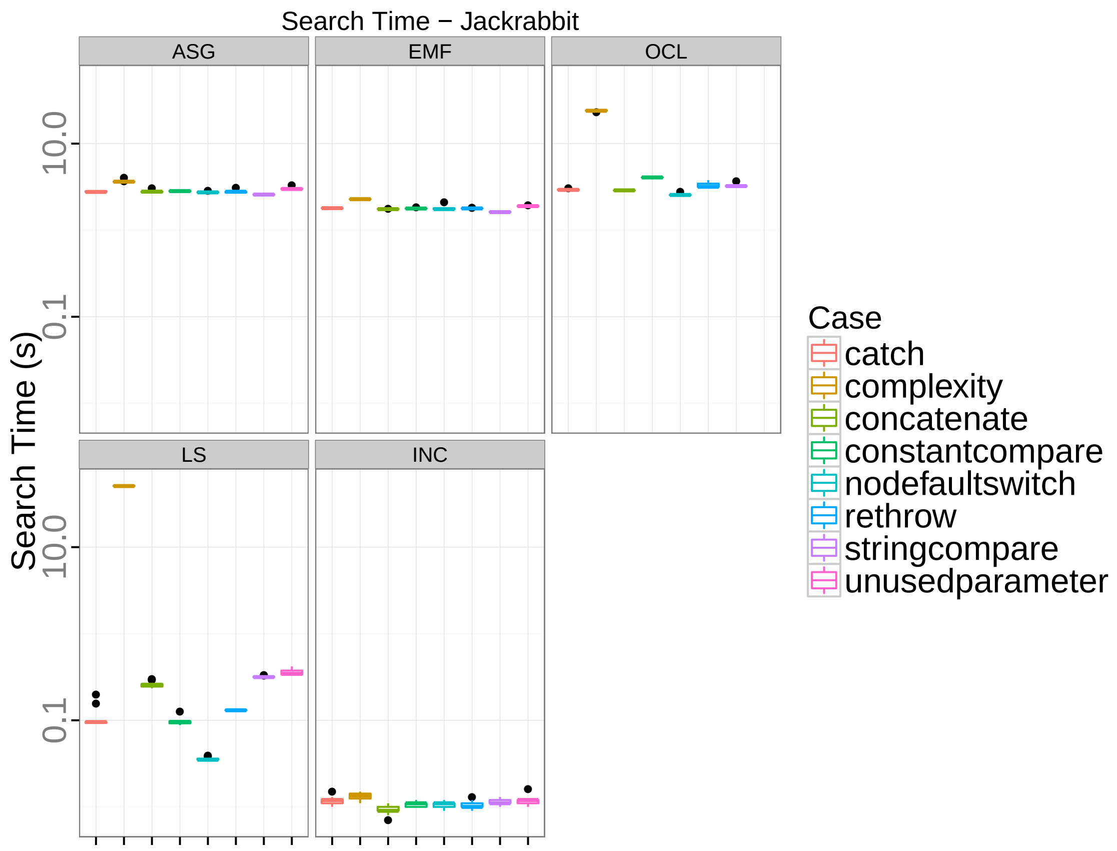


Figure 5.9. Distribution of Search Times for the Jackrabbit Project

tion timed out every time. This is most likely caused by the usage of the *allInstances* function that is used to find the source of an edge without reverse navigation options.

In addition, as seen in Figure 5.4c, the execution time of visitor implementations increases linearly. This is in line with our expectations, as visitors have to traverse the entire model during the search. In spite of this, the search time for incremental queries are roughly the same for all queries, as the search simply means returning the results. In most of our patterns, the local search is an order of magnitude slower than incremental queries. However, the concatenation pattern (see Figure 5.3c) runs just as slow as the visitors in this regard. This is in line with our earlier experience [132] with different pattern matching strategies that the execution performance for local search techniques depends on the query complexity and the model structure.

To validate the results, for each program and tool combination we have the maximum standard deviation in percentage terms of their corresponding search time. In most cases, the standard deviation is low; only 9 rows contain deviations over 20%. As our measurements have revealed time differences of orders of magnitude, these differences do not invalidate our conclusions drawn from the analysis.

5.7 Evaluation of Usage Profiles

Following the evaluation of the raw measurement data, we will now explain how the different approaches were compared in various usage profiles, and we will summarize our findings. Then we will discuss the different threats to validity, and the ways they were handled.

5.7.1 Usage Profiles

In order to compare the approaches, we calculated the total time required to execute program queries for three different *usage profiles*, namely one-time, commit-time, and save-time analysis. The profiles were selected by estimating the daily number of commits and file changes for a small development team.

One-time analysis consists of loading the model and executing each program query in a batch mode. In case the analysis needs to be repeated, the model is reloaded. In our measurements, this mode is represented by a *load* operation followed by a single *query* evaluation.

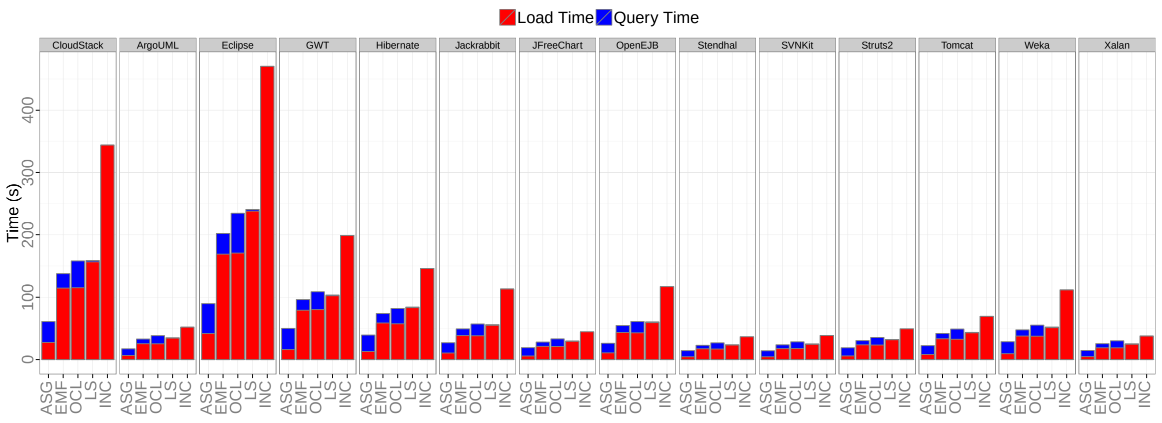
Commit-time analysis can be used in a program analysis server that keeps the model in-memory, and on each commit, it is updated as opposed to be reloaded, and then it re-executes all queries. In our case, this mode is represented by a *load* operation followed by 10 *query* evaluations.

Save-time analysis is executed whenever the programmer saves a file in the IDE, and then the IDE either executes the analysis itself, or notifies the analysis server. It is similar to commit-time analysis, but it is executed more often. In our measurements, this mode is represented by a *load* operation followed by 100 *query* evaluations.

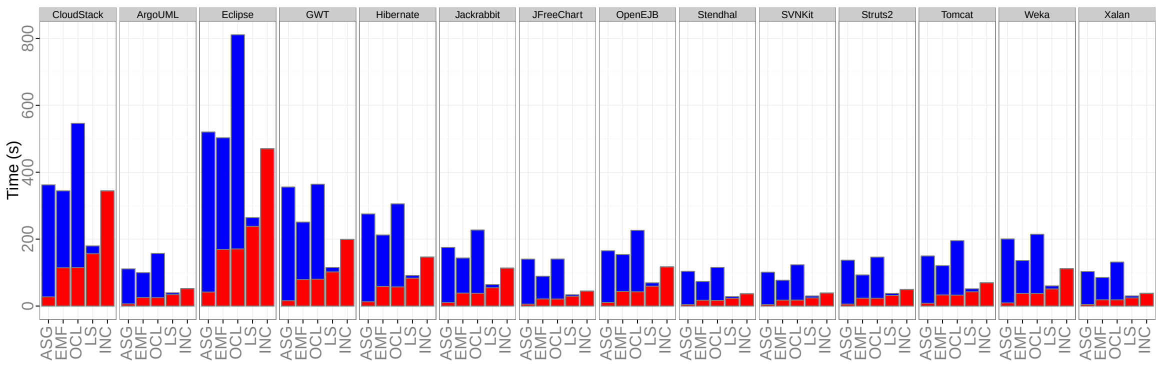
5.7.2 Usage Profile Analysis

We calculated the execution times for the search profiles for all the projects by considering the time to load the models (Figure 5.4a), and increasing it by 1, 10 and

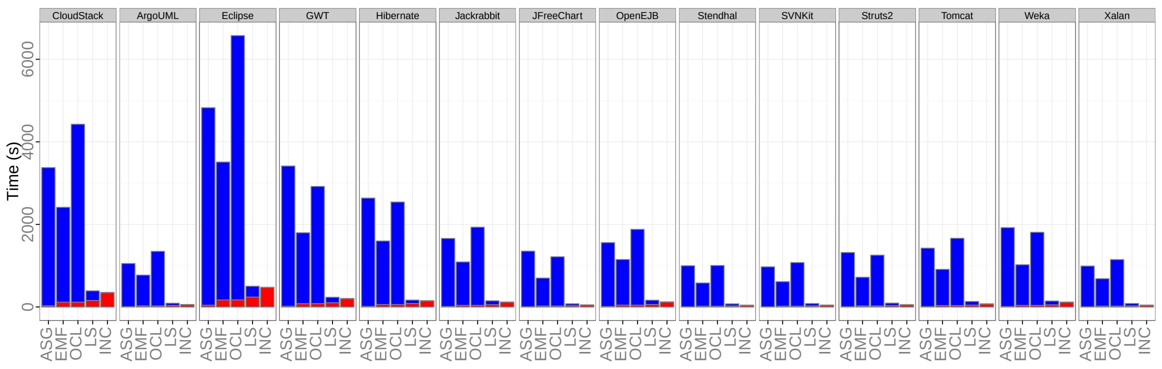
100 times the search time of six queries one after another, respectively. As the *unused parameter* and *cyclomatic complexity* query could not always be executed in OCL and the incremental matcher, respectively, to keep the results comparable, they were excluded from this calculation.



(a) One-time



(b) Commit time



(c) Save time

Figure 5.10. Execution Time over Models

Figure 5.10 shows our measured values for total execution times on the various usage profiles from two points of view. We included detailed graphs for the selected models where load times and query times can be observed (note the differences in the time axis).

The results indicate that albeit the *visitor* approaches execute queries more slowly, as there are no additional data structures initialized, the lower load time makes this approach very effective for one-time, batch analysis. However, as all visitors are implemented separately, as to execute all of them would require six model traversals; reducing this would provide a further time advantage of this solution over the local search based ones. This issue could be managed by combining all the queries in a single visitor, thus increasing its complexity. Still, visitors behave worse regarding the run time in the case of a repeated analysis: the mean time for executing 100 searches increased from 32 to 1967 seconds for the ASG-based implementation (and from 62 to 1257 when executed over EMF).

OCL queries behave in a similar way to visitor-based searches. Here, no indexing is used, but the model is traversed during search. Executing a single query is more expensive than executing a single visitor, and during the measurements nothing is shared between the different executions, making the mean one-time execution time of the six queries 71 seconds (almost the same as the result of the local search based pattern matcher), repeating it a hundred times is done in 2204 seconds (slower than the ASG version). However, selecting an OCL execution mode that evaluates multiple OCL queries during a single traversal where possible might significantly reduce the total search time, and help make this approach a viable alternative to hand-coded visitors.

The *local search* based approach is noticeably faster than visitor-based solutions with memory usage and initialization time penalties introduced by the use of caching. The mean execution times range from 69 to 171 seconds. These properties make the approach work very well in the Commit-time analysis profile, and other profiles with a moderate amount of queries. However, if a bad search plan is selected for a query, such as in the case of the Concatenation to Empty String pattern, its execution time may become similar to the visitor-based implementations.

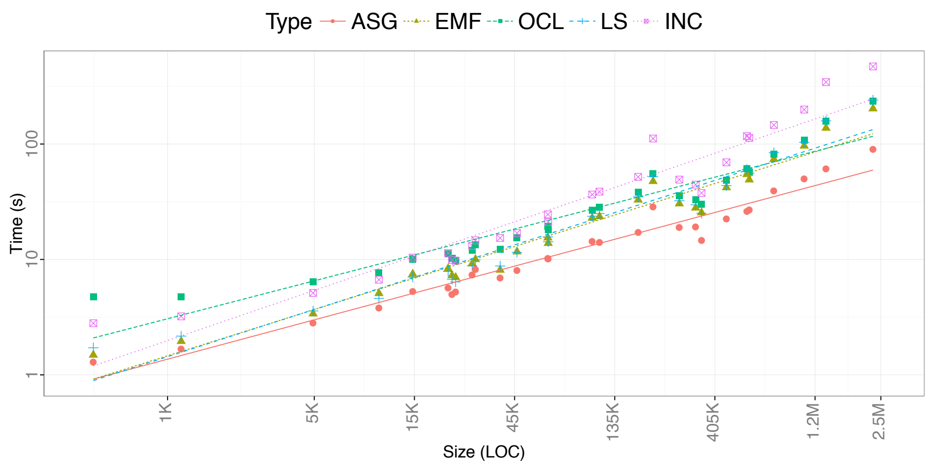
The *incremental*, Rete-based pattern matching approach provides instantaneous model query times, as the results are always available in a cache. This makes such an algorithm powerful for repeatedly executed analysis scenarios, such as the Save-time analysis profile (mean time: 131 seconds, the lowest from all approaches). However, to initialize the caches, a lengthy preparatory phase is required and it makes this technique the slowest for one-time analysis scenarios (mean time: 394 seconds).

