Cost-Effective Quality Assurance For Long-Lived Software Using Automated Static Analysis

Daniela Steidl

Technische Universität München

Institut für Informatik
der Technischen Universität München

Cost-Effective Quality Assurance For Long-Lived Software Using Automated Static Analysis

Daniela Steidl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Bernd Brügge, Ph.D.
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	Assoc. Prof. Andy Zaidman, Ph.D. Delft University of Technology/ Niederlande

Die Dissertation wurde am 05.11.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 26.01.2016 angenommen.

Abstract

Developing large-scale, long-lived software and adapting it to changing requirements is time and cost intensive. As many systems are maintained for decades, controlling their total life cycle costs is key for commercial success. With source code being the main software artifact, its quality significantly influences arising costs for maintenance. However, as many changes have to be performed under time pressure, suboptimal implementation decisions often lead to gradual code quality decay if no counter measures are taken.

To prevent code quality decay, automated static analysis is a powerful quality assurance technique that addresses different aspects of software quality—e. g. security, correctness, or maintainability—and reveals *findings*: Findings point to code that likely increases future costs in the life cycle of the software. Examples comprise hints to programming faults, vulnerabilities for security attacks, or maintainability problems like redundant code.

Although static analyses are widely used, they are often not applied effectively: When introduced to a grown, long-lived system, they typically reveal thousands of findings. As this number is too large and budget for quality improvements is usually limited, not all findings can be removed. Additionally, false positives disturb developers and new findings might be created when existing ones are removed. Due to these challenges, code quality often does not improve in the long term despite static analysis tools being installed.

This thesis provides an approach for using static analysis cost-effectively in practice. To reach this goal, we first analyze the state-of-the-practice of using static analysis and investigate the number and the nature of findings occurring in industry. As we show that not all findings can be removed at once, we argue that only those should be selected which reveal the best cost-benefit ratio. Consequently, we build a conceptual model which outlines costs and benefits of finding removal. From the model, we derive a prioritization approach which helps developers to effectively select a subset of all findings for removal. However, to actually remove the selected findings, developers also require allocated resources and, thus, management support. To convince management of the benefit of providing resources for quality improvement, we combine the prioritization approach with a continuous quality control process which provides transparency for managers. Further, we evaluate corresponding tool support necessary to apply the process in practice.

The combination of our prioritization approach, quality control process, and tooling enables the cost-effective usage of static analysis in practice. We evaluate the applicability and usefulness of the cost-benefit prioritization with several empirical studies, industrial case studies, and developer interviews. The evaluations show that the prioritization is appreciated by developers as quality-improving measure. We analyze the success of the quality control process with a longitudinal study with one of our industrial partners. The study shows that the number of findings can be decreased in the long term even if systems are still growing in size.

"But it ain't about how hard you hit. It's about how hard you can get hit and keep moving forward."

Rocky Balboa

Acknowledgements

I would like to thank everyone who supported me throughout my thesis. First of all, I thank Prof. Manfred Broy for supervising my thesis, for the fruitful scientific discussions, for his invitations to every *Lehrstuhlhuette*, and also for providing me my own office desk at university. I always felt warmly welcomed in his lab. My thanks go to the second member of my PhD committee, Prof. Andy Zaidman. His careful and detailed reviews as well as his remote support helped me tremendously in the final period of the PhD.

I would like to thank my company CQSE, my colleagues, and my customers, without whom this industry PhD would have not been possible. I greatly enjoyed the opportunity to conduct industry-oriented research while being closely in touch with our customers as a software quality consultant. I deeply appreciated the company environment providing first-hand industry experience combined with a surrounding scientific mind-set.

In particular, my thanks go to my advisor Dr. Florian Deissenboeck for his countless, honest and direct reviews of my work, numerous profound and inspiring scientific discussions, and his moral support in moments of frustration. I would also like to thank Dr. Elmar Juergens for his very helpful input on numerous paper stories and the thesis story. Additionally, my thanks go to Dr. Benjamin Hummel for his technical advice and support and for guiding me throughout my bachelor and master thesis as well as guided research. Without having known him, I would not have dared to do an industry PhD in such a small and young company—which my company was at the time I started my PhD.

I would also like to thank all my colleagues at university. They helped me to broaden my scientific horizon and prevented me from becoming a lone warrior. In particular, I thank Benedikt Hauptmann for his support, both scientifically and morally. He kept me grounded on earth and helped me to take it simply step after step. I also thank Sebastian Eder for his co-authorship on our best-paper award, his always available help, and for being the best office mate possible! I really enjoyed the days I was able to spend at university; they were primarily filled with laughter and joy. I certainly will miss these days and my colleagues.

Outside of work, I thank my underwater hockey team in Munich for coping with my aggressive playing style and my bad mood when I was not proceeding with my PhD as fast as I wanted to! Hockey has been the best way to get refreshed. The sport, its people, and their friendship have been my greatest source of energy.

Last but not least, I thank my parents for the way they raised me, for the possibility to study without any financial worries, for countless supporting phone calls, and for encouraging me to reach my goals day after day.

Publication Preface

The contribution of this thesis is based on the following five first-author papers.

A D. Steidl, B. Hummel, E. Juergens: *Incremental Origin Analysis for Source Code Files.* Working Conference on Mining Software Repositories, 2014, 10 pages, [89]

http://doi.acm.org/10.1145/2597073.2597111

B D. Steidl, F. Deissenboeck: *How do Java Methods Grow?.* Working Conference on Source Code Manipulation and Analysis, 2015, 10 pages, [84]

C D. Steidl, N. Goede: *Feature-based Detection of Bugs in Clones.* International Workshop on Software Clones, 2013, 7 pages, [87]

D D. Steidl, S. Eder: *Prioritizing Maintainability Defects Based on Refactoring Recommendations.* International Conference on Program Comprehension, 2014, 9 pages, [86]

http://doi.acm.org/10.1145/2597008.2597805

E D. Steidl, F. Deissenboeck et al: *Continuous Software Quality Control in Practice.* International Conference on Software Maintenance and Evolution, 2014, 4 pages, [85]

The other publications with major contributions as second author are also included.

F L. Heinemann, B. Hummel, D. Steidl: *Teamscale: Software Quality Control in Real-Time.* International Conference on Software Engineering, 2014, 4 pages, [40]

http://doi.acm.org/10.1145/2591062.2591068

Contents

"The bitterness of poor quality remains long after the sweetness of meeting the schedule has been forgotten."

Anonymous

1 Introduction

Developing large-scale software is a time and cost intensive challenge. As many systems in the business information or embedded domain are maintained for decades, controlling their total life cycle costs is key for commercial success. However, between 60% and 80% of the life cycle costs of long-lived systems are spent during the maintenance phase rather than during initial development [12, 33, 37, 63, 71]. Hence, to control the total life cycle costs, maintaining the system becomes crucial, i. e. adapting it to changing requirements and performing change requests correctly and in a timely manner.

The ability to control total life cycle costs strongly correlates to the product quality of software. Based on the ISO definition, software product quality comprises different aspects, e. g. functional suitability, security, maintainability, or portability [43, 44]. With source code being the essential core of a software product, its quality significantly impacts the overall quality [8, 13, 14, 28]. However, code quality often gradually decays due to software aging and continuous time pressure. Consequently, total life cycle costs increase when no effective counter measures are taken [31, 55, 61, 74]. In fact, many of today's systems reveal decayed quality because they have been developed on the *brown-field*, i. e. their code base grew over a long time without strong quality assurance [85].

