# Fighting Software Erosion with Automated Refactoring

Gábor Szőke

Department of Software Engineering
University of Szeged

Szeged, 2019

Supervisor:

Dr. Rudolf Ferenc

Summary of the Ph.D. thesis submitted for the degree of Doctor of Philosophy

of the University of Szeged

University of Szeged

PhD School in Computer Science

# Introduction

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 [4]. 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 [3]. 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.

## Goals of the Thesis

In this theses, our goal is to contribute to the automated support of software system maintenance. In particular, we 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.

This thesis research is driven by the following research questions:

**RQ1:** *What will developers do 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?*

Our results have been grouped into 6 contributions divided into three main parts according to the research topics. In the remaining part of the thesis summary, the following thesis points will be presented:

**I Evaluation of Developers' Refactoring Habits**

(a) Analysis of software developers behavior during hand-written refactoring tasks

(b) Empirical study on refactoring industrial systems and its effects on maintainability

**II Challenges and Benefits of Automated Refactoring**

(a) An automatic code smell refactoring toolset

(b) Industrial case study on automatic refactorings

**III Applications of Model-Queries in Anti-Pattern Detection**

(a) Anti-pattern detection with model queries

(b) Performance comparison of query-based techniques for anti-pattern detection

## Publications

Most of the research results presented in this thesis were published in journals or proceedings of international conferences and workshops. Table 1 is a summary of which publications cover which results of the thesis.

| No. | Contribution - short title | Publications |
|---|---|---|
| **I**. | Case study of a large-scale refactoring project | [7], [6], [11] |
| **II**. | An automated refactoring framework and an industrial case study | [8], [9], [10], [12] |
| **III**. | Benchmarking different anti-pattern detection techniques | [13], [14] |

Table 1: Thesis contributions and supporting publications.

# I 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 part, 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.

## Developers' Insights on Hand-written Refactoring Tasks

Much of the research work presented here was motivated by EU and Hungarian Government supported R&D project, called the *Refactoring Project*. One of the major goals of the 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.

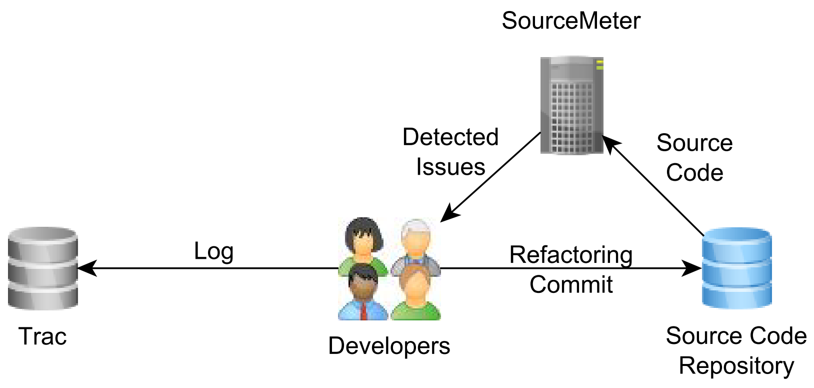


Figure 1: Overview of the refactoring process.

Figure 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 [2]. 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.

The evaluation of the survey showed us that developers choose to fix coding rule violations more often than any other indicators of code issues. In 85% of the cases rule violations were preferred, in opposed to bad smells, duplicated code segments and metric-based threshold violations.

Next, results indicate that the most challenging refactorings were the ones where the developers had to reduce code size, such as removing code clones. Similarly, removing empty code turned out to be quite a challenge as well. Some developers noted that they spent hours investigating empty `if` statements and the reasons what caused them. Other time consuming refactorings were in connection with security code guidelines and optimization rules.

We observed that developers ranked those refactorings the riskiest which included rules concerning basic Java functionalities, for example, `equals`, `hashCode`, or `java.lang.Object`'s `clone` method. In addition, we learned that they optimized the refactoring process and started fixing more serious issues first.