If the save-time analysis profile is used and the *required memory* of the incremental approach cannot be met, the complementing local search matcher can be used and it still has a performance advantage over the visitor-based solutions. Also, by moving the analysis to a distributed, cloud-based system, it is possible to manage even larger models using the incremental approach [133].

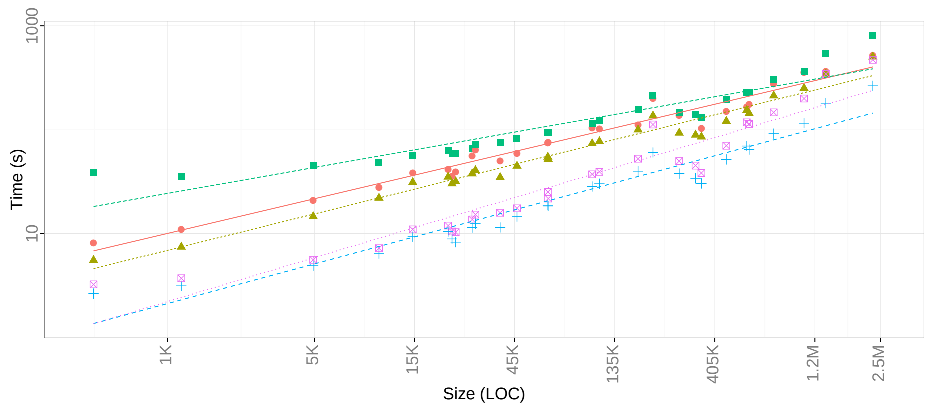
Next, we evaluated how execution times varied when increasing the model size. Figure 5.11 shows the analysis time using different tools over the model size in each usage profile and it adds linear trend lines to compare the rate of increase. We found that our findings were consistent over different models: regardless of the model size, the same relative ordering can be observed in the case of each profile.

5.7.3 Lessons Learned

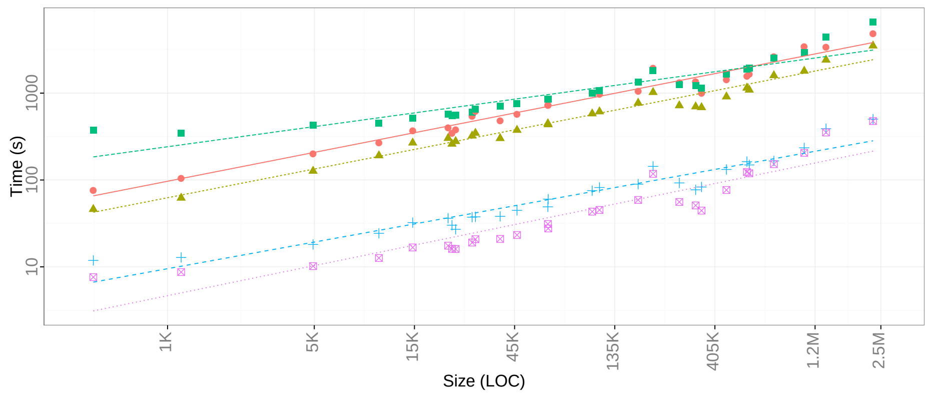
From a memory consumption perspective, the manually optimized ASG excels while providing a fast query execution for the one-time usage profile. However, a generic model implementation, such as EMF, may be a viable alternative when additional



(a) One-time Usage Profile



(b) Commit time Usage Profile



(c) Save time Usage Profile

Figure 5.11. Execution Time with regards to Model Sizes

features of these frameworks are used and the doubled memory usage is acceptable. Furthermore, the use of generic model implementations means generic query approaches can become an alternative for manually coded searches based on usage profiles:

- Batch solutions, such as the Eclipse OCL implementation have minimal additional memory requirements while their performance is similar to that of manually written visitors.
- Full incremental solutions, such as the Rete-based pattern matcher of the EMF-IncQuery, provide results instantaneously even after model changes, making it beneficial for recurring queries and evolving source code, to meet their memory requirements.
- The local search implementation of the EMF-IncQuery uses an incremental indexer to speed up search implementations, achieving query evaluation times that are orders of magnitude faster than non-indexed solutions, but with a lower memory consumption. This result is in line with the idea of hybrid pattern matching [132], where incremental and search-based approaches complement each other for better performance characteristics.

Both the OCL and the graph pattern formalism provide a higher-level specification of program queries, resulting in a more compact query description compared to manually coding visitors, and in our subjective experience, they are easier to understand and reduce query development time. Advanced features, such as the computation of transitive closures, are also supported, further reducing the length of query descriptions.

Regardless of the modeling technology, optimizing the queries, either for performance or memory consumption, may require a deep understanding of the behavior of the underlying algorithms. In some cases, this means a complete reformulation of the query. For instance, in the case of the *catch problem*, the pattern description requires an inverse navigation between the catch parameters and its references, while the visitor implementation traverses the containment subtree instead.

We have also identified cases where one of the selected tools works noticeably better or worse than the other candidates:

- If inverse relations are not modeled, some queries in OCL cannot be implemented efficiently (e.g. without iterating all instances of a type). Not surprisingly, adding the inverse relations increases the memory usage of the model.
- Navigating the containment hierarchy (especially transitively) requires a huge amount of memory with the Rete-based incremental approach, as it requires storing many model element-ancestor pairs in the memory.
- Visitor-based solutions can very effectively traverse the containment hierarchy. In the case of the *cyclomatic complexity* calculation, this is the main reason why the visitor implementations outperform all the others.

In addition, as a rule of thumb, we have created a simplified representation (see Figure 5.12) based on the lessons we learned from the results in a form of a decision model to choose the more suitable tools for the different usage scenarios. The figure servers as a supplementary guide to aid the understanding of our observations above, but it is not a complete presentation of our results.

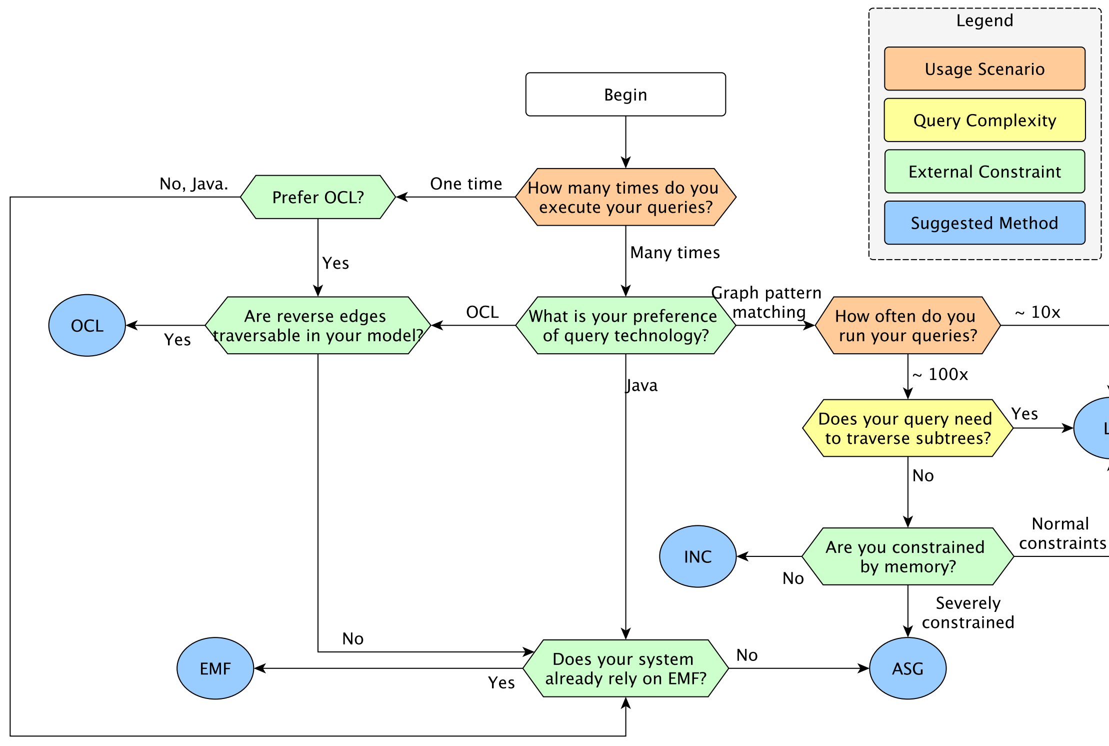

Legend
Usage Scenario
Query Complexity
External Constraint
Suggested Method
Begin
How many times do you execute your queries?
One time
Many times
Prefer OCL?
No, Java.
Yes
Are reverse edges traversable in your model?
Yes
OCL
No
What is your preference of query technology?
OCL
Java
Graph pattern matching
How often do you run your queries?
~ 10x
~ 100x
Does your query need to traverse subtrees?
Yes
No
LS
Are you constrained by memory?
Normal constraints
No
Severely constrained
INC
Does your system already rely on EMF?
Yes
No
EMF
ASG

Figure 5.12. Decision Model (Simplified Representation)

In the refactoring project, the implemented FaultBuster refactoring framework (presented in Chapter 4) applies the one-time scenario as the usage scenario was planned for the ASG which does not support incremental model updates. In addition, a large, 4M LOC proprietary program has been refactored, so the decision during this project was to keep the ASG and the one-time approach. From this research, we may conclude that generic solutions are viable alternatives and by using an incremental tool setup a huge performance gain can be achieved when sufficient memory is available.

5.7.4 Threats to Validity

We have identified several validity threats that can affect the construct, the internal and external validity of our results. The first is the low construct validity.

Low *construct validity* may threaten the results of various usage profiles, as the results do not include the time required to update the indexes and Rete networks on model changes. However, based on the previous measurement results related to EMF-IncQuery [128], we think that such slowdowns are negligible in cases where the change size is small compared to that of the model.

Furthermore, in the case of very large heap sizes (over 10 GB) the garbage collection of JVM instances may block the program execution for several minutes in a non-deterministic way. To make the measurements reproducible, the JVM instances were allocated their maximum heap size during startup instead of gradually extending it as needed.

We tried to mitigate *internal validity* threats by comparing the measurements when we changed only one measurement parameter at a time. For example, the EMF implementation of the Java ASG makes it possible to differentiate between the changes caused by different internal model representations by comparing the different model representations using the same search algorithm first, then we compare the EMF-based visitor with generic pattern matching solutions.

An important threat in a study to compare various methods is that the evaluation is carried out through actual implementations. The decisions in the implementation may affect the overall outcome and the judgement of the choice of method. To reduce this threat, the implementation can be performed by experts of the given technologies. Hence, the same query is implemented in a slightly different way in each method depending on the features of the methods like the availability of reverse edges.

Note that the authors are not experts of the OCL tools, and, the metamodel itself does not favor the structure expected by OCL. However, as we have found that OCL performs comparably to the visitor-based implementations, it is clearly a viable alternative to manually coded searches.

As regards *external validity*, the generalizability of our results largely depends on whether the selected program queries and models are representative for general applications. The queries were selected prior to the projects and scenarios. These refactorings were emphasized by project partners and were selected to cover several aspects of transformations.

The selected open-source projects differ in size and characteristics – including computational intensive programs, applications with heavy network and file access and with a graphical user interface. Moreover, the projects were selected from the testbed of the Columbus Java static analyzer and ASG builder program where the aim was to cover a wide range of Java language constructs.

As for projects from different programming languages, they require a corresponding metamodel and instance models. The Columbus framework itself provides metamodels and code analyzers for creating these models for various languages, such as C/C++, C# or RPG, and these metamodels can be ported similarly to the EMF. However, a further evaluation may be needed to validate whether the results still hold, as the properties of these program models may differ significantly.

Another issue is the selection of model query tools. Although several other tools are available, based on the results of over a decade of research on efficient graph pattern matching techniques, we think that other pattern matcher tools should provide similar results to either our local search or incremental measurements.

In our work, we used Java-based tools and the EMF framework so that the results of the tools could be compared. Despite this, the investigated tools support additional languages. For example, the Columbus API is available in C++, and OCL tools are available for different modeling formalisms and languages. The EMF-IncQuery framework has been implemented in Java and focuses on EMF models; however the language and runtime are also being adapted to different formalisms such as RDF and the metamodeling core of MPS.

OCL queries expect that a context object will be selected from the environment and expressions can be evaluated from this point. However, the standard does not specify how to select this context object, and different OCL tools support varying query execution modes. Such modes include the Impact Analyzer of the Eclipse OCL tool [131], which tracks model changes and just recomputes those results that rely on the modified model elements; or the model invariant formulation that can evaluate multiple boolean queries in parallel. In order to be able to measure the execution times of single queries, we selected all possible context objects by traversing the entire source model. To evaluate the effects of choosing a different context selection strategy or execution mode, additional measurements are needed.

Overall, our results were similar for all the models and queries, so we think our results should generalize well to other program queries and models, as far as the memory requirements of indexing or Rete building are met.

5.8 Related Work

In our comparison, we evaluated solutions that are specific to program models and generic methods not restricted to the domain of program models. Now, we present related research in two groups starting from generic to program model-specific solutions.