Low product quality increases the costs to maintain software [14, 24, 55]. In this thesis, we approximate product quality with the system's *non-conformance* costs during the maintenance process [21, 24, 51, 80, 82, 99]: In contrast to *conformance costs* as the amount spent to achieve high software quality (e. g. costs for training staff, testing before deployment), non-conformance costs include all expenses resulting from the failure of achieving high quality. Non-conformance costs comprise both *internal failure costs* before the software is delivered (e. g. costs for restructuring the code) as well as *external failure costs* arising from system failure in production (e. g. costs for fixing field defects, retesting). Modeling code quality in terms of non-conformance costs is established, but yet not widely spread [80, 82, 99].

In literature, gradual software quality decay leading to an increase of non-conformance costs has also been referred to as the accumulation of *technical debt* [17, 18, 22, 59, 65, 72]. Technical debt metaphorically describes reduced maintainability due to shortcuts taken during development to increase delivery speed. Just like actual debts, technical debt has to be paid back eventually and the longer it is not repaid—i. e. code not being refactored—the more interests are to be paid—i. e. additional time developers spend on code understanding and code modification. While this part of the metaphor is very intuitive and comprehensive, the metaphor also creates the temptation to calculate a debt value in money currency [62, 79]. As the calculation of a specific debt value is heavily influenced by many context factors, we abstain from using the term *technical debt* in this work.

To reduce and control non-conformance costs in the overall life cycle, gradual quality decay has to be prevented. For this purpose, automated static analyses are a powerful analytical quality assurance technique [75]. They analyze various different aspects of code quality [6, 19, 32, 58, 64, 81, 102] and reveal *findings*: Findings point to concrete locations in the source code that hamper quality, i. e. that are likely to increase non-conformance costs. For example, static bug pattern detection reveals functional defects. Structural metrics or redundancy measurements address the code's understandability and changeability. Whereas bug patterns point to a high risk of arising non-conformance costs from external failure, hampered changeability likely causes non-conformance costs from internal failure.

Even though automated static analyses are not capable of addressing the various quality aspects completely, they have several benefits when compared to other analytical measures such as dynamic analyses [96] or code reviews [9, 76]: First, they are easier to perform than dynamic analyses as they do not require a running system while analyzing the source code. They also can be applied effectively before the system is tested as they do not depend on runtime environment, surrounding systems, or underlying hardware [6, 32, 60]. Compared to fully manual approaches such as code reviews, automated static analyses provide more tool support and are cheaper to perform [60, 100]. Additionally, they can be used to enhance an expensive manual review process [73]. Consequently, they constitute a vital complement to the set of available quality assurance techniques [108]. For the remainder of this thesis, we use the term *static analysis* to refer to *automated static analysis*.

1.1 Problem Statement

Static analyses are widely known among developers [96] and also supported by a large variety of tools [11, 25, 32, 39, 40, 46, 77]. However, although many companies have static analysis tools installed, developers do not use them frequently [45], the number of findings keeps growing and the code quality of their systems often decays further [85]. Based on our industry experience of three years as software quality consultants working with customers from various different domains, we identified five major challenges developers usually face when applying static analyses. We identified these challenges across different domains based on numerous interviews conducted with developers, project managers, or chief information officers. While these claims are certainly not generalizable to all software engineering, we believe they yet apply to the substantially large subset of long-lived software.

Number of Findings When static analyses are introduced to software systems developed on the brown-field, they typically result in thousands of findings [15, 16, 32, 45, 54, 77, 86]: Removing all findings at once is not feasible as it would consume too much of current development time and budget for quality assurance is usually limited. Additionally, the risk of introducing new functional defects during an ad-hoc clean up session is likely to be bigger than the expected reduction of non-conformance costs. Hence, developers can only remove a subset of the revealed findings within the available time.

Benefit of Removal Even though many findings are relevant, they yet reveal different benefits on removal [39, 78]: Removing findings that point, e. g. to a programming fault reveals an immediate benefit. Hence, developers are often willing to remove them instantly

[6, 7]. Contrarily, removing findings that address maintainability provides only a long-term benefit. In the absence of immediate reward, developers often have a higher inhibition threshold to address these findings [78].

Relevance of Findings Among many relevant findings, static analyses also reveal *false positives* and *non-relevant* findings which disturb and distract developers [32, 39, 45, 77, 78, 100]. These findings should be eliminated from static analysis results while keeping the relevant findings.

Evolution of Findings Removing findings as a one-off event does not reduce the non-conformance costs in the long run: while removing existing findings, new findings can be introduced. Further, development continues and additional findings can emerge. These findings also have to be considered in the removal process.

Management Priority Change requests often dominate finding removal: Often, management is not convinced by the return-on-invest of using static analyses [60]. Under increasing time pressure and rising number of change requests, static analyses are easily omitted—management often prioritizes new features instead of assuring high code quality.

Problem Statement *Static analyses are often not applied effectively in practice: in brown-field software, they reveal thousands of relevant findings and budget for quality improvement is usually limited; they produce false positives disturbing developers; and they are ineffective as one-off events ignoring on-going development.*

1.2 Approach and Methodology

The goal of this thesis is to provide a cost-effective approach for using static analysis effectively as quality assurance technique, addressing the current challenges in practice. For this aim, we first analyze the current state-of-the-practice of using static analysis in industry. Based on the results, we derive subsequent steps. In the following, we outline the intermediate steps, which methodology we used and how the results of each step contribute to the overall goal of this thesis. The intermediate steps are visualized in Figure 1.1.

1.2.1 Understanding State-Of-The-Practice

First, we analyze the current state-of-the-practice of using static analysis. With a large-scale empirical study of 43 systems written in Java, CSharp or C/C++, we quantify how many findings are revealed by commonly used static analyses in industry and open source. In addition, we analyze the nature of the findings, i. e. how they affect the system's quality.