## Case Study on the Effects of Refactoring on Software Maintainability

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. Table 2 shows the size of the six chosen systems and the number of analyzed revisions including the number of refactoring commits.

Table 2: The main characteristics of the selected systems.

| 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** |

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.

The first diagram in Figure 2 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 2 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 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.

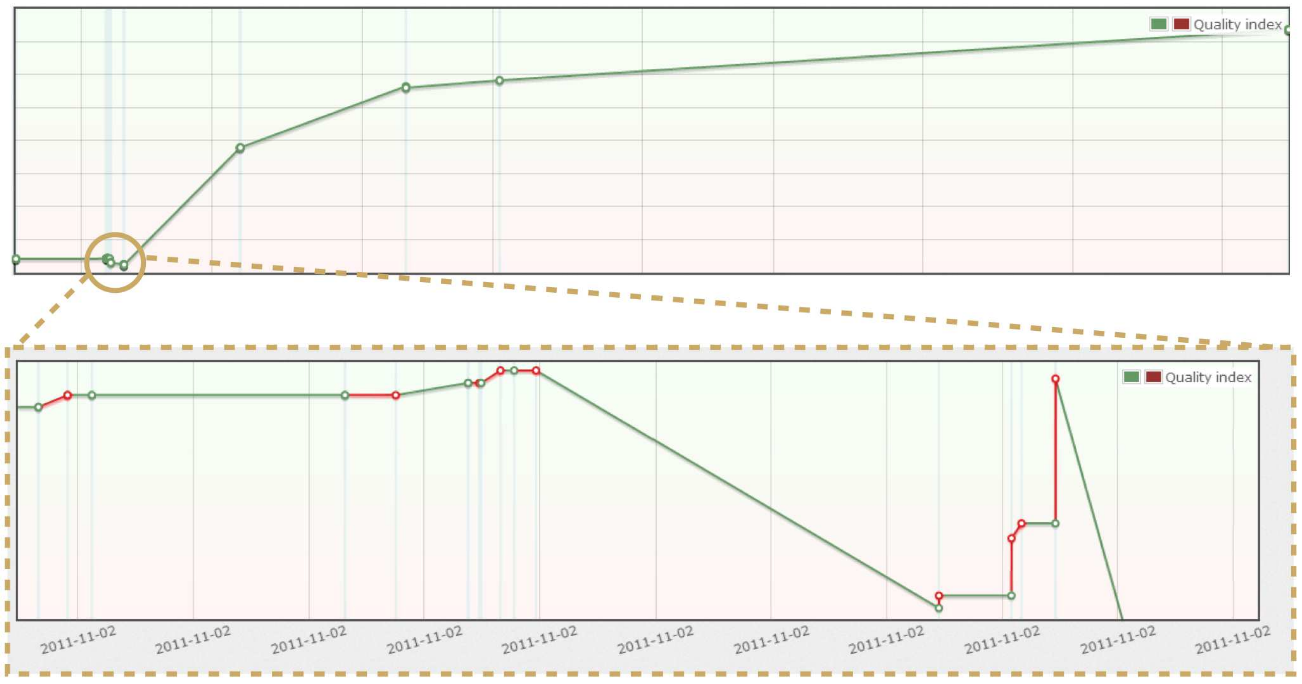


Figure 2: Maintainability of *System A* over the refactoring period and a selected subperiod where we highlighted in red the changes in maintainability caused by refactoring commits

Although each participating company in the Refactoring Project 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. 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. 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.

We found 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 (see Figure 3), 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.

## Own contributions

The author performed a survey on hand-written refactorings and evaluated the results. He performed an empirical case study on how manually written refactorings affect the software quality of a system. He identified commits which just included changes made by refactorings and performed measurements on them. He evaluated the results and concluded additional observations on the behavior of developers.