5.8.1 Software Analysis Using Generic Modeling Techniques

Program queries are a common use case for modeling and model transformation technologies including transformation tool contests. The program refactoring case of the GraBaTs Tool Contest 2009 [134] and the program understanding case of the Transformation Tool Contest 2011 [135] rely on a program query evaluation followed by some transformation rules, focusing on the applicability of modeling tools for refactoring and reverse engineering. In 2011, six tools were entered in the contest (GreTL, VIATRA2, Edapt, MOLA, GrGen.NET and Henshin), some of them were EMF-based, others relied on a different metamodeling approach, and for each tool the tasks were executed in a few seconds (albeit sometimes after costly model import operations). This work extends these results by comparing the costs of using generic modeling environments to manually optimized refactoring models; and extends the performance comparisons with a larger pool of real-world software models and the use of different model queries.

The refactoring case was reused in [136] to select a query engine for a model repository, but, its performance evaluations did not consider incremental cases.

A series of refactoring operations were defined as graph transformation rules by Mens et al. [137], and they were also implemented for both the Fujaba Tool Suite and the AGG graph transformation tools. Although the study presents the graph transformations that are useful as an efficient description of refactoring operations, no performance measurements were included. The Fujaba Tool Suite was also used to find design pattern applications [138]. As a Java model representation, the abstract syntax tree of the used parser generator was used, and the performance of the queries were also evaluated.

The Java Model Parser and Printer (JaMoPP) project [139] provides a different EMF metamodel for Java programs. It was created to directly open and edit Java source files using EMF-based techniques, and the changes were written back to the original source code. Despite this, the EMF model of the JaMoPP project does not support any existing model query or refactoring approaches, and every program query or refactoring is to be reimplemented to execute it over the JaMoPP models. This approach was used in [140], and it relies on the Eclipse OCL tool together with a display of the identified issues in the Eclipse IDE.

The EMF Smell and EMF Refactor projects [141] allow one to find design smells and execute refactorings over EMF models based on the graph pattern formalism. As Java programs can be translated into EMF models, this also permits the definition and execution of program queries.

One key difference between our experiment and the above-mentioned related studies is that we compare the performance characteristics of hand-coded and model-based query approaches.

When comparing the performance of the different approaches, an additional factor needs to be considered. Namely, as there are multiple different (sometimes not even EMF-based) metamodels used to describe Java applications, further measurements are required to evaluate the effects of a metamodel selection. However, we think that our test setup is general enough to handle the large set of tools, approaches and queries proposed by these studies.

The train benchmark described in [128] concentrates on on measuring the performance of incremental model query approaches. It relies on synthetic models scalable to any model size, and defines both query and model manipulation steps to measure the real impact of query re-evaluation. The author of [129] attempted to predict the query evaluation performance based both on metrics of models and queries. In our work, we applied these metrics on real-world models to evaluate the query engine instead of synthetic models, and while our results were quite similar, a more detailed comparison is required to analyze their usefulness.

5.8.2 Software Analysis Designed for Program Models

Several tools are available for detecting coding issues in Java programs. The closest solutions to our ASG+Visitor method are, for example, the PMD checker [27] and FrontEndART's FaultHunter [142], which in fact is built on the top of the Columbus ASG. These applications can be integrated into IDEs as plug-ins, and they can be extended with the searches implemented in Java code or in a higher level language, such as XPath queries in PMD. PMD provides rules for a great variety of coding problems, but the given model and query API is not as flexible as the solutions used in this research. The main usage scenario of these tools is to run the checkers once on (any version of) the source code and find coding issues. Unfortunately, they do not support incremental model updates yet.

In contrast to generic solutions, there are several systems that support (meta) modeling and querying especially program models. FAMIX [143] is a language-independent meta-model for representing procedural and object-oriented code, used in the Moose reverse engineering environment [144]. The MOOSE environment provides query possibilities in Smalltalk. The authors claim that their approach is not Smalltalk specific and it can be applied Java as well. The Rascal [145] metaprogramming language is designed for source code analysis and manipulation. Its analysis features are based on relational calculus, relation algebra and logic programming systems. Its tool support includes an Eclipse-based IDE, and the language provides Java integration. For any task not (readily) expressible in RASCAL, one may use Java method bodies inside Rascal functions. These solutions use their own meta model to represent Java programs, unlike solutions in our study, where the Columbus meta model is used via the EMF. Nevertheless, these tools are candidates for comparative research in the future.

In addition, several approaches allow one to define program queries using logical programming, such as the JTransformer [146] using Prolog clauses, the SOUL approach [147] that relies on logic metaprogramming, and CodeQuest [148], which is based on Datalog. However, none of these offer a comparison with hand-coded query approaches. The DECOR methodology [149] provides a high-level domain-specific language for evaluating program queries. It was evaluated in terms of performance on 11 open-source projects, including the Eclipse project. It took around one hour to find its defined smells. These results are difficult to compare to ours, as the evaluated queries are different (and some of them are more complex than the ones defined here), but they are described in enough detail to extend our environment. However, evaluating the effects of representation and tool selection is problematic, as neither the model representation, implementation structure nor the used programming language is shared between the different approaches.

A key advantage of our approach is the ability to select the query evaluation strategy based on the required usage profile. Additionally, it is possible to re-use the existing program query implementations while using a high-level, graph pattern-based definition for the new queries.

5.9 Summary

In this chapter, we evaluated different query approaches for locating anti-patterns for refactoring Java programs. In a traditional setup, an optimized Abstract Semantic Graph was built by SourceMeter, and it was processed by hand-coded visitor queries. In contrast, an EMF representation was built for the same program model which has various advantages from a tooling perspective. Furthermore, anti-patterns were identified by generic, declarative queries in different formalisms evaluated with an incremental and a local-search based strategy.

Our experiments that were carried out on 28 open source Java projects of varying size and complexity demonstrated that encoding ASG as an EMF model results in an up to 2-3 fold increase in memory usage and an up to 3-4 fold increase in model load time, while incremental model queries provided a better run time compared to hand-coded visitors with a 2-3 order of magnitude faster execution, at the cost of an additional increase in memory consumption by a factor of up to 10-15. Following this, we provided a detailed comparison of the different approaches and this made it possible to select one over the other based on the required usage profile and the expressive capabilities of the queries.

To sum up, we emphasize the expressiveness and concise formalism of pattern matching solutions over hand-coded approaches. They offer a quick implementation and an easier way to experiment with queries together with different available execution strategies. However, depending on the usage profile, their performance is comparable even with 2,000,000 lines of code.

"You can't connect the dots looking forward;
you can only connect them looking backwards."
— Steve Jobs

6 Conclusions

In this thesis we discuss different topics to support the 'continuous refactoring' of software systems. Now, we shall summarize our contributions and draw some pertinent conclusions. We will answer our research questions and elaborate on the main lessons we learned.

6.1 Summary of the thesis contributions

In general, the results presented indicate that refactoring should and can be automated via computer assistance. Developers are receptive to tools that suggest refactoring opportunities. They welcome tools even more when these are capable of providing solutions as well. We also showed that refactoring is a good practice in programming, and when performed continuously it has beneficial effects on measurable software maintainability. We provided a detailed comparison of different approaches to locate anti-patterns for refactoring Java programs. In addition, we identified several challenges of implementing an automated refactoring tool. Our recommendations may serve as a guideline for others who face similar challenges when designing and developing automatic refactoring tools that meet the high expectations of today's developers.

We should add here that the studies presented in the study are closely connected to the Refactoring Research Project. The project provided us with a good opportunity to do research in real-life industrial environment. This motivated us to carry out studies on more practical topics. The project spawned a lot of research papers over the years. Many of them were presented at international conferences, including the one ([8]) that won the best paper award at the IEEE CSMR-WCRE 2014 Software Evolution Week, Antwerp, in Belgium, February 3-6, 2014.

Now, we will restate our initial research questions and answer them one by one.

RQ1: What will developers do first when they have given the time and money to do refactoring tasks?

Our studies contain valuable insights into what developers do. Throughout our experiments, we collected 1,273 refactorings in the manual phase and at the end of the automatic phase we got about 6,000 tool-aided refactorings. We observed developers in a large, *in vivo*, industrial context while doing hand-written and automatic refactoring tasks.

In Chapter 3, we found that developers tend to fix coding rule violations more often than anti-patterns or metric value violations. We learned that they optimized the refactoring process and started fixing more serious issues first. Although keeping priority a concern, they kept choosing those issues which were easier to fix. Our interviews with the developers and our analysis of the evolution of their system revealed that by the end of the project developers had learned to write better code.

We continued our research in Chapter 4 by providing developers with an automatic refactoring tool. We learned that they are optimistic about automation and they thought that automated solutions could increase their efficiency. Developers thought that most of the coding issues could be easily fixed via automated transformations. This trust manifested itself when we noticed that sometimes they just blindly applied the automatic refactorings without taking a closer look at the proposed code modification. It happened several times that the automatic refactoring tool asked for user input to be able to select the best refactoring option, but developers used the default settings because it was easier. Partners were generally satisfied with the automated refactoring solutions and they enthusiastically asked us to extend its support with new types of fixable coding issues. We found that their most loved feature was batch refactoring – where they could fix several issues at once – because it greatly increased their productivity.

RQ2: What does an automatic refactoring tool need to meet developers requirements?

The manual phase of the Refactoring Project told us that developers seek to fix coding issues. We also learned that developers do not like switching between their normal development activities and a refactoring tool, and therefore the tool has to be integrated into their IDEs. Our results suggest that one of the most important features of a refactoring tool is to provide refactoring recommendations (i.e. what to refactor and how). This requires precise problem detection to avoid false positive matches. We found that the refactoring transformations have to be transparent and well documented because we noticed that developers tended to use simpler refactorings because they lacked the understanding of more complex ones (e.g. clone extraction). An interesting find was that the partner companies often demanded different solutions for the same coding issue. This tied in with developers requests to allow some parametrization of refactoring algorithms. To fulfill the latter two requirements a fully-automated method did not suffice. Instead, a semi-automatic solution was necessary. Besides the control over the refactoring algorithms, developers wanted to have a decision in the end as well, whether to accept or reject the suggested fix. Here, developers can compare the original and the refactored version of the code; and they can run unit and integration tests on the system before accepting a fix. What is more, what developers would like from the refactoring transformation is to use correct code formatting, identification, and to modify it as little code as possible. They also asked for comment handling, such as removing comments with remove method refactoring.

Based on the former guidelines, in Chapter 4, we introduced FaultBuster, an automatic refactoring toolset. FaultBuster has two special properties that makes it unique among other tools. First, it is designed as a server-client refactoring framework which has built in issue management that ensures that no issues are fixed by different developers at the same time. Second, it allows programmers to fix multiple coding issues at once, in so-called batches. FaultBuster's main target is coding rule violations and code smells. Under the hood, it uses a well-defined automated refactoring process to perform transformations on the program model. This model includes the Reverse AST-search Algorithm which maps coding issues to source code elements.

RQ3: How does manual and automatic-tool aided refactoring activity affect software maintainability?

We identified lots of refactoring commits throughout the project. First, it was 315 in the manual phase (Chapter 3) and later, 1,048 in the automatic phase (Chapter 4). By employing the QualityGate SourceAudit tool (which implements the ColumbusQM quality model), we analyzed the maintainability changes induced by the different refactoring tasks. By measuring the maintainability of the involved subject systems before and after the refactorings, we got valuable insights into the effect of these refactorings on large-scale industrial projects.

We learned that the outcome of one refactoring on the global maintainability of the software product is hard to predict; moreover, it might sometimes actually have a detrimental effect. Generally speaking, though a whole refactoring process can have a significant beneficial effect on the measurable maintainability. We found that fixing anti-patterns have larger positive effect on quality than fixing either coding issues or metric values. In addition, our study shed light on some important aspects of measuring software maintainability. Some of the unexpected effects of refactorings (like the detrimental effect of removing unnecessary constructors on maintainability) are caused by the special features of the maintainability model applied.

Our results do not suggest significant differences between manual and automatic-tool aided refactoring activity from the maintainability point of view. If refactoring is a way to either software heaven or hell, automated refactoring is just a faster way of getting there.

RQ4: Can we utilize graph pattern matching to identify anti-patterns as the starting point of the refactoring process?

In Chapter 5, we investigated the costs and benefits of using the popular industrial Eclipse Modeling Framework (EMF) as an underlying representation of program models processed by four different general-purpose model query techniques based on native Java code, OCL evaluation and (incremental) graph pattern matching. We provided an in-depth comparison of these techniques on the source code of 28 Java projects using anti-pattern queries taken from refactoring operations in different usage profiles.

Our main finding is that advanced generic model queries over EMF models can run several orders of magnitude faster than dedicated, hand-coded techniques. However, this performance gain is offset by an up to 10-15 fold increase in memory usage (in the case of full incremental query evaluation) and an up to 3-4 fold increase in the model load time for EMF based tools and queries, compared to native Columbus results. Hence the best strategy should be planned in advance, depending on how many times the queries should be evaluated after loading the model from scratch. This is why, any of these four techniques is sufficient for creating an anti-pattern detection tool that is capable of identifying refactoring suggestions.

Bibliography

Corresponding publications of the Thesis

[1] Gábor Szőke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. "A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve Source Code Quality". In: *Computational Science and Its Applications - ICCSA 2014 - 14th International Conference, Guimarães, Portugal, June 30 - July 3, 2014, Proceedings, Part V*. 2014, pp. 524–540. DOI: 10.1007/978-3-319-09156-3_37. URL: http://dx.doi.org/10.1007/978-3-319-09156-3_37.

[2] Gábor Szőke, Gabor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. "Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?" In: *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. 2014, pp. 95–104. DOI: 10.1109/SCAM.2014.18. URL: http://dx.doi.org/10.1109/SCAM.2014.18.