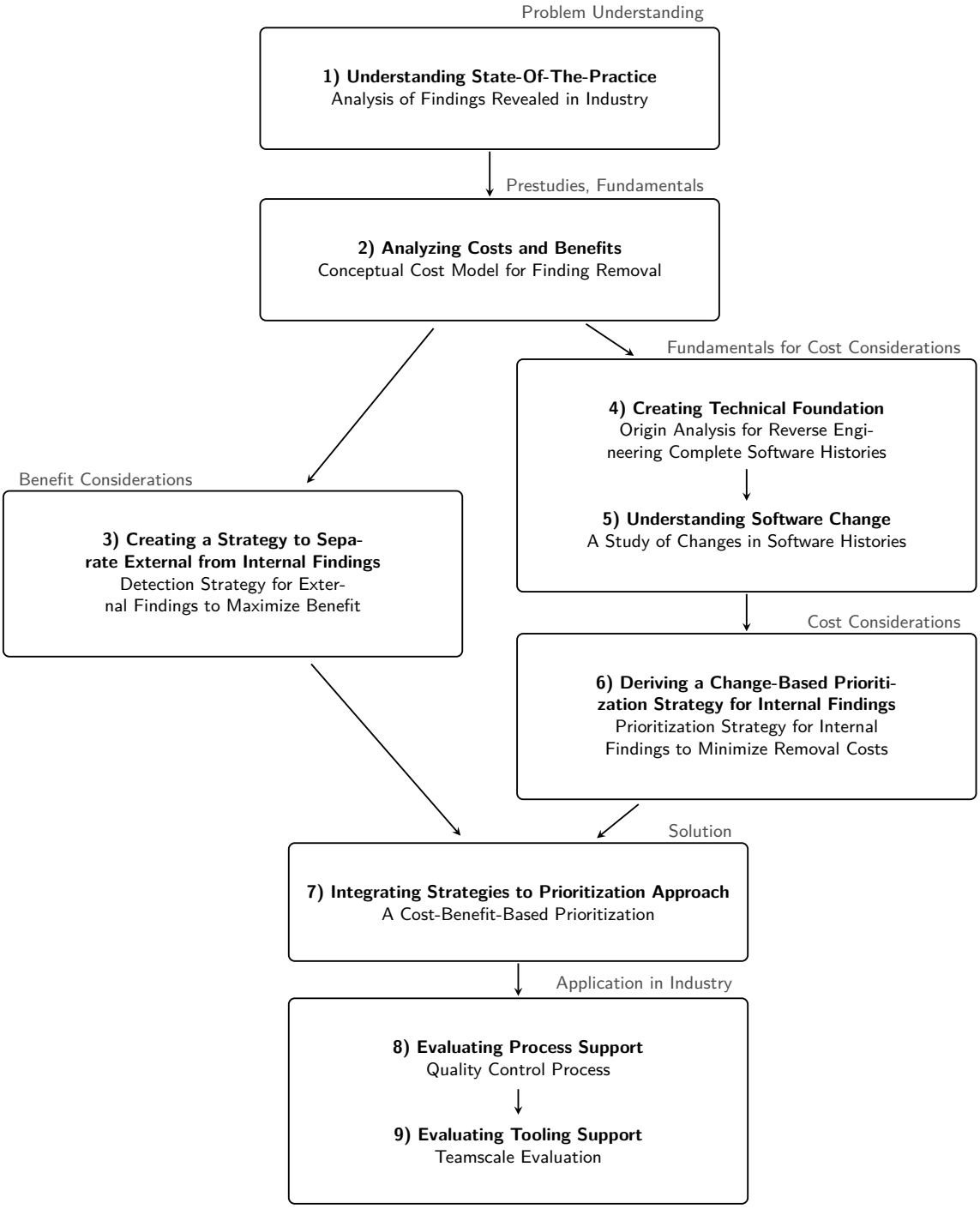


Figure 1.1: Overview of Approach and Methodology

Results The study shows that there are two kinds of findings: While *external* findings present a high risk of a system failure observable by a user, *internal* findings reveal code pieces that hamper the system's internal (non-user-visible) quality such as its maintainabil-

ity or portability. The two kinds of findings differ notably in number: In our study systems, external findings were small in number. In contrast, thousands of internal findings were already revealed by only few maintainability analyses.

Implications The number of findings is too large to be removed at once because quality assurance budget is usually limited. To effectively use the available budget, those findings should be selected which reveal the best cost-benefit ratio. The benefit of removing external findings is quite obvious—a potential system failure is eliminated. However, the benefit of removing internal findings is harder to quantify. In addition, little is known about the removal costs. In the next step, we deeper analyze costs and benefits of finding removal.

1.2.2 Analyzing Costs and Benefits of Finding Removal

To prioritize the large number of findings, we need a better understanding of the cost-benefit ratio of their removal. We provide a conceptual cost model that formalizes costs to remove a finding as well as associated benefit. We enrich the model with industrial evidence which we gathered in empirical studies of one Java system in the insurance domain. We use their issue tracking system to analyze costs and their version control system to approximate benefit. The empirical evidence allows us to base the model on realistic assumptions.

Results For external findings, the benefit of their removal is quite obvious, i. e. a potential system failure is eliminated. On the cost side, the model reveals that external findings are usually very easy to remove. As external findings reveal an immediate reward upon removal, our approach prioritizes them differently than internal findings: External findings should be removed as soon as possible to reduce costly system failures. The small number of external findings and their low removal costs make this prioritization feasible.

For internal findings, the model shows that removing them is beneficial only if the corresponding code is maintained again. Contrarily to commonly made assumptions [67], our empirical case study reveals that static analyses have limited impact to predict future maintenance. Future changes are usually rather derived from (unknown) changing requirements and evolving market situations. We assume that existing software artifacts generally cannot provide sufficient information for an automated maintenance prediction.

Additionally, internal findings create higher removal costs than external findings. As they are also much larger in number, only a small subset can be removed within one development iteration. While we cannot reliably prioritize based on the benefit of their removal, we can yet prioritize based on their removal costs. Low removal costs reduce the inhibition thresholds of developers and motivate them to remove internal findings despite missing immediate reward. The model shows that removing internal findings is less costly when aligned to ongoing development: If a finding is removed in code that has to be changed anyway, overhead in code understanding and testing can be saved—performing a change request already comprises both.

Implications The considerations from our cost model have two implications (see Figure 1.1). First, to prioritize external over internal findings, we need to accurately differentiate between the two. This differentiation is targeted by our next step in Section 1.2.3.

Second, to align the removal of internal findings with software changes, we have to better understand how software is actually changed. For that matter, the software's repository provides insights about its change history. However, history as recorded in version control systems is often not complete, because class or method refactorings are not recorded [83,89]. Hence, the history of a class or method before the refactoring is often lost. Incomplete histories of code entities result in inaccurate analysis results. To accurately analyze change evolution, we first need an approach to reverse engineer a complete history. We target the reverse engineering of history in Subsection 1.2.4.

1.2.3 Creating a Strategy to Separate External from Internal Findings

Based on the results from our cost model, our approach prioritizes the removal of external findings over the removal of internal findings. Hence, we need to accurately differentiate between the two. Sometimes, differentiating between external and internal findings is trivial: As, e. g. a missing comment cannot affect the execution of the program, it also cannot lead to user-visible failure. Thus, it represents an internal finding. A null pointer exception, in contrast, is an external finding as it likely represents a programming fault which can lead to a system failure. For other findings, however, this differentiation is more complex. For example, code clones can be internal findings as they increase the change overhead for maintenance. But they can also be external findings as they are error-prone if changed unintentionally inconsistently [49, 68].

Results In this thesis, we suggest an approach to differentiate between external and internal findings. In particular, for the complex example of code clones, we provide a classification algorithm that separates external from internal clone findings. In a case study, we reuse an existing industrial data set containing clones that were manually labeled by developers as unintentionally inconsistent or intentionally inconsistent [49]. Based on this data set, we evaluate the precision of our classification algorithm.

Implications The differentiation allows us to separate external from internal findings and prioritize them differently: External findings are selected for immediate removal. As a next step, we target the remaining internal findings. Based on the results from the cost model (see Section 1.2.2), we strive to align the prioritization of internal findings with ongoing software changes and, hence, need a better understanding of how software is changed.

1.2.4 Creating Technical Foundation

In order to perform an accurate analysis of software changes, we first need to obtain a complete software history even if code artifacts were refactored or moved. In this thesis, we provide an *origin analysis* which reconstructs the change history on file and method level.

Results and Implications The origin analysis detects refactorings and moves of files and methods and, hence, allows to track the complete history. We evaluate the results within a case study of seven open source systems in the Java domain. The recall is evaluated automatically based on information from version control systems. To evaluate precision, we use developer interrogation. Based on the origin analysis, we can now perform an accurate fine-grained study of how software systems are changed.

1.2.5 Understanding Software Change

The origin analysis allows us to gain deeper insights about how software systems evolve by conducting a case study of ten open source and industry systems in the Java domain. From the results, we strive to derive a cost-based prioritization for internal findings.

Results The study reveals that software systems are changed primarily by adding new methods, not through modification in existing methods. In fact, on average, half of the existing methods are not changed again after their initial commit.

Implications Having obtained a deeper understanding of how systems change, we address the original goal to prioritize the large number of internal findings (Section 1.2.2). Our next step describes how we derive a prioritization from the results obtained in this study.