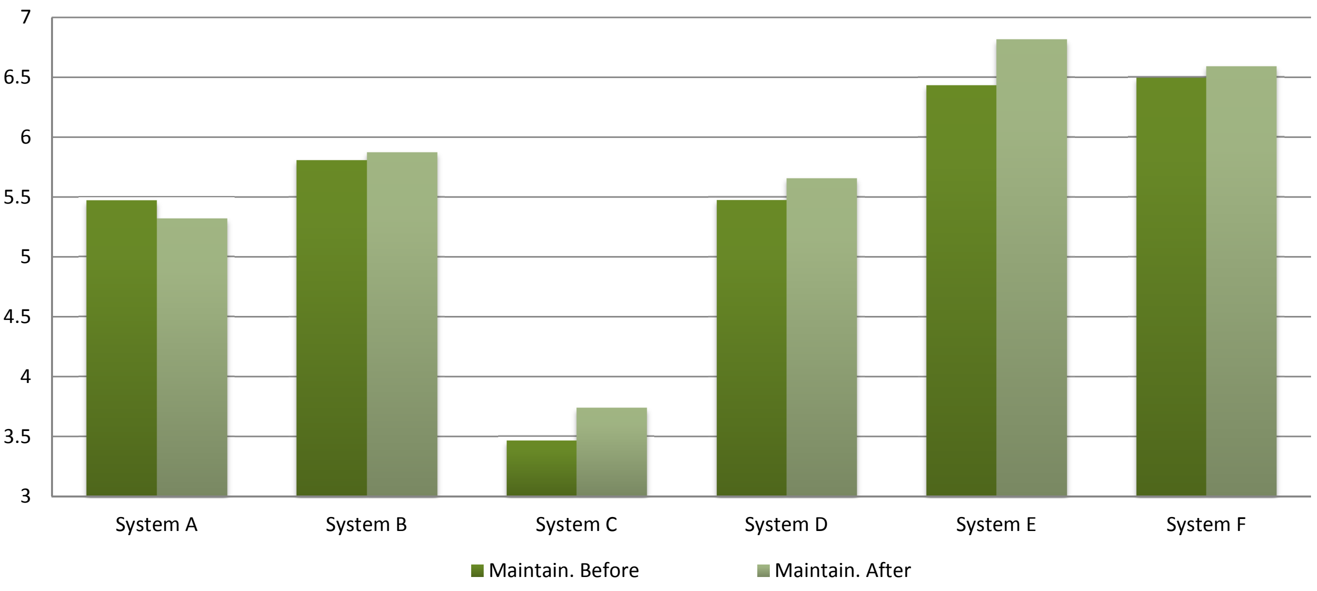


Figure 3: Maintainability of the projects before and after the hand-written refactoring period

## II Challenges and Benefits of Automated Refactoring

We will provide a 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.

### An Automatic Refactoring Framework for Java

Based on the results of the manual refactoring phase of the Refactoring Project, we designed an automatic code smell refactoring toolset, called FaultBuster. We created FaultBuster 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.

FaultBuster is a client-server application where the IDEs are the clients. Developers use the IDE plugins to select problematic code parts and send those to the server-side where they will get refactored and the diff will be sent back to the clients for review. The IDE plugin provides options for the developers to customize the refactoring solutions where they may chose which refactoring method they wish to apply, and they may set several parameters of the underlying transformation. What is more, the framework provides a ticketing system where developers can track the state of coding issues (i.e. open, under refactoring, untested, commited, rejected) and each issue may be assigned to developers and therefore concurrent modifications could be prevented.

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.

Refactoring transformations performed by FaultBuster are carried out via refactoring algorithms.

We implemented algorithms that can solve 40 different kinds of coding issues 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.