[3] Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. "Empirical Study on Refactoring Large-Scale Industrial Systems and Its Effects on Maintainability". In: *Journal of Systems and Software* (2016). ISSN: 0164-1212. DOI: http://doi.org/10.1016/j.jss.2016.08.071. URL: http://www.sciencedirect.com/science/article/pii/S0164121216301558.

[4] Gábor Szőke, Csaba Nagy, Lajos Jeno Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. "FaultBuster: An automatic code smell refactoring toolset". In: *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*. 2015, pp. 253–258. DOI: 10.1109/SCAM.2015.7335422. URL: http://dx.doi.org/10.1109/SCAM.2015.7335422.

[5] Gábor Szőke, Csaba Nagy, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. "Do automatic refactorings improve maintainability? An industrial case study". In: *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*. 2015, pp. 429–438. DOI: 10.1109/ICSM.2015.7332494. URL: http://dx.doi.org/10.1109/ICSM.2015.7332494.

[6] Gábor Szőke. "Automating the Refactoring Process". In: *Acta Cybernetica* 23.2 (2017), pp. 715–735. DOI: http://doi.org/10.14232/actacyb.23.2.2017.16. URL: http://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/4022.

[7] Gábor Szoke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. "Designing and Developing Automated Refactoring Transformations: An Experience Report". In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1.* 2016, pp. 693–697. DOI: 10.1109/SANER.2016.17. URL: http://dx.doi.org/10.1109/SANER.2016.17.

[8] Zoltán Ujhelyi, Ákos Horváth, Dániel Varró, Norbert Istvan Csiszár, Gábor Szőke, László Vidács, and Rudolf Ferenc. "Anti-pattern detection with model queries: A comparison of approaches". In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014.* 2014, pp. 293–302. DOI: 10.1109/CSMR-WCRE.2014.6747181. URL: http://dx.doi.org/10.1109/CSMR-WCRE.2014.6747181.

[9] Zoltán Ujhelyi, Gábor Szőke, Ákos Horváth, Norbert Istvan Csiszár, László Vidács, Dániel Varró, and Rudolf Ferenc. "Performance comparison of query-based techniques for anti-pattern detection". In: *Information & Software Technology* 65 (2015), pp. 147–165. DOI: 10.1016/j.infsof.2015.01.003. URL: http://dx.doi.org/10.1016/j.infsof.2015.01.003.

References

[10] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. "Characteristics of application software maintenance". In: *Communications of the ACM* 21.6 (1978), pp. 466–471.

[11] William F Opdyke. "Refactoring object-oriented frameworks". PhD thesis. University of Illinois, 1992.

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

[13] Andrew Koenig. "Patterns and antipatterns". In: *The patterns handbook: techniques, strategies, and applications* 13 (1998), p. 383.

[14] *ISTQB Exam Certification - What is Software Quality?* http://istqbexamcertification.com/what-is-software-quality/. (Visited on 03/28/2016).

[15] *Olivier Coudert - What is software quality?* http://www.ocoudert.com/blog/2011/04/09/what-is-software-quality/. (Visited on 03/28/2016).

[16] T. Bakota, P. Hegedus, G. Ladanyi, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. "A cost model based on software maintainability". In: *Proc. of the 28th IEEE Int. Conference on Software Maintenance (ICSM2012).* 2012, pp. 316–325.

[17] Joshua Kerievsky. *Refactoring to Patterns.* Pearson Higher Education, 2004. ISBN: 0321213351.

[18] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. DOI: 10.1109/32.295895.

[19] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. “Empirical validation of object-oriented metrics on open source software for fault prediction”. In: *IEEE Transactions on Software Engineering* 31.10 (2005), pp. 897–910. DOI: `10.1109/TSE.2005.112`.

[20] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. “A Probabilistic Software Quality Model”. In: *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*. ICSM ’11. IEEE, 2011, pp. 243–252. ISBN: 978-1-4577-0663-9. DOI: `10.1109/icsm.2011.6080791`.

[21] ISO/IEC. *ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. ISO/IEC, 2005.

[22] *The QualityGate Homepage*. URL: `http://quality-gate.com/`.

[23] FrontEndART Ltd. *SourceMeter*. URL: `https://www.sourcemeter.com`.

[24] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. “A Field Study of Refactoring Challenges and Benefits”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. Cary, North Carolina: ACM, 2012, 50:1–50:11. ISBN: 978-1-4503-1614-9.

[25] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. “Columbus – Reverse Engineering Tool and Schema for C++”. In: *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*. Montréal, Canada: IEEE Computer Society, Oct. 2002, pp. 172–181.

[26] *FrontEndART Software Ltd.* URL: `http://www.frontendart.com`.

[27] PMD. *PMD website*. URL: `https://pmd.github.io/`.

[28] Oracle Corporation. *NetBeans IDE*. `https://netbeans.org/`. (Visited on 05/16/2013).

[29] William C. Wake. *Refactoring Workbook*. 1st ed. Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321109295.

[30] Tom Mens and Tom Tourwé. “A survey of software refactoring”. In: *IEEE Transactions on Software Engineering* 30.2 (2004), pp. 126–139.

[31] Bart Du Bois, Serge Demeyer, and Jan Verelst. “Does the "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension?” In: *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*. CSMR ’05. IEEE Computer Society, 2005, pp. 334–343. ISBN: 0-7695-2304-8.

[32] N. Rachatasumrit and Miryung Kim. “An empirical investigation into the impact of refactoring on regression testing”. In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. 2012, pp. 357–366.

[33] Gustavo H. Pinto and Fernando Kamei. “What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow”. In: *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*. WRT ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 33–36. ISBN: 978-1-4503-2604-9.

[34] Houari A Sahraoui, Robert Godin, and Thierry Miceli. "Can metrics help to bridge the gap between the improvement of oo design quality and its automation?" In: *Proc. of Int. Conference on Software Maintenance.* IEEE. 2000, pp. 154–162.

[35] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. "Metrics based refactoring". In: *Proc. of the Fifth European Conference on Software Maintenance and Reengineering.* IEEE. 2001, pp. 30–38.

[36] Ladan Tahvildari and Kostas Kontogiannis. "A metric-based approach to enhance design quality through meta-pattern transformations". In: *Proc. of the Seventh European Conference on Software Maintenance and Reengineering.* IEEE. 2003, pp. 183–192.

[37] Ladan Tahvildari, Kostas Kontogiannis, and John Mylopoulos. "Quality-driven software re-engineering". In: *Journal of Systems and Software* 66.3 (2003), pp. 225–239.

[38] Yijun Yu, John Mylopoulos, Eric Yu, Julio Cesar Leite, Linda Liu, and Erik D'Hollander. "Software refactoring guided by multiple soft-goals". In: *Proc. of the 1st Workshop on Refactoring: Achievements, Challenges, and Effects, in conjunction with the 10th WCRE conference 2003.* IEEE Comp. Soc. 2003, pp. 7–11.

[39] Panita Meananeatra. "Identifying Refactoring Sequences for Improving Software Maintainability". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* ASE 2012. Essen, Germany: ACM, 2012, pp. 406–409. ISBN: 978-1-4503-1204-2.

[40] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. "On the Relation of Refactorings and Software Defect Prediction". In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories.* MSR '08. Leipzig, Germany: ACM, 2008, pp. 35–38. ISBN: 978-1-60558-024-1.

[41] Carsten Görg and Peter Weißgerber. "Error Detection by Refactoring Reconstruction". In: *Proceedings of the 2005 International Workshop on Mining Software Repositories.* MSR '05. St. Louis, Missouri: ACM, 2005, pp. 1–5. ISBN: 1-59593-123-6.

[42] Carsten Görg and Peter Weißgerber. "Error Detection by Refactoring Reconstruction". In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948.

[43] Peter Weißgerber and Stephan Diehl. "Identifying Refactorings from Source-Code Changes". In: *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering.* ASE '06. IEEE Computer Society, 2006, pp. 231–240. ISBN: 0-7695-2579-2.

[44] Peter Weißgerber and Stephan Diehl. "Are Refactorings Less Error-prone Than Other Changes?" In: *Proceedings of the 2006 International Workshop on Mining Software Repositories.* MSR '06. Shanghai, China: ACM, 2006, pp. 112–118.

[45] Eleni Stroulia and Rohit Kapoor. "Metrics of refactoring-based development: An experience report". In: *Proc. of the 7th Int. Conf. on Object-Oriented Information Systems (OOIS2001).* Springer, 2001, pp. 113–122.

[46] Bart Du Bois and Tom Mens. “Describing the impact of refactoring on internal program quality”. In: *Proc. of the Int. Workshop on Evolution of Large-scale Industrial Software Applications.* 2003, pp. 37–48.

[47] Bart Du Bois, Serge Demeyer, and Jan Verelst. “Refactoring-improving coupling and cohesion of existing code”. In: *Proc. of the 11th Working Conference on Reverse Engineering.* IEEE. 2004, pp. 144–151.

[48] Bart Du Bois. “A Study of Quality Improvements by Refactoring”. PhD thesis. University of Antwerp, 2006.

[49] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. “A quantitative evaluation of maintainability enhancement by refactoring”. In: *Proc. of the Int. Conference on Software Maintenance.* IEEE. 2002, pp. 576–585.

[50] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. “Balancing Agility and Formalism in Software Engineering”. In: ed. by Bertrand Meyer, Jerzy R. Nawrocki, and Bartosz Walter. Springer-Verlag, 2008. Chap. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pp. 252–266. ISBN: 978-3-540-85278-0.

[51] Jacek Ratzinger, Michael Fischer, and Harald Gall. “Improving Evolvability Through Refactoring”. In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5.

[52] Serge Demeyer. “Refactor conditionals into polymorphism: what’s the performance cost of introducing virtual calls?” In: *Proc. of the 21st IEEE Int. Conference on Software Maintenance, 2005. ICSM’05.* IEEE. 2005, pp. 627–630.

[53] Konstantinos Stroggylos and Diomidis Spinellis. “Refactoring–Does It Improve Software Quality?” In: *Proc. of the 5th Int. Workshop on Software Quality.* IEEE Comp. Soc. 2007, p. 10.

[54] Mohammad Alshayeb. “Empirical Investigation of Refactoring Effect on Software Quality”. In: *Inf. Softw. Technol.* 51.9 (Sept. 2009), pp. 1319–1326.

[55] Birgit Geppert, Audris Mockus, and Frank Rossler. “Refactoring for Changeability: A Way to Go?” In: *Proceedings of the 11th IEEE International Software Metrics Symposium.* METRICS ’05. IEEE Computer Society, 2005, pp. 13–. ISBN: 0-7695-2371-4.

[56] D. Wilking, U. F. Kahn, and S. Kowalewski. “An empirical evaluation of refactoring”. eng. In: *e-Informatica Software Engineering Journal* Vol. 1, nr 1 (2007), pp. 27–42.

[57] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. “A Comparative Study of Manual and Automated Refactorings”. In: *Proc. of the 27th European Conference on Object-Oriented Programming (ECOOP2013).* Montpellier, France: Springer-Verlag, 2013, pp. 552–576. ISBN: 978-3-642-39037-1.

[58] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. “How We Refactor, and How We Know It”. In: *Proc. of the 31st Int. Conference on Software Engineering (ICSE2009).* IEEE Comp. Soc., 2009, pp. 287–297. ISBN: 978-1-4244-3453-4.

[59] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. "Refactoring a Legacy Component for Reuse in a Software Product Line: A Case Study: Practice Articles". In: *J. Softw. Maint. Evol.* 18.2 (Mar. 2006), pp. 109–132. ISSN: 1532-060X.

[60] Miryung Kim, Dongxiang Cai, and Sunghun Kim. "An Empirical Investigation into the Role of API-level Refactorings During Software Evolution". In: *Proceedings of the 33rd International Conference on Software Engineering.* ICSE '11. ACM Press, 2011, pp. 151–160. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985815.

[61] Aiko Yamashita and Leon Moonen. "To What Extent Can Maintenance Problems Be Predicted by Code Smell Detection? - An Empirical Study". In: *Inf. Softw. Technol.* 55.12 (Dec. 2013), pp. 2223–2242. ISSN: 0950-5849.

[62] Aiko Yamashita and Steve Counsell. "Code smells as system-level indicators of maintainability: An empirical study". In: *Journal of Systems and Software* 86.10 (2013), pp. 2639 –2653. ISSN: 0164-1212.

[63] Aiko Yamashita. "Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data". In: *Empirical Softw. Engg.* 19.4 (Aug. 2014), pp. 1111–1143. ISSN: 1382-3256.

[64] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. "Some Code Smells Have a Significant but Small Effect on Faults". In: *ACM Trans. Softw. Eng. Methodol.* 23.4 (Sept. 2014), 33:1–33:39. ISSN: 1049-331X.

[65] Ali Ouni, Marouane Kessentini, Slim Bechikh, and Houari Sahraoui. "Prioritizing Code-smells Correction Tasks Using Chemical Reaction Optimization". In: *Software Quality Control* 23.2 (June 2015), pp. 323–361. ISSN: 0963-9314.

[66] Everton. Guimaraes, Alessandro Garcia, Eduardo Figueiredo, and Yuanfang Cai. "Prioritizing Software Anomalies with Software Metrics and Architecture Blueprints: A Controlled Experiment". In: *Proceedings of the 5th International Workshop on Modeling in Software Engineering.* MiSE '13. San Francisco, California: IEEE Press, 2013, pp. 82–88. ISBN: 978-1-4673-6447-8.

[67] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness". In: *Empirical Softw. Engg.* 17.3 (June 2012), pp. 243–275. ISSN: 1382-3256.