1.2.6 Deriving a Change-Based Prioritization for Internal Findings

Combining the cost model with the results from the change study, we derive a prioritization of the large number of internal findings.

Results The cost model already showed that aligning finding removal with ongoing changes reduces the removal costs. As software grows through new methods, we prioritize internal findings first in newly added code and, second, in lately modified code. As many existing methods remain unchanged, this prioritization, in fact, reduces the number of internal findings significantly. We refer to this prioritization as *change-based* prioritization.

For highly dynamic systems, i. e. systems that are modified and extended extensively, the number of findings in new code might be still too large, however. In this case, further prioritization becomes necessary. We suggest to select only those findings that have the lowest removal costs. Therefore, we provide an approach to detect findings than can be removed with a very fast and, hence, cheap refactoring. We provide a case study on two industrial Java systems for the exemplary findings of code clones and long methods. We show how fast-to-refactor findings can be detected, and evaluate their acceptance for removal amongst developers based on individual interviews.

Implications Based on all intermediate results, we obtain a benefit-based strategy for external findings and a cost-based strategy for internal findings. It remains to combine both in an applicable approach that developers can apply effectively during development.

1.2.7 Integrating Strategies to Prioritization Approach

Combining previous results, we propose a prioritization approach to address the current challenges and to make static analysis successfully applicable in practice. The approach consists of two steps—an *automated recommendation* and a *manual inspection* of the findings. Both steps are applied *iteratively* during development.

Results To reduce the large number of findings, an automated prioritization mechanism *recommends* a subset of the findings for removal (addressing Challenge **Number of Findings**). To maximize benefit, external findings should be removed immediately in each iteration (Challenge **Benefit of Removal**). As they are usually small in number and easy to fix, their removal is feasible during an iteration. Internal findings are prioritized based on their removal costs: our approach recommends findings only in newly added or lately changed code. This change-based prioritization saves overhead in code understanding and testing. For highly-dynamic systems, our approach focuses only on the subset of findings that can be removed with a fast refactoring. Hence, they reveal the lowest removal costs.

After the recommendation, a *manual* inspection *accepts* (or rejects) the recommended findings for removal: The manual inspection can eliminate false positives and non-relevant findings (Challenge **Relevance of Findings**). As new findings can be introduced while existing findings are removed or while the system is further developed, both automated recommendation and manual inspection are to be applied *iteratively* during development. Hence, the approach takes the ongoing development and maintenance of the system into account (Challenge **Evolution of Findings**). Also, it can be flexibly adjusted to changing budget, team transitions, or developer fluctuation, or extended with additional analyses.

Implications Our approach allows developers to effectively select a subset of findings for removal. However, if management prioritizes new change requests over quality improvement, findings can still barely be removed due to missing resources. To lead to measurable success, the approach requires the collaboration of managers and developers.

1.2.8 Evaluating Process Support

To enable a collaboration of management and development, a process is required that meets the needs of both parties: developers require resources to remove findings and management requires transparency about the resource usage as well as its impact (Challenge **Management Priority**). To address both, we suggest a continuous quality control process which embeds the change-based prioritization of internal findings.

Results We evaluate the process with a longitudinal case study with one of our industrial partners: We use quantitative measures to reveal its success in terms of a long-term declining number of findings for several systems. Qualitatively, we rely on management decisions of the study object to extend to process to most of its development teams.

Implications The longitudinal study shows that our prioritization approach and its quality control process require substantial amount of tool support. As a final step, we evaluate which needs developers have with respect to tool support.

1.2.9 Evaluating Tooling Support

From the quality control process, we derive numerous requirements for static analysis tools to be able to support the process. As one example, our change-based prioritization requires the differentiation between findings in new or modified code and old findings. Many existing tools analyze only a snapshot of a system. Thus, they are not designed to provide this distinction. Teamscale, a tool developed by our company, however, analyzes the full history and provides findings tracking through all commits. Hence, Teamscale can accurately differentiate between newly added and old findings.

Results We provide an evaluation of Teamscale and its suitability for a control process. The evaluation is based on a guided developer survey. To familiarize the developers with the tool, we design several tasks which they completed with the help of the tool. We evaluate the correct completion of the tasks and, finally, conduct an opinion poll.

1.2.10 Summary

Based on a quantitative study of state-of-the-practice, a cost-benefit model, and a fine-grained analysis of software evolution, we derive a prioritization approach. The approach helps developers to effectively select findings for removal. Combined with a quality control process and supported by adequate tools, this approach enables the cost-effective usage of static analysis in practice. To obtain the necessary intermediate results and the final evaluation, we use different methodologies, comprising empirical studies, open-source and industrial case studies, developer interviews as well as developer surveys.

1.3 Contributions

This thesis includes several contributions to both industry and academia. In the following, we list each contribution and denote the corresponding paper when published.

- **Quantitative Study of Static Analysis Findings.** With a large-scale quantitative study on open-source and industrial systems, we show how many findings are revealed by commonly used static analyses. Future researchers can refer to this study to describe current state-of-the-practice of static analysis findings in industry.

- **Cost Model for Finding Removal.** In a conceptual cost model, we formalize costs and benefits associated with finding removal. The instantiation based on industrial evidence is a first step towards a return-on-invest calculation for project management in industry. The cost model further contributes to current research as it evaluates current state-of-the-art assumptions when justifying static analysis as quality assurance technique. It reveals weak assumptions that require future work.

- **Origin Analysis for Code Artifacts.** We provide an origin-analysis which enables an accurate and fine-grained analysis of software histories (Paper **A**). Our evaluation shows that our origin-analysis is highly accurate. The origin analysis is substantial for other researchers mining software histories to gain accurate results. Further, it supports developers as its application in quality analysis tools helps to obtain complete histories for code reviews, metric trend inspections, or change logs.

- **Fine-grained Study of Software Evolution.** Based on an open-source and industrial case study, we provide a theory of how software systems are changed and how they grow (Paper **B**). The study shows that software systems grow primarily through adding new methods and not through modification of existing methods. These results can serve as foundation for other researchers building a theory of software evolution.

- **Prioritization Approach for Static Analysis Findings.** As main contribution of our thesis, we provide a prioritization approach. The approach addresses the current challenges in practice and helps development teams to effectively use static analysis as quality assurance technique. As specific instantiations, it comprises a classification of inconsistent clones (Paper **C**) and also a detection approach for fast refactoring opportunities for code clones and long methods (Paper **D**).

- **Quality Control Process and Tool Support.** Finally, we show how the prioritization can be successfully applied within a quality control process (Paper **E**). Our evaluation serves as foundation to convince project management of the benefits of using static analysis as quality assurance technique: It shows that teams succeed to reduce the number of findings long-term even while the systems keep growing in size. Additionally, it reveals a management decision of one of our customers to extend the process to most of its development teams as they are convinced by its benefits. Beyond the process aspects, we further show which tool support developers require to apply the process successfully (Paper **F**).

1.4 Outline

The remainder of this thesis is organized as follows: Chapter 2 contains the terms and definitions for this work. Chapter 3 introduces the fundamentals of this thesis, i. e. the underlying models of software maintenance and software quality as well as an overview of current application and limitations of static analysis as quality assurance technique. Chapter 4 presents the quantitative study of static analysis results in industry. Chapter 5 contains our cost model for finding removal and its instantiation. Chapter 6 describes the iterative approach for finding prioritization. The remaining Chapter 7 contains the publications. Papers **A**–**F** are briefly summarized with one page each to show their role in the overall approach for finding prioritization. Related work for this thesis can be found in the papers and partly in Chapter 3 and 5. Chapter 8 concludes this work and presents an outlook to the future.