The input of such a refactoring 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. The coding issue is identified via the output of a third-party static analyzer, namely PMD. To fix the issue that the developer selected we have to locate the problematic code part in the ASG. Since we only got the textual position of the issue (PMD output has only file, line, and column details) we have to determine the proper node in the ASG to start the refactoring transformation. Hence, we created an algorithm (Reverse-AST-SeArch) 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.

## Evaluating the Connection between Automatic Refactorings and Maintainability

Developers tested FaultBuster on their own products which provided us with 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. 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. Table 3 gives a overview of the software systems selected in this study and the number of refactorings that was performed on them.

Table 3: 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** |

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, three out of the four companies involved in the Refactoring Project 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.

### Own contributions

The author's contribution was designing and implementing the Refactoring Framework. Although the IDE plugins was the work of the industrial partners, the author designed a Java Swing-based client that was used as a blueprint for the IDE plugins. The author also defined the process of the refactoring algorithms and integrated them into the framework. He implemented a survey into the Refactoring Framework and collected the results and assessed them. The author performed and evaluated the case study of automated refactorings and measured their effect on software maintainability.

## III Applications of Model-Queries in Anti-Pattern Detection

In FaultBuster we used a third-party coding rule violation detection tool called PMD. PMD is a widely used, open-source static analyzer tool in the Java community however it has some drawbacks. First, as we mentioned beforehand, 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. Third, the reports of developers indicated 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 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.

### Anti-pattern detection with model queries

In this part, we focused on the detection of coding anti-patterns. We investigated 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.

The first option was fulfilled by the static analyzer, SourceMeter. In order to assist the processing aspect of the model (e.g., executing a program query), SourceMeter API supports visitor-based traversal on the ASG. These visitors can be used to process each element on-the-fly during traversal, without manually coding the (usually preorder) traversal algorithm.

To fulfill the second option we created an EMF meta-model based on SourceMeter's ASG representation. Because its implementation of the Java ASG and the generated code from the EMF metamodel use similar interfaces, it 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.

Given the EMF model of the source code, we were able to use graph pattern matching. Graph patterns [1] 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. Here, we use it to implement anti-pattern detection program queries.

First, we created local-search algorithms. Local-search based pattern matching (LS) is commonly used in graph transformation tools, 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.

Later, as an alternative to local-search based pattern matching, we utilized the incremental graph pattern matching approach. Incremental pattern matching 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.

Lastly, we constructed Model Queries with OCL (Object Constraint Language). OCL [5] is a standardized, pure functional model validation and query language for defining expressions in the context of a meta-model. The language itself is very expressive, exceeding the expressive power of first order logic by offering constructs such as collection aggregation operations. OCL expressions can be evaluated as a search of the model, where the corresponding search plan is encoded in the expression itself.

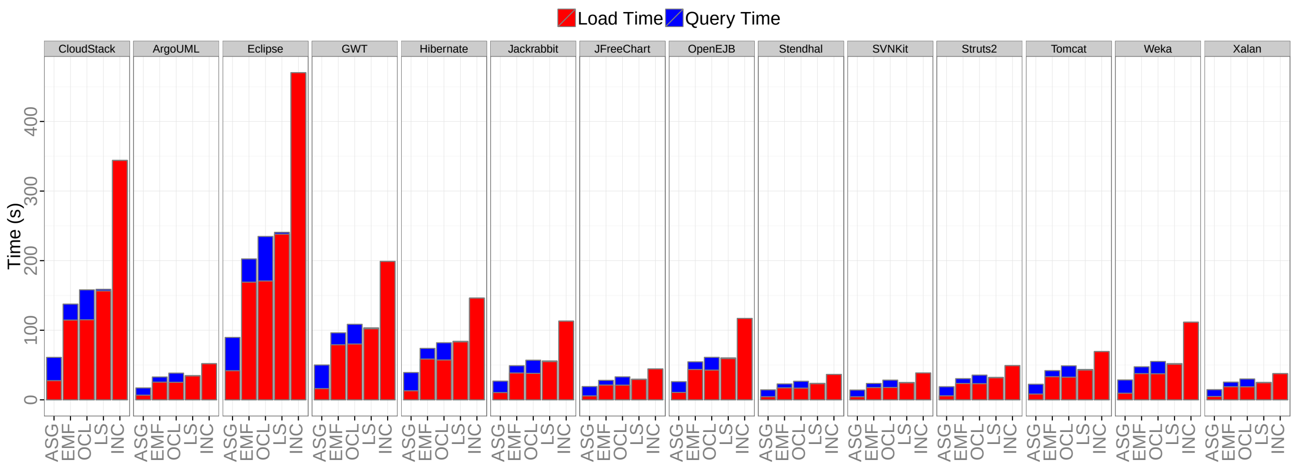
We selected six types of anti-patterns based on the feedback of project partners and formalized them as model queries. The diversity of the problems was among the most important selection criteria, resulting in queries that varied both in complexity and programming language context ranging from simple traverse-and-check queries to complex navigation queries potentially with negative conditions.