[68] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension". In: *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering.* CSMR '11. IEEE Computer Society, 2011, pp. 181–190. ISBN: 978-0-7695-4343-7.

[69] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. "On the Impact of Design Flaws on Software Defects". In: *Proceedings of the 2010 10th International Conference on Quality Software.* QSIC '10. IEEE Computer Society, 2010, pp. 23–31. ISBN: 978-0-7695-4131-0.

[70] Alexander Chatzigeorgiou and Anastasios Manakos. “Investigating the Evolution of Code Smells in Object-oriented Systems”. In: *Innov. Syst. Softw. Eng.* 10.1 (Mar. 2014), pp. 3–18. ISSN: 1614-5046.

[71] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. “A Multidimensional Empirical Study on Refactoring Activity”. In: *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research.* CASCON ’13. Ontario, Canada: IBM Corp., 2013, pp. 132–146.

[72] Francesca Arcelli Fontana, Marco Mangiacavalli, Domenico Pochiero, and Marco Zanoni. “On Experimenting Refactoring Tools to Remove Code Smells”. In: *Scientific Workshop Proceedings of the XP2015.* XP ’15 workshops. Helsinki, Finland: ACM, 2015, 7:1–7:8. ISBN: 978-1-4503-3409-9. DOI: `10.1145/2764979.2764986`.

[73] Oracle Corporation. *OpenJDK website: http://openjdk.java.net/.*

[74] Mark Harman and Laurence Tratt. “Pareto Optimal Search Based Refactoring at the Design Level”. In: *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation.* GECCO ’07. London, England: ACM, 2007, pp. 1106–1113. ISBN: 978-1-59593-697-4. DOI: `10.1145/1276958.1277176`. URL: `http://doi.acm.org/10.1145/1276958.1277176`.

[75] Mark O’Keeffe and Mel Ó Cinnéide. “Search-based refactoring for software maintenance”. In: *Journal of Systems and Software* 81.4 (2008). Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006), pp. 502 –516. ISSN: 0164-1212. DOI: `http://dx.doi.org/10.1016/j.jss.2007.06.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0164121207001409`.

[76] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi. “Search-based refactoring: Towards semantics preservation”. In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on.* 2012, pp. 347–356. DOI: `10.1109/ICSM.2012.6405292`.

[77] Philippe Rigaux, Michel Scholl, and Agnes Voisard. *Spatial databases: with application to GIS.* Morgan Kaufmann, 2001.

[78] Antonin Guttman. “R-trees: A Dynamic Index Structure for Spatial Searching”. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’84. Boston, Massachusetts: ACM, 1984, pp. 47–57. ISBN: 0-89791-128-8. DOI: `10.1145/602259.602266`. URL: `http://doi.acm.org/10.1145/602259.602266`.

[79] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications.* Advanced Information and Knowledge Processing. Springer London, 2010. ISBN: 9781846282935. URL: `https://books.google.hu/books?id=1mu099DN9UwC`.

[80] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. “The Vision of Software Clone Management: Past, Present, and Future (keynote paper)”. In: *Proc. of CSMR-WCRE.* IEEE. 2014, pp. 18–33.

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

[82] The Eclipse Project. *Eclipse Java development tools (JDT).* http://www.eclipse.org/jdt/. (Visited on 05/16/2013).

[83] JetBrains. *IntelliJ IDEA – The Java and Polyglot IDE.* http://www.jetbrains.com/idea/. (Visited on 05/16/2013).

[84] Aikaterini Christopoulou, E. A. Giakoumakis, Vassilis E. Zafeiris, and Vasiliki Soukara. "Automated Refactoring to the Strategy Design Pattern". In: *Information and Software Technology* 54.11 (Nov. 2012), pp. 1202–1214. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.05.004.

[85] *RefactorIt.* http://sourceforge.net/projects/refactorit/. 2008.

[86] *iPlasma.* http://loose.upt.ro/reengineering/research/iplasma. (Visited on 05/16/2013).

[87] *inCode.* http://www.intooitus.com/products/incode. (Visited on 05/16/2013).

[88] Javier Pérez and Yania Crespo. "Perspectives on Automated Correction of Bad Smells". In: *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops.* IWPSE-Evol '09. Amsterdam, The Netherlands: ACM, 2009, pp. 99–108. ISBN: 978-1-60558-678-6. DOI: 10.1145/1595808.1595827.

[89] Klocwork Inc. *Klocwork Insight.* http://www.klocwork.com/. (Visited on 05/16/2013).

[90] Coverity. *Coverity Static Analysis Verification Engine.* http://www.coverity.com/. (Visited on 05/16/2013).

[91] FindBugs. – *Find Bugs in Java Programs.* http://findbugs.sourceforge.net/. (Visited on 05/16/2013).

[92] Checkstyle. – *static code analysis tool.* http://checkstyle.sourceforge.net/. (Visited on 05/16/2013).

[93] *FxCop.* https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx. (Visited on 05/16/2013).

[94] *DMS Software Reengineering Toolkit.* http://www.semdesigns.com/products/DMS/DMSToolkit.html. (Visited on 05/16/2013).

[95] *ReSharper.* http://www.jetbrains.com/resharper/. (Visited on 05/16/2013).

[96] *CodeRush.* https://www.devexpress.com/products/coderush/. (Visited on 05/16/2013).

[97] *Refaster.* https://github.com/google/Refaster. (Visited on 05/16/2013).

[98] Louis Wasserman. "Scalable, Example-based Refactorings with Refaster". In: *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools.* WRT '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 25–28. ISBN: 978-1-4503-2604-9. DOI: 10.1145/2541348.2541355.

[99] Zhenchang Xing and Eleni Stroulia. "Refactoring Practice: How It is and How It Should Be Supported - An Eclipse Case Study". In: *Proc. of the 22nd IEEE Int. Conference on Software Maintenance (ICSM2006).* IEEE Comp. Soc., 2006, pp. 458–468. ISBN: 0-7695-2354-4. DOI: `10.1109/ICSM.2006.52`.

[100] Danny Dig and Ralph Johnson. "The Role of Refactorings in API Evolution". In: *Proc. of the 21st IEEE Int. Conference on Software Maintenance (ICSM2005).* IEEE Comp. Soc., 2005, pp. 389–398. ISBN: 0-7695-2368-4. DOI: `10.1109/ICSM.2005.90`.

[101] Xi Ge and Emerson Murphy-Hill. "BeneFactor: A Flexible Refactoring Tool for Eclipse". In: *Proc. of the ACM Int. Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA2011).* Portland, Oregon, USA: ACM, 2011, pp. 19–20. ISBN: 978-1-4503-0942-4.

[102] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. "Reconciling Manual and Automatic Refactoring". In: *Proc. of the 34th Int. Conference on Software Engineering (ICSE2012).* IEEE Press, 2012, pp. 211–221. ISBN: 978-1-4673-1067-3.

[103] Johannes Henkel and Amer Diwan. "CatchUp!: Capturing and Replaying Refactorings to Support API Evolution". In: *Proc. of the 27th Int. Conference on Software Engineering (ICSE2005).* St. Louis, MO, USA: ACM, 2005, pp. 274–283. ISBN: 1-58113-963-2. DOI: `10.1145/1062455.1062512`.

[104] Adam C. Jensen and Betty H.C. Cheng. "On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns". In: *Proc. of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO2010).* Portland, Oregon, USA: ACM, 2010, pp. 1341–1348. ISBN: 978-1-4503-0072-8. DOI: `10.1145/1830483.1830731`.

[105] Asger Feldthaus and Anders Møller. "Semi-automatic Rename Refactoring for JavaScript". In: *SIGPLAN Not.* 48.10 (Oct. 2013), pp. 323–338. ISSN: 0362-1340. DOI: `10.1145/2544173.2509520`.

[106] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. "Recommending Automated Extract Method Refactorings". In: *Proc. of the 22nd Int. Conference on Program Comprehension (ICPC2014).* Hyderabad, India: ACM, 2014, pp. 146–156. ISBN: 978-1-4503-2879-1. DOI: `10.1145/2597008.2597141`.

[107] Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. "Active Support for Clone Refactoring: A Perspective". In: *Proc. of the 2013 ACM Workshop on Workshop on Refactoring Tools (WRT2013).* Indianapolis, Indiana, USA: ACM, 2013, pp. 13–16. ISBN: 978-1-4503-2604-9. DOI: `10.1145/2541348.2541352`.

[108] Miryung Kim, T. Zimmermann, and N. Nagappan. "An Empirical Study of Refactoring Challenges and Benefits at Microsoft". In: *IEEE Transactions on Software Engineering* 40.7 (2014), pp. 633–649. ISSN: 0098-5589.

[109] *SonarQube Homepage.* URL: `https://www.sonarqube.org/`.

[110] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. "A Graph Query Language for EMF Models". In: *Theory and Practice of Model Transformations.* Vol. 6707. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 167–182. ISBN: 978-3-642-21731-9. DOI: `10.1007/978-3-642-21732-6_12`.

[111] *Object Constraint Language Specification (Version 2.3.1).* `http://www.omg.org/spec/OCL/2.3.1/`. Object Management Group. 2012.

[112] László Vidács. "Refactoring of C/C++ Preprocessor Constructs at the Model Level". In: *Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT 2009).* Sofia, Bulgaria, 2009, pp. 232–237.

[113] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. "An Algorithm for Generating Model-Sensitive Search Plans for EMF Models". In: *Theory and Practice of Model Transformations.* Vol. 7307. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 224–239. ISBN: 978-3-642-30475-0. DOI: `10.1007/978-3-642-30476-7_15`.

[114] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró. "Incremental pattern matching in the VIATRA transformation system". In: *Proceedings of 3rd International Workshop on Graph and Model Transformation (GRaMoT 2008).* 30th International Conference on Software Engineering. ACM, 2008, pp. 25–32.

[115] László Vidács, Árpád Beszédes, and Rudolf Ferenc. "Columbus Schema for C/C++ Preprocessing". In: *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004).* Tampere, Finland: IEEE Computer Society, Mar. 2004, pp. 75–84.

[116] Lars Hamann, László Vidács, Martin Gogolla, and Mirco Kuhlmann. "Abstract Runtime Monitoring with USE". In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012).* Szeged, Hungary: IEEE Computer Society, 2012, pp. 549–552. DOI: `10.1109/CSMR.2012.73`.

[117] L. Schrettner, L.J. Fülöp, R. Ferenc, and T. Gyimóthy. "Visualization of software architecture graphs of Java systems: managing propagated low level dependencies". In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010).* New York, NY, USA: ACM, 2010, 148–157. ISBN: 978-1-4503-0269-2. DOI: `http://doi.acm.org/10.1145/1852761.1852783`.

[118] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[119] Gábor Bergmann. "Incremental Model Queries in Model-Driven Design". Ph.D. dissertation. Budapest: Budapest University of Technology and Economics, 2013.

[120] Zoltán Ujhelyi, Ákos Horváth, Dániel Varró, Norbert István Csiszár, Gábor Szőke, László Vidács, and Rudolf Ferenc. "Anti-pattern detection with model queries: A comparison of approaches". In: *Proceedings of IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014), 2014 Software Evolution Week.* 2014, pp. 293–302. DOI: `10.1109/CSMR-WCRE.2014.6747181`.

[121] U. Nickel, J. Niere, and A. Zündorf. "Tool demonstration: The FUJABA environment". In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000).* Limerick, Ireland: ACM Press, 2000, pp. 742–745.

[122] *The ATLAS Transformation Language.* `http://www.eclipse.org/atl`. ATLAS Group. 2014.

[123] Rubino Geiß, GernotVeit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. “GrGen: A Fast SPO-Based Graph Rewriting Tool”. English. In: *Graph Transformations*. Vol. 4178. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 383–397. ISBN: 978-3-540-38870-8. DOI: 10.1007/11841883_27. URL: http://dx.doi.org/10.1007/11841883_27.

[124] David Hearnden, Michael Lawley, and Kerry Raymond. “Incremental Model Transformation for the Evolution of Model-Driven Systems”. English. In: *Model Driven Engineering Languages and Systems*. Vol. 4199. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 321–335. ISBN: 978-3-540-45772-5. DOI: 10.1007/11880240_23. URL: http://dx.doi.org/10.1007/11880240_23.

[125] Charles L. Forgy. “Rete: A fast algorithm for the many pattern/many object pattern match problem”. In: *Artificial Intelligence* 19.1 (Sept. 1982), pp. 17–37. ISSN: 0004-3702. DOI: 10.1016/0004-3702(82)90020-0.

[126] *Drools - The Business Logic integration Platform*. http://www.jboss.org/drools. 2014.

[127] AmirHossein Ghamarian, Arash Jalali, and Arend Rensink. “Incremental pattern matching in graph-based state space exploration”. In: *Electronic Communications of the EASST* 32 (2011).

[128] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. “EMF-IncQuery: An integrated development environment for live model queries”. In: *Science of Computer Programming* 98, Part 1.0 (2015). Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012), pp. 80–99. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2014.01.004. URL: http://www.sciencedirect.com/science/article/pii/S0167642314000082.

[129] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth, and István Ráth. “Towards Precise Metrics for Predicting Graph Query Performance”. In: *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE 2013)*. Silicon Valley, CA, USA: IEEE, 2013, pp. 412–431. DOI: 10.1109/ASE.2013.6693100.

[130] Jordi Cabot and Ernest Teniente. “A metric for measuring the complexity of OCL expressions”. In: *Proceedings of the Model Size Metrics Workshop @ MoDELS 2006*. 2006.

[131] Eclipse OCL Project. *MDT-OCL website*. https://projects.eclipse.org/projects/modeling.mdt.ocl. 2014.