2 Terms and Definitions

This chapter presents the underlying terms and definitions used throughout this work.

2.1 Analysis Terms

When referring to static analyses as a quality assurance technique, we use the following terms:

Static Analysis Static analysis is "the process of evaluating a system or component based on its form, structure, content, or documentation" [2]. Static analysis does not require the actual execution of the system. Instead, static analysis operates on the code of the system, for example in form of source code or the object code, i. e. byte code in Java. Static analyses comprise fully automated analyses (supported by tools) or manual static analyses (carried out by a human, e. g. code reviews). In our thesis, we generally use the term *static analyses* if we refer to fully automated static analysis. Otherwise, we will explicitly use the term *manual static analysis*. Examples for static analyses comprise bug pattern detection, redundancy detection, structure analysis, or comment analysis.

Precision Many (automated static) code analyses come along with a certain *false positive* rate (see Section 2.2) and are, hence, not 100% *precise*: In the context of this work, we define precision as the number of true positives over the sum of true and false positives.

2.2 Finding Terms

The following terms are used when referring to findings as a result of a static analysis:

Finding Category One single static analysis is associated with potentially multiple finding categories. A finding category is used to group the different results of one analysis. For example, the redundancy detection can result in exact copies of code fragments (type I clones), syntactically identical copies (type II clones) or inconsistent copies (type III clones) [57]. Hence, depending on its configuration, the analysis can produce findings in three different categories. Another example is the structure analysis that can reveal overly long files, overly long methods, or deep nesting as three different categories.

Finding A Finding is a specific result instance of one static analysis in one finding category: Findings represent concrete locations in the code that hamper quality, i. e. that are likely to increase non-conformance costs [40, 64]. The scope of the concrete location of a finding can vary from a single code line (e. g. a missing comment) over a complete class (e. g. an overly long class) to multiple classes (e. g. code snippets duplicated between several classes).

False Positive We define a finding to be a false positive if it represents a technical analysis mistake—a false positive is a problem of the analysis itself. An example is a data flow analysis that cannot parse customized assertions: For example, developers use customized asserts to ensure that variables are not null. The assert will handle the null-case by, for example, throwing a customized exception. After the assert, the variable can be dereferenced. However, if the data flow analysis cannot parse the assert, it will mark the subsequent dereference as a possible null pointer dereference. This will be a false positive, as it is asserted that the variable is never null.

Non-Relevant Finding Even if a finding is a true positive, in few occasions, it might not be *relevant* for removal. In contrast to false positives, we define a non-relevant finding not to be a problem of the analysis, but of missing context information—information that is not available in the code itself. An example can be duplicated code in a system which stems from two duplicated, yet different specifications [86]: Technically, the code in the system is indeed a code clone. However, the duplication cannot be resolved by removing the duplicated parts, because each part serves its own specification. Even though both specifications use the same identifier names by chance, their meanings can be different. As both specification change and can diverge in the future, developers consider this clone to be not relevant for removal.

External Finding An external finding is a finding representing a high risk of leading to a system failure observable by a user.

Internal Finding Internal findings, in contrast, pose a risk for the internal quality of the system, e. g. its maintainability. These findings cannot be noted by the user of the system.

Finding Removal We refer to finding removal when a finding is removed from the code base—for example, with an appropriate code refactoring, a programming fault removal, or a fix of a security leak.

2.3 System Terms

In general, we use the following definitions when we describe a software system:

Code Type Different code serves different purposes. In this thesis, we differentiate between four types of code:

- *Application code* is code that is manually maintained and deployed with the application in production.
- *Test code* is code used for testing the application code. It is usually not deployed with the application.
- *Generated code* is code which is not manually written but automatically generated.

Application, test, and generated code constitute a partition of the code base. Amongst it, we further classify *unmaintained code*:

- *Unmaintained code* is dead code that could be deleted without changing the behavior of the system (if part of the application code) or without changing the tested functionality (if part of the test code). It comprises, for example, outdated experimental code, example code from external libraries, or outdated test cases.

"Quality is never an accident; it is always the result of intelligent effort."

John Ruskin

3 Fundamentals

This chapter provides the fundamentals for this thesis: It describes the underlying maintenance model (Section 3.1) as well as the underlying software quality model and introduces the concept of non-conformance costs as measurement for quality (Section 3.2). It further gives an overview of existing quality assurance techniques (Section 3.3) and outlines the impact and limitations of automated static analysis as quality assurance technique (Section 3.4 and 3.5). Section 3.6 summarizes this chapter.

3.1 Activity-Based Maintenance Model

As fundamental model for this thesis, we rely on an activity-based view of software maintenance. Based on the three maintenance characteristics as defined by Boehm [13]—modifiability, testability, and understandability—we define maintenance to consist of three main activities [24, 27]:

Definition 1 (Software Maintenance) *Software maintenance comprises three main activities: code reading, code modification, and code testing.*

This definition is still very coarse and has already been refined in related work. For example, Mayrhauser refines the activities based on adaptive, perfective, and corrective maintenance tasks [98]. Depending on the task category, code modification, for example, is subdivided into code changes, code additions, or code repairs. Additionally, Bennett and also Yau specified the *change mini-cycle* [10, 107] to consist of planning phase, program comprehension, change impact analysis, change implementation, restructuring for change, change propagation, verification and validation, and re-documentation.

For the purpose of this work, however, we can rely on the most simple definition because it suffices our context of *finding removal* (for definition, see Chapter 2). To remove most static analysis results, the change is rather small and usually only affects few files. Activities such as request for change, planning phase, change impact analysis, change propagation, or re-documentation on a system-level can be neglected for such small change.

3.2 Software Product Quality Model

During ongoing maintenance, continuous change can impact the quality of software. Often, quality gradually decays if no effective counter measures are taken [31, 55, 61, 74]. Independent from software, quality in general “is a complex and multifaceted concept” as

recognized by Garvin already in the 1980s [36]. Specifically for software products, quality has a magnitude of different facets.

3.2.1 ISO Definition

In this thesis, we refer to the ISO 25010 Standard [44] (see Figure 3.1):

Definition 2 (Software Product Quality) *Software product quality can be described with eight characteristics—functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability.*

We chose this quality model as it is widely spread and generally accepted: even though it is often criticized for its lack of measure-ability and, hence, its applicability in practice [50, 91, 101], it yet outlines the diversity of software product quality and provides a suitable classification for state-of-the-art static analyses (see Section 3.3).

In the quality model, functional suitability describes whether the software provides all of its specified functionality and whether it provides it correctly. Performance efficiency covers how efficiently the software performs given its resources. Compatibility describes the degree to which the system can exchange information with other systems using the same hardware or software environment. Usability refers to the user experience (effectiveness, efficiency, satisfaction). Reliability comprises the availability of the software, its fault tolerance, or its recoverability, for example. Security covers aspects of data protection by the system such that only authorized persons can access the data. Maintenance describes the effectiveness and efficiency with which the system can be modified. Portability examines how well the system can be transferred from one hard- or software platform to another.