## Performance comparison of query-based techniques for anti-pattern detection

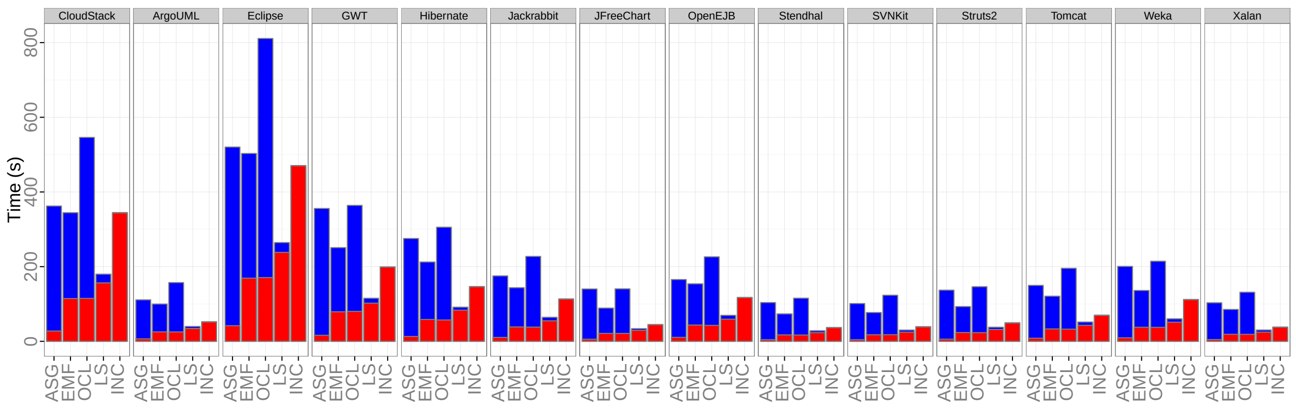
Several years ago, we found that typical modeling tools were able to handle only middle-sized program graphs [15]. We now re-exam this question and assess whether model-based generic solutions have evolved to compete with hand-coded Java-based solutions. Therefore, to provide a context for our performance evaluation we created a test set of $28$ open-source projects of varying size and complexity. On this set, every program query was executed ten times. We measured model load-time and query execution-time on different usage profiles: one-time, commit-time, and save-time scenarios.