[132] Ákos Horváth, Gábor Bergmann, István Ráth, and Dániel Varró. “Experimental assessment of combining pattern matching strategies with VIATRA2”. English. In: *International Journal on Software Tools for Technology Transfer* 12.3-4 (2010), pp. 211–230. ISSN: 1433-2779. DOI: 10.1007/s10009-010-0149-7. URL: http://dx.doi.org/10.1007/s10009-010-0149-7.

[133] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. "IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud". English. In: *Model-Driven Engineering Languages and Systems.* Vol. 8767. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 653–669. ISBN: 978-3-319-11652-5. DOI: `10.1007/978-3-319-11653-2_40`. URL: `http://dx.doi.org/10.1007/978-3-319-11653-2_40`.

[134] Javier Pérez, Yania Crespo, Berthold Hoffmann, and Tom Mens. "A case study to evaluate the suitability of graph transformation tools for program refactoring". English. In: *International Journal on Software Tools for Technology Transfer* 12.3-4 (2010), pp. 183–199. ISSN: 1433-2779. DOI: `10.1007/s10009-010-0153-y`.

[135] Tassilo Horn. "Program Understanding: A Reengineering Case for the Transformation Tool Contest". In: Proceedings Fifth *Transformation Tool Contest,* Zürich, Switzerland, June 29-30 2011. Vol. 74. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2011, pp. 17–21. DOI: `10.4204/EPTCS.74.3`.

[136] Javier Espinazo Pagán and Jesús García Molina. "Querying large models efficiently". In: *Information and Software Technology* 56.6 (2014), pp. 586 –622. ISSN: 0950-5849. DOI: `http://dx.doi.org/10.1016/j.infsof.2014.01.005`. URL: `http://www.sciencedirect.com/science/article/pii/S0950584914000160`.

[137] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. "Formalizing refactorings with graph transformations". In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.4 (2005), pp. 247–276. ISSN: 1532-0618. DOI: `10.1002/smr.316`.

[138] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. "Towards pattern-based design recovery". In: *Proceedings of the 24th International Conference on Software Engineering.* (ICSE 2002). Orlando, Florida: ACM, 2002, pp. 338–348. ISBN: 1-58113-472-X. DOI: `10.1145/581339.581382`.

[139] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. "Closing the Gap between Modelling and Java". English. In: *Software Language Engineering.* Vol. 5969. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 374–383. ISBN: 978-3-642-12106-7. DOI: `10.1007/978-3-642-12107-4_25`. URL: `http://dx.doi.org/10.1007/978-3-642-12107-4_25`.

[140] Mirko Seifert and Roland Samlaus. "Static Source Code Analysis using OCL". en. In: *Electronic Communications of the EASST* 15.0 (Jan. 2008). ISSN: 1863-2122. URL: `http://journal.ub.tu-berlin.de/eceasst/article/view/174` (visited on 06/15/2014).

[141] Thorsten Arendt and Gabriele Taentzer. "Integration of Smells and Refactorings Within the Eclipse Modeling Framework". In: *Proceedings of the Fifth Workshop on Refactoring Tools.* (WRT 2012). Rapperswil, Switzerland: ACM, 2012, pp. 8–15. ISBN: 978-1-4503-1500-5. DOI: `10.1145/2328876.2328878`. URL: `http://doi.acm.org/10.1145/2328876.2328878`.

[142] FrontEndART Software Ltd. *SourceMeter module: FaultHunter*. `http://www.frontendart.com/`. 2014. (Visited on 2014).

[143] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. “Why Unified Is not Universal”. In: *«UML» ’99 — The Unified Modeling Language*. Vol. 1723. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 630–644. ISBN: 978-3-540-66712-4. DOI: `10.1007/3-540-46852-8_44`.

[144] Stéphane Ducasse, Tudor Girba, Adrian Kuhn, and Lukas Renggli. “Meta-environment and executable meta-language using Smalltalk: an experience report”. In: *Software & Systems Modeling* 8.1 (2009), pp. 5–19. ISSN: 1619-1366. DOI: `10.1007/s10270-008-0081-4`.

[145] P. Klint, T. van der Storm, and J. J. Vinju. “Rascal: A Domain Specific Language For Source Code Analysis And Manipulation”. In: *Proceedings of IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009)*. IEEE, 2009, pp. 168–177.

[146] Daniel Speicher, Malte Appeltauer, and Günter Kniesel. “Code Analyses for Refactoring by Source Code Patterns and Logical Queries”. In: *Proceedings of the 1st Workshop on Refactoring Tools (WRT 2007)*. 2007, pp. 17–20.

[147] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. “The SOUL tool suite for querying programs in symbiosis with Eclipse”. In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. (PPPJ 2011). Kongens Lyngby, Denmark: ACM, 2011, pp. 71–80. ISBN: 978-1-4503-0935-6. DOI: `10.1145/2093157.2093168`.

[148] Elnar Hajiyev, Mathieu Verbaere, and Oege Moor. “codeQuest: Scalable Source Code Queries with Datalog”. In: *ECOOP 2006 – Object-Oriented Programming*. Vol. 4067. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 2–27. ISBN: 978-3-540-35726-1. DOI: `10.1007/11785477_2`.

[149] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur. “DECOR: A Method for the Specification and Detection of Code and Design Smells”. In: *Software Engineering, IEEE Transactions on* 36.1 (2010), pp. 20–36. ISSN: 0098-5589. DOI: `10.1109/TSE.2009.50`.

This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled “Internet of Living Things”.

Appendices

A
PMD rule violations

The following pages contain descriptions of rule violations from the official PMD website [27]. Only those violations are listed here, which are mentioned in the thesis.

AddEmptyString

Finds empty string literals which are being added. This is an inefficient way to convert any type to a String.

```
String s = "" + 123; // bad
String t = Integer.toString(456); // ok
```

ArrayIsStoredDirectly

Constructors and methods receiving arrays should clone objects and store the copy. This prevents that future changes from the user affect the internal functionality.

```
public class Foo {
  private String [] x;
  public void foo (String [] param) {
    // Don't do this, make a copy of the array at least
    this.x = param;
  }
}
```

AtLeastOneConstructor

Each class should declare at least one constructor.

```
public class Foo {
  // no constructor!  not good!
}
```

AvoidCatchingNPE

Code should never throw NPE under normal circumstances. A catch block may hide the original error, causing other more subtle errors in its wake.

```java
public class Foo {
  void bar() {
    try {
      // do something
    } catch (NullPointerException npe) {
    }
  }
}
```

AvoidCatchingThrowable

This is dangerous because it casts too wide a net; it can catch things like OutOfMemoryError.

```java
public class Foo {
  public void bar() {
    try {
      // do something
    } catch (Throwable th) {  //Should not catch
       throwable
      th.printStackTrace();
    }
  }
}
```

AvoidDuplicateLiterals

Code containing duplicate String literals can usually be improved by declaring the String as a constant field.

```java
public class Foo {
  private void bar() {
    buz("Howdy");
    buz("Howdy");
    buz("Howdy");
    buz("Howdy");
  }
  private void buz(String x) {}
}
```

AvoidInstanceofChecksInCatchClause

Each caught exception type should be handled in its own catch clause.

```
try { // Avoid this
   // do something
} catch (Exception ee) {
  if (ee instanceof IOException) {
    cleanup();
  }
}
try {  // Prefer this:
  // do something
} catch (IOException ee) {
  cleanup();
}
```

AvoidPrintStackTrace

Avoid printStackTrace(); use a logger call instead.

```
class Foo {
  void bar() {
    try {
      // do something
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

AvoidReassigningParameters

Reassigning values to parameters is a questionable practice. Use a temporary local variable instead.

```
public class Foo {
  private void foo(String bar) {
    bar = "something else";
  }
}
```

AvoidSynchronizedAtMethodLevel

Method level synchronization can backfire when new code is added to the method. Block-level synchronization helps to ensure that only the code that needs synchronization gets it.

```
public class Foo {
  // Try to avoid this
  synchronized void foo() {
  }
  // Prefer this:
  void bar() {
    synchronized(this) {
    }
  }
}
```

AvoidThrowingNullPointerException

Avoid throwing a NullPointerException - it is confusing because most people will assume that the virtual machine threw it. Consider using an IllegalArgumentException instead; this will be clearly seen as a programmer-initiated exception.

```
public class Foo {
  void bar() {
    throw new NullPointerException();
  }
}
```

AvoidThrowingRawExceptionTypes

Avoid throwing certain exception types. Rather than throw a raw RuntimeException, Throwable, Exception, or Error, use a subclassed exception or error instead.

```
public class Foo {
  public void bar() throws Exception {
    throw new Exception();
  }
}
```

BigIntegerInstantiation

Don't create instances of already existing BigInteger (BigInteger.ZERO, BigInteger.ONE) and for 1.5 on, BigInteger.TEN and BigDecimal (BigDecimal.ZERO, BigDecimal.ONE, BigDecimal.TEN)

```
public class Test {
  public static void main(String[] args) {
    BigInteger bi=new BigInteger(1);
    BigInteger bi2=new BigInteger("0");
    BigInteger bi3=new BigInteger(0.0);
    BigInteger bi4;
    bi4 = new BigInteger(0);
  }
}
```

BooleanGetMethodName

Looks for methods named 'getX()' with 'boolean' as the return type. The convention is to name these methods 'isX()'.

```java
public boolean getFoo(); // bad
public boolean isFoo(); // ok
public boolean getFoo(boolean bar); // ok, unless
    checkParameterizedMethods=true
```

BooleanInstantiation

Avoid instantiating Boolean objects; you can reference Boolean.TRUE, Boolean.FALSE, or call Boolean.valueOf() instead.

```java
public class Foo {
  Boolean bar = new Boolean("true"); // just do a
      Boolean bar = Boolean.TRUE;
  Boolean buz = Boolean.valueOf(false); // just do a
      Boolean buz = Boolean.FALSE;
}
```

ConsecutiveLiteralAppends

Consecutively calling StringBuffer.append with String literals

```java
public class Foo {
  private void bar() {
    StringBuffer buf = new StringBuffer();
    buf.append("Hello").append(" ").append("World"); //
        bad
    buf.append("Hello World");//good
  }
}
```

ConstructorCallsOverridableMethod

Calling overridable methods during construction poses a risk of invoking methods on an incompletely constructed object and can be difficult to discern. It may leave the sub-class unable to construct its superclass or forced to replicate the construction process completely within itself, losing the ability to call super(). If the default constructor contains a call to an overridable method, the subclass may be completely uninstantiable. Note that this includes method calls throughout the control flow graph - i.e., if a constructor Foo() calls a private method bar() that calls a public method buz(), this denotes a problem.

```java
public class SeniorClass {
  public SeniorClass(){
      toString(); //may throw NullPointerException if overridden
  }
  public String toString(){
    return "IAmSeniorClass";
  }
}
public class JuniorClass extends SeniorClass {
  private String name;
  public JuniorClass(){
    super(); //Automatic call leads to NullPointerException
    name = "JuniorClass";
  }
  public String toString(){
    return name.toUpperCase();
  }
}
```

EmptyCatchBlock

Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported.

```
public void doSomething() {
  try {
    FileInputStream fis = new FileInputStream("/tmp/bugger");
  } catch (IOException ioe) {
      // not good
  }
}
```

EmptyIfStmt

Empty If Statement finds instances where a condition is checked but nothing is done about it.

```
public class Foo {
  void bar(int x) {
    if (x == 0) {
      // empty!
    }
  }
}
```

ExceptionAsFlowControl

Using Exceptions as flow control leads to GOTOish code and obscures true exceptions when debugging.

```java
public class Foo {
  void bar() {
    try {
      try {
      } catch (Exception e) {
        throw new WrapperException(e);
        // this is essentially a GOTO to the
           WrapperException catch block
      }
    } catch (WrapperException e) {
      // do some more stuff
    }
  }
}
```

IfElseStmtsMustUseBraces

Avoid using if..else statements without using curly braces.

```java
public void doSomething() {
  // this is OK
  if (foo) x++;
  // but this is not
  if (foo)
    x=x+1;
  else
    x=x-1;
}
```

InefficientStringBuffering

Avoid concatenating non literals in a StringBuffer constructor or append().

```java
public class Foo {
  void bar() {
    // Avoid this
    StringBuffer sb=new StringBuffer("tmp = "+System.
       getProperty("java.io.tmpdir"));
    // use instead something like this
    StringBuffer sb = new StringBuffer("tmp = ");
    sb.append(System.getProperty("java.io.tmpdir"));
  }
}
```

IntegerInstantiation

In JDK 1.5, calling new Integer() causes memory allocation. Integer.valueOf() is more memory friendly.

```java
public class Foo {
  private Integer i = new Integer(0); // change to Integer i = Integer.valueOf(0);
}
```

LocalVariableCouldBeFinal

A local variable assigned only once can be declared final.

```java
public class Bar {
  public void foo () {
    String a = "a"; //if a will not be assigned again it is better to do this:
    final String b = "b";
  }
}
```

LooseCoupling

Avoid using implementation types (i.e., HashSet); use the interface (i.e, Set) instead.

```java
import java.util.ArrayList;
import java.util.HashSet;
public class Bar {
  // Use List instead
  private ArrayList list = new ArrayList();
  // Use Set instead
  public HashSet getFoo() {
    return new HashSet();
  }
}
```

MethodNamingConventions

Method names should always begin with a lower case character, and should not contain underscores.

```java
public class Foo {
  public void fooStuff() {
  }
}
```

MethodReturnsInternalArray

Exposing internal arrays directly allows the user to modify some code that could be critical. It is safer to return a copy of the array.