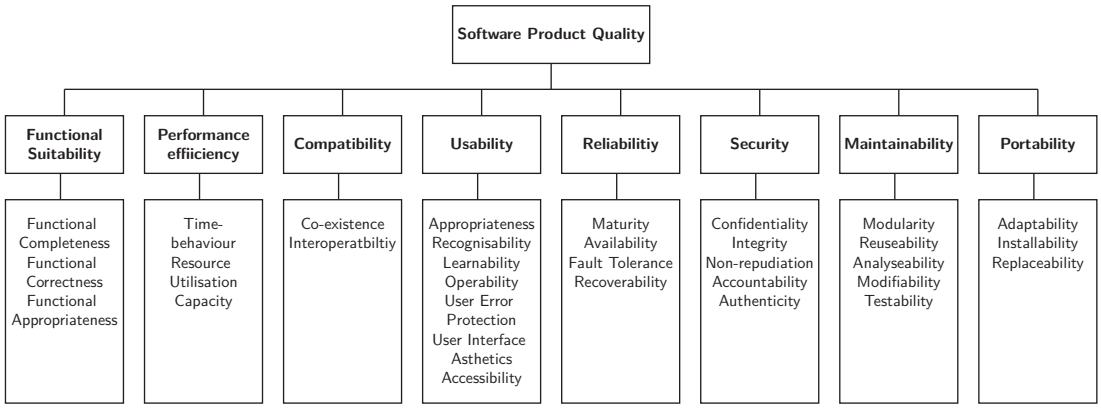


Figure 3.1: Characteristics of Software Quality (ISO 25010)

3.2.2 Non-Conformance as Quality Measurement

In all business areas, achieving high quality products comes at a certain cost. Independent from software, Crosby defines quality as *conformance to requirements* and established the price of non-conformance as the measurement of quality [21]. Furthermore, Gryna defines *conformance* and *non-conformance costs* [51]: Whereas the cost of conformance is the amount spent to achieve high quality, non-conformance costs include all expenses resulting from insufficient quality.

While the original idea of non-conformance stems from manufacturing [51], it has been transferred and applied within software engineering as well [80, 82, 99]. In particular in activity-based software quality models, conformance/non-conformance impact costs for software maintenance—low software product quality increases software maintenance costs [14, 24, 55]. For example, non-conformance costs comprise retesting after a software failure occurred in production or restructuring the code to be able to implement new change requests. In contrast, conformance costs include expenses to train developers in new design or process methodologies or testing before deployment.

In contrast to manufacturing, however, long-lived software product are continuously adapted after their initial development. In traditional manufacturing, once a product is sold, only its original functionality is maintained: A fridge, for example, will be repaired during its warranty time. However, it will not be extended to also include a freezer if it has not done so when it was sold. For software, requirements are quite different: Long-lived software is expected to be extended with new features and adapted to changing requirement as well as to changing hardware and technology.

As long-lived software is often maintained as a service, the differentiation between conformance and non-conformance costs becomes more difficult: When software reveals quality deficits after its release, the development team takes measures to redeem them, i. e. to test more, to restructure the code, or to refine the architecture. Based on Gryna's definition, these activities could be classified either as conformance costs—they are used to achieve high quality—or as non-conformance costs—they are necessary due to a lack of quality.

For the purpose of this thesis, we refine Gyrna's definition [99]:

Definition 3 (Conformance Costs) *The cost of conformance is the amount spent to prevent quality deficits of a software release before it is deployed in production.*

Conformance costs include all expenses prior to deployment that do not directly affect the code itself, for example training the developers, establishing adequate tool infrastructure, or creating coding conventions. Additionally, conformance costs comprises activities that are performed on the source code before it is released, e. g. testing before deployment.

Definition 4 (Non-Conformance Costs) *Costs of non-conformance arise after a software release was deployed in production. They arise due to two different reasons: First, they can result directly from quality deficits of the software in production. These are called*

external non-conformance costs. Second, they can stem from measures to remove existing quality deficits. These are called internal non-conformance costs.

External non-conformance costs arise from software failures in production. They comprise, for example, costs due to software downtime, failure follow up costs, or costs for loss of reputation. Internal non-conformance costs arise if a release has quality deficits that should be redeemed. For example, software that has lots of failures in production has to be tested more. Software that reveals higher-than-average costs to perform changes has to be refactored. A software architecture that does not allow adaption to new technology or underlying hardware has to be refined. Based on our classification, these activities comprise internal non-conformance costs.

3.3 Quality Assurance Techniques

With source code being the essential core of a software product, its quality significantly impacts the overall quality [8, 13, 14, 28]. To ensure high code quality over time and reduce non-conformance in the overall life cycle, there exists a large variety of quality assurance activities. Quality assurance activities can be subdivided into analytical and constructive measures [75] (see Figure 3.2): Whereas constructive measures help to *achieve* high quality (e. g. educating staff, providing appropriate tools, designing flexibly architectures), analytical measures *analyze* the existing quality of the product (e. g. testing, reviews etc.).

While constructive quality assurances create conformance costs, analytical measures can create both conformance or non-conformance costs. On the one hand, analytical measures are used to achieve a software product of high quality. Hence, they create conformance costs (testing before deployment, for example). On the other hand, if analytical measures are used to redeem existing quality defects, they address the internal non-conformance of the system (for example, re-testing after fixing a field bug). In highly agile software development, the line between using analytical measures to achieve high quality (conformance) and using them to analyze insufficient quality (non-conformance) becomes very blurry: For example,

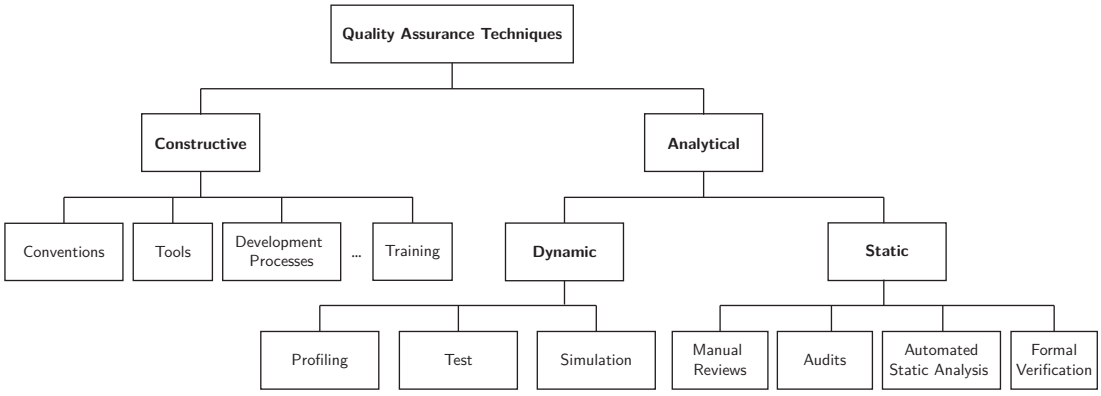


Figure 3.2: Software Quality Assurance Techniques [75]

in the SCRUM development approach, each sprint—usually comprising two or four weeks of development— targets the delivery of a product increment. Testing during a sprint can target both the quality assurance of the next product increment as well fixing existing defects created during the last sprint. For this thesis, we classify the costs for analytical measures as internal non-conformance costs if they arise for code *after* it has been released and deployed in production.

3.3.1 Analytical Quality Assurance

Based on our year-long industry experience as software quality consultants, we consider the usage of analytical quality assurance still offers big room for improvement in practice: We have been observing this based on anecdotal evidence and numerous interviews in customer projects throughout different domains, for example, insurance, energy, automotive, or avionic. Yet, our claim is certainly not generalizable to software development in practice. However, to improve quality assurance in our consultancy domain, we only focus on analytical quality assurance in this thesis. Analytical measures comprise *dynamic* and *static* analyses, see Figure 3.2. Both are applied after the code is written. Dynamic analyses require the execution of the system, e. g. various forms of testing, profiling, or simulation.[1] Static analyses comprise both manual analyses (code reviews, code audits, walk throughs) or automated analyses (code analysis or code verification). As opposed to dynamic analyses, static analyses analyze the code without running it. Hence, they can be a used before a system is tested [6]. Furthermore, they mostly do not require the complete code base but can also be used on a subset of the code.