Our experiments 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

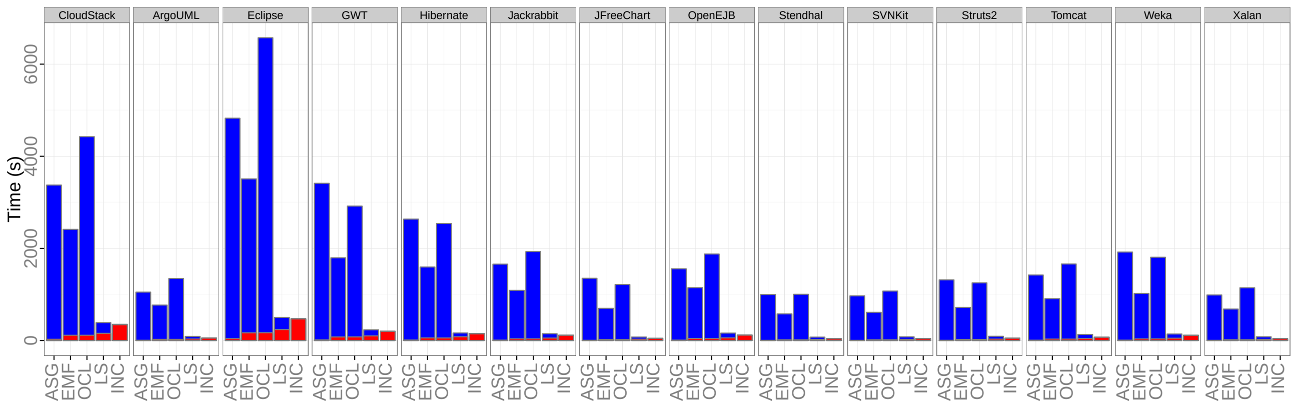
10-15. A comparison on execution times is available on Figure 4.



(a) One-time



(b) Commit time



(c) Save time

Figure 4: Execution Time over Models

Following this, we provided a detailed comparison of the different approaches and that made it possible to select one over the other based on the required usage profile and the expressive capabilities of the queries. Expressiveness and concise formalism of pattern matching solutions 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.

## Own contributions

The author adapted the Columbus ASG to Java and EMF platforms and created tools to utilize this version of the ASG. He created a test set of projects of varying size and complexity. He performed measurements on this set and extracted different characteristics of the projects. He performed the empirical validation of the measurements, and evaluated the results. The author created a decision model to represent the findings and to serve as a supplementary guide to aid the choosing of more suitable tools for the different usage scenarios.

# Summary

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 this thesis 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 ([13]) 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.

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 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, 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 affect maintainability?

We identified lots of refactoring commits throughout the project. First, it was 315 in the manual phase and later, 1,048 in the automatic phase. 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?**

We investigated the costs and benefits of using the popular industrial Eclipse Modeling Framework 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.

## Acknowledgements

First of all, I would like to express my gratitude to my supervisor Dr. Rudolf Ferenc, who kept me motivated while doing my studies and for inspiring me in times when I needed motivation. 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, June 2019*

## References

[1] *Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for EMF models. In* Theory and Practice of Model Transformations, *volume 6707 of* Lecture Notes in Computer Science, *pages 167–182. Springer Berlin / Heidelberg, 2011.*

[2] *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), *pages 172–181. IEEE Computer Society, October 2002.*

[3] *Martin Fowler.* Refactoring: Improving the Design of Existing Code. *Addison-Wesley Longman Publishing Co., Inc., 1999.*

[4] *Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. Characteristics of application software maintenance.* Communications of the ACM, *21(6):466–471, 1978.*

[5] *Object Management Group.* Object Constraint Language Specification (Version 2.3.1), *May 2012.* `http://www.omg.org/spec/OCL/2.3.1/`.

[6] *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, *pages 95–104, 2014.*

[7] *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, *pages 524–540, 2014.*

[8] *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, *pages 253–258, 2015.*

[9] *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, *pages 429–438, 2015.*

[10] *Gábor Szőke. Automating the refactoring process.* Acta Cybernetica, *23(2):715–735, Jun. 2017.*

[11] *Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Empirical study on refactoring large-scale industrial systems and its effects on maintainability.* Journal of Systems and Software, *2016.*

[12] *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, *pages 693–697, 2016.*

[13] *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, *pages 293–302, 2014.*

[14] *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.* Information & Software Technology, *65:147–165, 2015.*

[15] *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), *pages 232–237, July 2009.*