```java
public class SecureSystem {
  UserData [] ud;
  public UserData [] getUserData() {
    // Don't return directly the internal array, return
       a copy
    return ud;
  }
}
```

MethodWithSameNameAsEnclosingClass

Non-constructor methods should not have the same name as the enclosing class.

```java
public class MyClass {
  // this is bad because it is a method
  public void MyClass() {}
  // this is OK because it is a constructor
  public MyClass() {}
}
```

NonThreadSafeSingleton

Non-thread safe singletons can result in bad state changes. Eliminate static singletons if possible by instantiating the object directly. Static singletons are usually not needed as only a single instance exists anyway. Other possible fixes are to synchronize the entire method or to use an initialize-on-demand holder class (do not use the double-check idiom). See Effective Java, item 48.

```java
private static Foo foo = null;

//multiple simultaneous callers may see partially
   initialized objects
public static Foo getFoo() {
  if (foo==null)
    foo = new Foo();
  return foo;
}
```

OverrideBothEqualsAndHashcode

Override both public boolean Object.equals(Object other), and public int Object.hashCode(), or override neither. Even if you are inheriting a hashCode() from a parent class, consider implementing hashCode and explicitly delegating to your superclass.

```
// this is bad
public class Bar {
  public boolean equals(Object o) {
    // do some comparison
  }
}

// and so is this
public class Baz {
  public int hashCode() {
    // return some hash value
  }
}

// this is OK
public class Foo {
  public boolean equals(Object other) {
    // do some comparison
  }
  public int hashCode() {
    // return some hash value
  }
}
```

PositionLiteralsFirstInComparisons

Position literals first in String comparisons - that way if the String is null you won't get a NullPointerException, it'll just return false.

```
class Foo {
  boolean bar(String x) {
    return x.equals("2"); // should be "2".equals(x)
  }
}
```

PreserveStackTrace

Throwing a new exception from a catch block without passing the original exception into the new exception will cause the true stack trace to be lost, and can make it difficult to debug effectively.

```
public class Foo {
  void good() {
    try{
      Integer.parseInt("a");
    } catch(Exception e){
      throw new Exception(e);
    }
  }
  void bad() {
    try{
      Integer.parseInt("a");
    } catch(Exception e){
      throw new Exception(e.getMessage());
    }
  }
}
```

ProperCloneImplementation

Object clone() should be implemented with super.clone().

```
class Foo{
  public Object clone(){
    return new Foo(); // This is bad
  }
}
```

ReplaceHashtableWithMap

Consider replacing this Hashtable with the newer java.util.Map

```
public class Foo {
  void bar() {
    Hashtable h = new Hashtable();
  }
}
```

ReplaceVectorWithList

Consider replacing Vector usages with the newer java.util.ArrayList if expensive thread-safe operation is not required.

```
public class Foo {
  void bar() {
    Vector v = new Vector();
  }
}
```

ShortMethodName

Detects when very short method names are used.

```
public class ShortMethod {
  public void a(int i) { // Violation
  }
}
```

SignatureDeclareThrowsException

It is unclear which exceptions that can be thrown from the methods. It might be difficult to document and understand the vague interfaces. Use either a class derived from RuntimeException or a checked exception.

```
public void methodThrowingException() throws Exception {
}
```

SimpleDateFormatNeedsLocale

Be sure to specify a Locale when creating a new instance of SimpleDateFormat.

```
public class Foo {
  // Should specify Locale.US (or whatever)
  private SimpleDateFormat sdf = new SimpleDateFormat("
     pattern");
}
```

SimplifyConditional

No need to check for null before an instanceof; the instanceof keyword returns false when given a null argument.

```
class Foo {
  void bar(Object x) {
    if (x != null && x instanceof Bar) {
      // just drop the "x != null" check
    }
  }
}
```

SuspiciousHashcodeMethodName

The method name and return type are suspiciously close to hashCode(), which may mean you are intending to override the hashCode() method.

```
public class Foo {
  public int hashcode() {
    // oops, this probably was supposed to be hashCode
  }
}
```

SwitchStmtsShouldHaveDefault

Switch statements should have a default label.

```java
public class Foo {
  public void bar() {
    int x = 2;
    switch (x) {
     case 2: int j = 8;
    }
  }
}
```

UnnecessaryConstructor

This rule detects when a constructor is not necessary; i.e., when there's only one constructor, it's public, has an empty body, and takes no arguments.

```java
public class Foo {
  public Foo() {}
}
```

UnnecessaryLocalBeforeReturn

Avoid creating unnecessarily local variables.

```java
public class Foo {
  public int foo() {
    int x = doSomething();
    return x;  // instead, just 'return doSomething();'
  }
}
```

UnnecessaryWrapperObjectCreation

Parsing method should be called directly instead.

```java
public int convert(String s) {
  int i, i2;

  i = Integer.valueOf(s).intValue(); // this wastes an
     object
  i = Integer.parseInt(s); // this is better

  i2 = Integer.valueOf(i).intValue(); // this wastes an
     object
  i2 = i; // this is better

  String s3 = Integer.valueOf(i2).toString(); // this
     wastes an object
  s3 = Integer.toString(i2); // this is better

  return i2;
}
```

UnsynchronizedStaticDateFormatter

SimpleDateFormat is not synchronized. Sun recomends separate format instances for each thread. If multiple threads must access a static formatter, the formatter must be synchronized either on method or block level.

```
public class Foo {
  private static final SimpleDateFormat sdf = new
     SimpleDateFormat();
  void bar() {
    sdf.format(); // bad
  }
  synchronized void foo() {
    sdf.format(); // good
  }
}
```

UnusedImports

Avoid unused import statements.

```
// this is bad
import java.io.File;
public class Foo {}
```

UnusedLocalVariable

Detects when a local variable is declared and/or assigned, but not used.

```
public class Foo {
  public void doSomething() {
    int i = 5; // Unused
  }
}
```

UnusedModifier

Fields in interfaces are automatically public static final, and methods are public abstract. Classes or interfaces nested in an interface are automatically public and static (all nested interfaces are automatically static). For historical reasons, modifiers which are implied by the context are accepted by the compiler, but are superfluous.

```java
public interface Foo {
  public abstract void bar(); // both abstract and
     public are ignored by the compiler
  public static final int X = 0; // public, static, and
     final all ignored
  public static class Bar {} // public, static ignored
  public static interface Baz {} // ditto
}
public class Bar {
  public static interface Baz {} // static ignored
}
```

UnusedPrivateField

Detects when a private field is declared and/or assigned a value, but not used.

```java
public class Something {
  private static int FOO = 2; // Unused
  private int i = 5; // Unused
  private int j = 6;
  public int addOne() {
    return j++;
  }
}
```

UnusedPrivateMethod

Unused Private Method detects when a private method is declared but is unused.

```java
public class Something {
  private void foo() {} // unused
}
```

UseArrayListInsteadOfVector

ArrayList is a much better Collection implementation than Vector.

```java
public class SimpleTest extends TestCase {
  public void testX() {
    Collection c = new Vector();
    // This achieves the same with much better
       performance
    // Collection c = new ArrayList();
  }
}
```

UseEqualsToCompareStrings

Using '==' or '!=' to compare strings only works if intern version is used on both sides

```
class Foo {
  boolean test(String s) {
    if (s == "one") return true; //Bad
    if ("two".equals(s)) return true; //Better
    return false;
  }
}
```

UseIndexOfChar

Use String.indexOf(char) when checking for the index of a single character; it executes faster.

```
public class Foo {
  void bar() {
    String s = "hello world";
    // avoid this
    if (s.indexOf("d") {}
    // instead do this
    if (s.indexOf('d') {}
  }
}
```

UselessParentheses

Sometimes expressions are wrapped in unnecessary parentheses, making them look like a function call.

```
public class Foo {
  boolean bar() {
    return (true);
  }
}
```

UseLocaleWithCaseConversions

When doing a String.toLowerCase()/toUpperCase() call, use a Locale. This avoids problems with certain locales, i.e. Turkish.

```
class Foo {
 // BAD
 if (x.toLowerCase().equals("list"))...
 /*
 This will not match "LIST" when in Turkish locale
 The above could be
```

```
  if (x.toLowerCase(Locale.US).equals("list")) ...
  or simply
  if (x.equalsIgnoreCase("list")) ...
  */
  // GOOD
  String z = a.toLowerCase(Locale.EN);
}
```

UseStringBufferForStringAppends

Finds usages of += for appending strings.