The different analytical measures target different aspects of the eight software quality characteristics (see Figure 3.3). Nevertheless, they all aim to reduce the non-conformance costs of the system. Whereas testing, for example, is mostly used to ensure functional suitability, it can also address to some degree reliability (e. g. recovery tests), efficiency (performance tests), usability (usability tests), security (penetration tests), compatibility (compatibility tests) or portability (installation tests). However, testing is less used to address the maintainability of the system, i. e. its modifiability or its analyzability. Automated static analyses, in contrast, are mostly used to analyze the maintainability of the system [3,29,58,64,81,99] followed by the functional suitability [6,23,52,53,92,102,108]. Additionally, some automated static analyses also address aspects of security, usability, efficiency, and reliability and few address portability, or compatibility [19,32,35,58,69,81,108].

3.3.2 Dynamic vs. Automated Static Analysis

Even though static and dynamic analyses primarily focus on different quality ISO quality criteria, they yet overlap in some. However, when overlapping in one criterion, they yet address different aspects of it: For example, both static and dynamic analyses address the functional suitability. Nevertheless, they find different types of programming faults [69,100]:

[1] We generally refer to testing as the execution of the software with the goal to verify a certain property of the software. In contrast, profiling only monitors the execution of the software and provides dynamic usage data for later inspection.

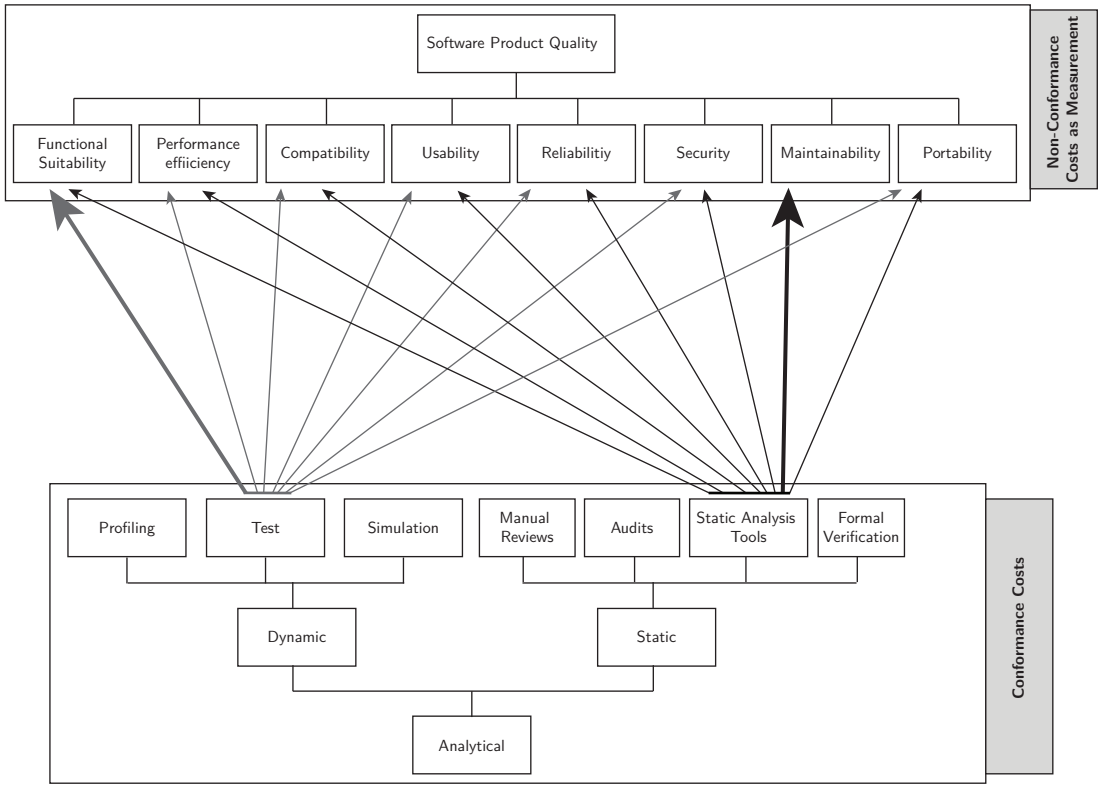


Figure 3.3: Targets of Quality Assurance Techniques

Whereas automated static analyses can find low-level programming errors, e. g. buffer overflows or null pointer dereferences [70, 99], testing can reveal deep functional or design errors [69]. Static analyses can find programming errors even on paths that are not executed during a test. Dynamic analyses, in contrast, can find defects that only occur for a certain input. In addition, static analyses can be applied even before the system is tested [6, 32]. They can reveal faults directly during development. Hence, many companies try to perform a combination of both [108,108]. However, even in combination, static and dynamic analyses often cannot assure the functional suitability completely as field bugs still occur after the system is deployed in production.

As another example, both dynamic and static analyses analyze maintainability; yet, they analyze different aspects: Dynamic analyses—profiling usage data, for example—can detect unused code, i. e. code that is unnecessary and, hence, never used during program execution. This code creates unnecessary maintenance costs [30]. As detecting unused code requires the execution of the program, static analyses are not capable of analyzing this. Contrarily, static analyses address other aspects of maintainability, for example, higher maintenance costs due to code duplication [47]. Detecting duplicated code cannot be done dynamically but requires the analysis of the code itself—hence, it can only be done statically. Similarly to not fully addressing the functional suitability, static analyses can neither quantify all aspects of software maintenance completely. Instead, they can only provide some insights. We will discuss this limitation in Section 3.5.

Compared to dynamic analyses and manual static approaches, automated static analyses have several advantages that make them a vital part of available quality assurance techniques. Dynamic analyses such as testing or security attack simulations can be quite expensive [99]. As static analyses can be applied directly even before the system is tested [6], they do not depend on the runtime environment, surrounding systems, or underlying hardware. Hence, they are easier to perform than dynamic analyses [60, 96]. Compared to fully manual static approaches such as code reviews, automated static analyses provide more tool support and are, hence, cheaper to perform [100]. Also, they can be used as aid to enhance manual review processes [73]. As many static analysis tools are open-source, companies can use them with very little conformance costs.

3.3.3 Automated Static Analysis Examples

In the following, we describe a list of exemplary static analyses to assess code quality that we will refer to in this thesis. However, due to the large number of available static analyses, an exemplary list of will never be complete and is also subjective [95]. Yet, the core idea of the approach presented in this thesis (Chapter 6) is independent from the selected static analyses. Hence, we believe that the subjective selection of static analyses does not reduce the contribution of this thesis.

We selected our exemplary analyses based on our three-year industrial experience as software quality consultants, as we perceive them to have gained acceptance in industry as well as in academia [8, 29, 66, 85, 94, 95, 103–105]. Even though we have anecdotal evidence that these analyses are a useful quality assurance, they can yet, of course, be questioned and debated. But based on the current state-of-the-art, we also have no evidence that there are better analyses available.

In the remainder of this thesis, we will refer to the following examples of static analyses. In this section, we only describe the analysis on an abstract-level. For possible specific configuration for each analysis, please refer to Chapter 4. For each analysis, we indicate which quality aspect they address based on the ISO definition and whether they reveal external or internal findings (see Terms & Definitions in Chapter 2).