```
public class Foo {
  void bar() {
    String a;
    a = "foo";
    a += " bar";
    // better would be:
    // StringBuffer a = new StringBuffer("foo");
    // a.append(" bar);
  }
}
```

B Summary

At some stage in their career every developer eventually encounters the code that no one understands and that no one wants to touch in case it breaks. But how did the software become so bad? Presumably no one set out to make it like that. The process that the software is suffering from is called *software erosion* – the constant decay of a software system that occurs in all phases of software development and maintenance.

Software erosion is inevitable. It is typical of software systems that they evolve over time, so they get enhanced, modified, and adapted to new requirements. As a side-effect the source code usually becomes more complex, and drifts away from its original design, then the maintainability costs of the software increases. This is one reason why a major part of the total software development cost (about 80%) is spent on software maintenance tasks [10]. One solution to prevent the detrimental effects of this software erosion, and to improve the maintainability is to perform refactoring tasks regularly.

The term *refactoring* became popular after Fowler published a catalog of refactoring transformations [12]. These transformations were meant to fix so-called 'bad smells' (a.k.a. 'code smells'). Bad smells indicate badly constructed and hard-to-maintain code segments. For example, the method at hand may be very long, or it may be a near duplicate of another nearby method. The benefit of understanding code smells is to help one discover and correct the anti-patterns and bugs that are the real problems. Eliminating these issues should help one to create quality software.

Keeping software maintainability high is in everybody's interest. The users get their new features faster and with fewer bugs, the developers have an easier job modifying the code, and the company should have lower maintenance costs. Good maintainability can be achieved via very detailed specification and elaborated development plans. However, this is very rare and only specific projects have the ability to do so. Because software is always evolving, in practice, the continuous-refactoring approach seems more feasible. This means that developers should from time to time refactor the code to make it more maintainable. A maintenance activity like this keeps the code "fresh" and hopefully extends its lifetime.

A key goal of this thesis is to contribute to the automated support of software system

maintenance. More specifically, the thesis seeks to propose methodologies, techniques and tools for:

- analyzing software developers behavior during hand-written and tool-aided refactoring tasks;
- evaluating the beneficial and detrimental effects of refactoring on software quality;
- adapting local-search based anti-pattern detection to model-query based techniques in general, and to graph pattern matching in particular.

Evaluation of Developers' Refactoring Habits

The aim of our experiments was to learn how developers refactor in an industrial context when they have the required resources (time and money) to do so. Our experiments were carried out on six large-scale industrial Java projects of different sizes and complexity. We studied refactorings on these systems, and learned which kinds of issues developers fixed the most, and which of these refactorings were best according to certain attributes. We investigated the effects of refactoring commits on source code maintainability using maintainability measurements based on the ColumbusQM maintainability model [20].

We found that developers tried to optimize their refactoring process to improve the quality of these systems and that they preferred to fix concrete coding issues rather than fix code smells suggested by metrics or automatic smell detectors. We think that the outcome of one refactoring on the global maintainability of the software product is hard to predict; moreover, it might sometimes have a detrimental effect. However, a big refactoring process can have a significant beneficial effect on the maintainability, which is measurable using a maintainability model. The reason for this is not only because the developers improve the maintainability of their software, but also because they will learn from the process and pay more attention to writing more maintainable new code in the future.

Challenges and Benefits of Automated Refactoring

Here, we sought to develop automated refactorings and for this purpose we designed FaultBuster, which is an automated refactoring framework. We presented an automated process for refactoring coding issues. We used the output of a third-party static analyzer to find refactoring suggestions, then we created an algorithm that was capable of locating a source code element in an AST based on textual position information. The algorithm transforms the source code into a searchable geometric space by building a spatial database.

We had to take into account several expectations of the developers when we designed and implemented the automatic refactoring tools. Among several challenges of the implementation, we identified some quite important ones, such as performance, indentation, formatting, understandability, precise problem detection, and the necessity of a precise syntax tree. Some of these have strong influence on the usability of a refactoring tool, hence they should be considered early on the design phase. We performed an exhaustive evaluation, which confirmed that our approach can be adapted to a real-life scenario, and it provides viable results.

We made interesting observations about the opinions of the developers who utilized our tools. The results showed that they found most of the manual refactorings of coding issues easily implementable via automatic transformations. Also, when we implemented these transformations and observed the automated solutions, we found that almost all refactoring types helped them to improve their code.

Employing the QualityGate SourceAudit tool, we analyzed the maintainability changes caused by the different refactoring tasks. The analysis revealed that out of the supported coding issue fixes, all but one type of refactoring operation had a consistent and traceable positive impact on the software systems in the majority of cases. Three out of the four companies got reached a better maintainable system at the end of the refactoring phase. We observed however, that the first company preferred low-cost modifications, hence they performed only two types of refactorings from which removing unnecessary constructors had a controversial effect on maintainability. Another observation was that it was sometimes counter productive to just blindly apply the automatic refactorings without taking a closer look at the proposed code modification. On several occasions it transpired that the automatic refactoring tool asked for user input to be able to select the best refactoring option, but developers used the default settings because this was easier. Some of these refactorings then introduced new coding issues, or failed to effectively remove the original issue. So human factor is still important, but the companies were able to achieve a measurable increase in maintainability just by applying automatic refactorings.

Last but not least, this study shed light on some important aspects of measuring software maintainability. Some of the unexpected effects of refactorings (like the negative impact of removing unnecessary constructors on maintainability) are caused by the special features of the maintainability model applied.

The fact that developers tested the tool on their own products provided a real-world test environment. Thanks to this context, the implementation of the toolset was driven by real, industrial motivation and all the features and refactoring algorithms were designed to fulfill the requirements of the participating companies. We implemented refactoring algorithms for 40 different coding issues, mostly for common programming flaws. By the end of the project the companies refactored their systems with over 5 million lines of code in total and fixed over 11,000 coding issues. FaultBuster gave a complex and complete solution for them to improve the quality of their products and to implement continuous refactoring to aid their development processes.

Model-Queries in Anti-Pattern Detection

We evaluated different query approaches for locating anti-patterns for refactoring Java programs. In a traditional setup, an optimized Abstract Semantic Graph was built by SourceMeter, and processed by hand-coded visitor queries. In contrast, an EMF representation was built for the same program model which has various advantages from a tooling perspective. Furthermore, anti-patterns were identified by generic, declarative model-queries in different formalisms evaluated with an incremental and a local-search based strategy.

Our experiments that were carried out on 28 open source Java projects of different size and complexity demonstrated that encoding ASG as an EMF model results in an up to 2-3 fold increase in memory usage and an up to 3-4 fold increase in model load time, while incremental model queries provided a better run time compared to hand-coded visitors with 2-3 orders of magnitude faster execution, at the cost of an increase in memory consumption by a factor of up to 10-15. In addition, we provided a detailed comparison between the different approaches that enabled them to select one over the other based on the required usage profile and the expressive capabilities of the queries.

To sum up, we emphasize the expressiveness and concise formalism of pattern matching solutions over hand-coded approaches. They offer a quick implementation and an easier way to experiment with queries together with different available execution strategies; however, depending on the usage profile, their performance is comparable even on 2,000,000 lines of code.

C
Összefoglaló

A karrierje egy pontján minden programozó szembesül egy olyan kódrészlettel aminek működését senki sem érti és senki sem szeretne hozzányúlni, nehogy véletlen elrontsa. A kérdés az, hogy hogyan keletkezett ez a siralmas kódrészlet. Feltételezhetően senki sem önszántából írta ilyenre. Sokkal valószínűbb, hogy a programunk a *szoftver erózió* áldozata, amely a szoftver egész életciklusára – legyen az fejlesztés vagy karbantartás – jellemző folyamatos hanyatlás.

A szoftver eróziója elkerülhetetlen. Egy szoftverrendszer folyamatosan fejlődik az idő múlásával: új funkciók kerülnek bele, korábbi funkciók módosulnak vagy tűnnek el; egyszóval igazodik az új igényekhez és környezethez. Ezek velejárója, hogy a forráskód általában bonyolultabb lesz és egyre inkább eltávolodik a kezdeti állapotától. Következményképp megnő a szoftver karbantartásának költsége. Ez is nagyrészt hozzájárul ahhoz, hogy a szoftverfejlesztési költségek nagyobb része (kb. 80%) a karbantartásra megy el [10]. A költségek csökkentéséhez redukálni kell a szoftvererózió okozta hatást, és növelni kell a karbantarthatóságot rendszeresen végrehajtott refaktoring műveletek segítségével.

A refaktoring kifejezés aztán vált népszerűvé, hogy Fowler publikálta katalógusát a refaktoring átalakításokról [12]. Ezen átalakítások célja az úgy nevezett "bűzlő" kódok helyrehozása. Itt a "bűzlő" szó a nehezen karbantartható vagy rosszul megkonstruált kódrészekre utal. Ilyen például, ha egy metódus nagyon hosszú, vagy ha egy metódus szinte másolta egy másiknak. Az ilyen "bűzlő" szerkezetek megértése segítséget nyújt, hogy felfedjünk hibákat és antimintákat, amik a valós problémákat jelentik a szoftverben. Ezek kiiktatása jobb minőségű szoftvert eredményez.

Mindenki közös érdeke, hogy a szoftver karbantarthatósága megmaradjon könnyűnek. A felhasználók így hamarabb kapnak új funkciókat, melyekben kevesebb hiba lesz. Emellett a fejlesztőknek is egyszerűbb dolguk lesz a kód módosításával, és a fejlesztőcégeknek is csökken a karbantartásra költött költségük. Jó karbantarthatóságot nagyon részletes specifikációval és alaposan kidolgozott tervekkel lehet a legjobban elérni. Azonban olyan helyzetek nagyon ritkán adódnak, ahol ezek a feltételek teljesülnek. Mivel a legtöbb szoftverre inkább a állandó fejlődés jellemző, ezért a gyakorlatban az időről-időre történő folyamatos refaktoring hatékonyabbnak bizonyult a könnyű karbantarthatóság szinten tartására. A tevékenység okán a kód "friss" marad és megnövekedik az élettartama.

Jelen tézis célja, hogy elősegítse a szoftverrendszerek karbantartását automatizálással. Kiváltképpen olyan módszertanok, technológiák és eszközök kidolgozásával foglalkozik, amik az alábbi témakörökre terjednek ki:

- Szoftverfejlesztők viselkedésének elemzése kézzel írott és gépi refaktoring tevékenységek közben.
- A refaktoring szoftver minőségre gyakorolt pozitív és negatív hatásának kiértékelése.
- Lokális-keresésre épülő antiminta felismerés átdolgozása modellalapú technológiára általános és gráfillesztéses módszerekkel.

Szoftverfejlesztők tevékenységeinek elemzése

Kísérleteink fő motivációja, hogy kiderítsük miképpen refaktorálnak a fejlesztők ipari környezetben abban az esetben ha rendelkezésükre áll minden szükséges erőforrás (pénz és idő) ehhez. A felmérésünket hat olyan nagyméretű, ipari, Java projekten végeztük melyek különböznek méretben és komplexitásban. A rendszereken folytatott refaktoringok tanulmányozása közben megfigyeltük mely típusú hibákat javítják a fejlesztők leginkább és mely refaktoringok bizonyultak a legjobbnak bizonyos szempontok alapján. Megvizsgáltuk a refaktoring kommitok forráskód minőségre gyakorolt hatását a ColumbusQM minőség modell segítségével [20].

A vizsgálat során azt találtuk, hogy a fejlesztők megpróbálták optimalizálni a refaktoring folyamatukat és először inkább konkrét kódolási szabálysértéseket kezdték el javítani, mintsem metrikák vagy antiminta detektorok által sugallt problémákat. További elemzések azt mutatták, hogy egy refaktoring hatása a szoftver termék globális karbantarthatóságra nehezen megítélhető. Néha előfordulnak olyan esetek is, ahol egy refaktoring rontja a globális karbantarthatóságot. Azonban maga a refaktoring folyamat hosszabb időre kivetített hatása jelentős javulással járhat a szoftver minőségre, amely ki is mutatható a minőség modell segítségével. Ennek oka nem csak a karbantartási munkálatok végzése, hanem hogy a folyamat során a fejlesztők megtanulnak egyre karbantarthatóbb kódot írni.

Automatikus refaktorálás előnyeinek és hátrányainak vizsgálata

Kísérletink kivitelezéséhez szükségünk volt automatikusan végrehajtható refaktoringok kifejlesztésére. Ebből kifolyólag alkottuk meg a FaultBuster nevű, automatikus refaktoring keretrendszert, amely képes számos kódolási hiba eltüntetésére automatikus refaktoring műveletek végrehajtásával. A FaultBuster alapjaként szolgáló folyamatot részleteiben bemutattuk. A folyamat során felhasználtunk egy harmadik féltől származó statikus elemzőt, amely refaktoring lehetőségeket javasol. Létrehoztunk egy algoritmust, ami képes a forráskód elemek megtalálására a saját elemzési fánkban pusztán szövegpozíció adatok alapján. Az algoritmus átalakítja a forráskódot kereshető geometriai térbe azáltal, hogy épít egy térbeli adatbázist.

A keretrendszer tervezése során a fejlesztők több elvárását is figyelembe kellett vennünk: a teljesítményt, eredmény kód formázását és tagolását, a folyamat érthetőségét, és a kijavítandó probléma precíz detektálását. A felsoroltak közül több is komoly kihatással van a refaktoring eszköz használhatóságára és ezért már a tervezéi folyamat elejétől fogva számolni kellett velük. A fejlesztés befejeztével átfogó kiértékelésnek vetettük alá az eszközt, amely alátámasztotta, hogy a módszer alkalmazható a gyakorlatban is és látható eredményekkel szolgál.

A kiértékelés során több érdekes visszajelzést kaptunk az eszközünket használó fejlesztőktől. Egyrészt kiderült, hogy a résztvevők szerint könnyen implementálhatók az általuk végzett kézi refaktoringok automatikusan működő átalakításokká. Másrészt az is megállapítható, hogy az implementált automatikus megoldások szinte minden esetben segítettek nekik a forráskódjuk továbbfejlesztésében.

A QualityGate SourceAudit eszközének alkalmazásával megvizsgáltuk a karbantarthatóságbeli értékeket különböző refaktoring műveletek előtt és után. Az elemzés kimutatta, hogy egy kivételével minden FaultBuster által támogatott kódolási szabálysértés kijavítása nyomon követhetően pozitív hatással volt a szoftver rendszerek minőségére. A kísérletben résztvevő négy partnercég közül három számszerűsíthetően jobban karbantartható rendszerrel fejezte be a projektet a refaktoring fázis végén. Ugyanakkor, azt is észre vettük, hogy az egyik cég kizárólag költséghatékony átalakításokat keresett, és ezért csak két fajta refaktoringot hajtott végre. Ezek közül az egyik a nem-használt konstruktorok eltávolítása volt, amely gyakran negatív hatással van a karbantarthatóságra. Egy másik megfigyelés pedig arra enged következtetni, hogy a felkínált gépi refaktoringok vakon való elfogadása legtöbbször célszerűtlen. Gyakran előfordult, hogy amikor a refaktoring eszköz a felhasználótól kérte, hogy válassza ki a szerinte a helyzethez legmegfelelőbb refaktoring beállítást, akkor a fejlesztők lustaság miatt az alapbeállításokat választották, ahelyett, hogy komolyabban áttanulmányozták volna a helyzetet. Ezért néhány ilyen refaktoring nem orvosolta az eredeti problémát, sőt, néha egészen új kódolási hibákat vezetett be. Tehát az emberi oldal még itt is fontos tényező, de ennek ellenére is sikerült a partnercégeknek jelentősebb javulást előidézni a szoftverük minőségében csak azáltal, hogy gépi refaktoringokat alkalmaztak.

A kutatásunk rávilágított néhány érdekességre a szoftver karbantarthatóság mérésével kapcsolatban is. Például, hogy a refaktoringok pár nem várt következménye (mint az, hogy a nem-használt konstruktorok törlése negatív hatással van a karbantarthatóságra) az alkalmazott minőségmodell különlegessége.

A FaultBuster fejlesztéséhez nagyon jó teszt környezetet biztosított, hogy a eszközt a fejlesztők a saját valós rendszereiken tesztelték élesben. Ennek a környezetnek hála az eszköz továbbfejlesztését valós, ipari célok motiválták és minden képessége és refaktoring algoritmusa úgy lett megtervezve, hogy kielégítse a projektben résztvevő cégek igényeit. A kutatás során olyan refaktoring algoritmusokat fejlesztettünk ki, amelyek képesek 40 különböző fajta tipikus kódolási hiba kijavítására. A projekt végeztével a partnercégek refaktorálták az összesen több, mint 5 millió kódsorból álló kódbázisukat, és több, mint 11,000 szabálysértést elimináltak. A FaultBusterrel a cégek kaptak egy megoldást, amivel elősegíthetik a szoftvereik minőségének javulását és segítségével be tudják építeni a folyamatos-refaktorálás módszertanát az fejlesztési folyamataikba.

Modell-alapú módszerek az antiminta detektálásban

Ebben a fejezetben többfajta forráskód-elem kereső módszert értékeltünk ki. E módszerek segítségével refaktorálható antimintákat detektálhatunk a kódbázisunkban. Korábbi kísérleteink során hagyományosan egy optimalizált absztrakt szemantikus gráfot építettünk fel a SourceMeter segítségével, melyet később kézzel írt *vizitor* lekérdezésekkel dolgoztuk fel. A kutatás során megnéztük, hogy mennyiben változik a teljesítménye a lekérdezéseknek, ha az alatta fekvő reprezentációt lecseréljük EMF-re. Az EMF támogatásnak hála, kipróbálhattuk miben változik a teljesítmény, ha átírjuk a lekérdezéseket modell-alapú kérdéseké, különböző formalizációval, mind inkrementális, mind pedig lokális-keresés alapú stratégiákkal.

A kísérleteinket 28 darab különböző méretű és komplexitású, nyílt forráskódú Java projekten végeztük el. Eredményeink azt mutatják, hogy az EMF-be kódolt modellt 2-3-szoros memórianövekedés és körülbelül 3-4-szeres modell betöltési idő jellemzi. Az inkrementális modell lekérdezések futásideje 2-3 nagyságrenddel gyorsabb volt, mint a kézzel kódolt lekérdezéseké, de ez egy további 10-15-szörös memórianövekedéssel járt. Az eredményekről egy részletes összehasonlítást készítettünk, aminek segítségével egyszerűbben eldönthető, hogy saját projektünk tulajdonságaihoz melyik módszer választása a legkifizetődőbb.