Clone Detection (Redundancy Assessment) Detecting code clones, i. e. duplicated pieces of source code, is a measure of code redundancy within a system. Depending on the configuration, there are three different types of clones—exact (identical) copies of code fragments (type I clones), syntactically identical copies (type II clones) or inconsistent copies (type III clones) [57]. All three types hamper maintainability and, hence, reveal internal findings: As changes have to be propagated to all duplicated pieces, clones increase the effort to perform a change request. Hence, they result in non-conformance costs from internal failure. In fact, research has shown that every second inconsistent clone class represent a fault in the system [49]. In particular, inconsistent clones do not only hamper maintainability, but are also error-prone if the inconsistency was unintentional. Hence, they also affect the functional correctness of the system. In this case, inconsistent clones can be external findings as well. For a classification between external and internal, please refer to Chapter 6.

Long File Detection (Structure Assessment) As one structural metric, detecting overly long files reveals internal findings that hamper understandability and changeability of the code: If a file is overly long, it is harder to understand and developers need to take a lot of context information into account when trying to find a specific location for a change. Hence, overly long files are likely to increase internal non-conformance costs.

Long Method Detection (Structure Assessment) Besides the detection of long files, we use a second structural metric that detects overly long methods. Overly long methods are harder to understand, more difficult to test, and lead to coarser profiling information than short methods. Similarly to long files, overly long methods hamper maintainability and are, hence, internal findings.

Nesting Depth (Structure Assessment) The nesting depth denotes the number of open scopes for each line of code. In Java, for example, it corresponds to the number of open (i. e. unclosed) curly braces. Deep nesting is classified as an internal finding because it hampers readability and understandability of the code: If a developer has to change a deeply nested line of code, he needs to take all information from the open scopes into account to understand which conditions hold for the line to be changed. Hence, deeply nested code hinders maintainability. Nesting findings are internal findings as well.

Comment Completeness (Comment Assessment) Comments in code is a main source for code documentation. Comments should, amongst others, document the interface of the code to be used by others. Quantitatively, they should document the interface completely, i. e. every public class and public method of the interface should have a comment. Missing comments are classified as internal findings. Qualitatively, comments should provide information beyond what is obvious from the code itself; in [88], few static analyses are provided to partly assess comment quality. These analyses can reveal internal findings as well.

Bug Pattern (Functional Suitability Assessment) For several object-oriented programming languages, a variety of bug pattern detection tools exist: In the domain of Java programs, for example, Findbugs[2], for instance, performs a data flow analysis on byte code and provides rules for the developers. Some rules are helpful to detect bug patterns that likely point to a programming faults [6, 7, 97]. As some of theses programming faults can lead to null pointer exceptions that, in turn, might cause an unintended behavior, these findings contain a risk for a system failure. Hence, they are external findings. Similar tools exist for other domains as well, such as Coverity[3] for C and C++.

[2] www.findbugs.sourceforge.net
[3] www.coverity.com

The quality consultancy company CQSE has been evaluating the code quality of many of their customers in one-time assessments, called *audits*. Besides manual inspection, static analysis is the main assessment technique with the focus on maintainability and functional suitability. During various audits, bug pattern detectors revealed programming faults (confirmed by the developers). After one such programming fault was revealed in the final audit presentation, the manager of a Finnish customer stated that the costs of the audit were already redeemed; if the programming fault led to a system failure in production, the follow-up costs for debugging, retesting, and redeployment or hotfix would have been significantly higher than the costs for the code audit.

Case 3.1: Cost-Effective Bug Pattern Detection

3.4 Application of Automated Static Analysis

Even though automated static analyses are not capable of addressing the various quality aspects completely, they are still a powerful approach. Compared to other analytical measures, they have several benefits that make them a vital complement to the set of available quality assurance techniques. We demonstrate the application of static analyses with the help of several anecdotal industrial cases. Due to NDA restrictions, we have to make all cases anonymous.

3.4.1 Cost-Effective Prevention of System Failures

Many static analyses target the maintainability of the code [58]. However, they are also extremely useful to determine programming faults or security vulnerabilities in the system [69]—in particular, because other quality assurance techniques such as testing or security attack simulations can be quite expensive [60]. As many static analysis tools are open-source, companies can use them with very little conformance costs. Yet, when static analyses reveal programming faults or security vulnerabilities, the following three industrial cases (Cases 3.1, 3.2, and 3.3) show that the save of non-conformance costs often redeems the conformance costs very quickly.

3.4.2 Fundamental Assurance for Maintainability

Beyond saving non-conformance costs by revealing programming faults or security vulnerabilities, static analyses are most often used to save non-conformance costs by assuring maintainable code [3, 29, 58, 64, 81, 99]. A well-maintained code base often becomes crucial for long-lived software systems: it allows the code base to be adapted to changing requirements, evolving technologies, or new hardware.

In the following, we discuss the importance of static analyses to assure maintainability based on two industrial cases, see Case 3.4 and 3.5. The first one illustrates an insurance company

As parts of the audits as described in the previous industrial case, we use a clone analysis to assess the redundancy within a system. Redundant code created by copy&paste programming does not only create a change-overhead, but is also prone to become unintentionally inconsistent [49]. In almost every audit, we detect programming faults by inspecting inconsistent clones and confirm them with the developers. One example comprises the audit of a German mobile security provider: We detected a programming fault, because code with String constants was copied and pasted and in one instance, the String constant was forgotten to be adapted. The manager of the system was enthusiastic that a fault was revealed within this audit because—according to him—it would have taken the team much longer to detect this fault with testing techniques. Hence, he was convinced by the cost-effectiveness of the static analyses used in the audit.

Case 3.2: Programming Faults Through Copy&Paste

In regular Java audits, we often assess the exception handling of a system, i. e. its ability to react to any kind of unexpected behavior at runtime. In particular, we use a static analysis that reveals catching a general Java class *Exception* with an empty catch-block. In this case, not only null pointer exceptions are catched but also any error if the Java runtime environment stops working. While it is already critical that a generic exception handling is used instead of a specific one, it is even more critical that the exception is not handled at all in the empty catch block. During an audit of a German insurance company, we found one such finding that had a direct impact on security: The affected code fragment checks, whether a user has access rights for a specific insurance tariff. During this check, an exception can be thrown. If the exception is thrown, the empty catch block is executed. As the catch block oppresses the problem, the subsequent code will be executed as if the user had the correct access rights. Hence, in case of the exception, the security policy is violated as a user obtains access rights that he should not have. This can—in the worst case scenario—cause data losses or misuse and create extremely high non-conformance costs.

Case 3.3: Security Problems due to Exception Handling

being convinced of the benefit of using static analyses as quality assurance technique. The second one provides anecdotal evidence of the impact of not addressing static analyses and their findings. It shows that maintainability problems such as the redundancy in a system can cause an entire stop of maintenance: in the worst-case scenario, the non-conformance costs finally comprise costs of redevelopment.

The insurance company has been using static analyses for many years with the primary focus on functional suitability and maintainability. For one of their projects, they regularly devote time and budget to remove manually selected findings from the code base. Further, they have been actively using and evaluating the static analysis tool Teamscale for two years. Recently, heads of the development department and team managers discussed benefits and costs of using static analyses. When discussing whether static analysis is the right quality assurance technique to invest money on, the insurance company was convinced that removing findings continuously enables them to achieve high code quality. Whereas high code quality is not a self-fulfilling purpose, they consider it the fundamental basis to be able to perform major architectural changes. Hence, they consider static analyses to be a fundamental quality assurance technique.

Case 3.4: Static Analyses as Maintainability Assurance

As many long-lived systems cause unexpected life cycle costs, several customers of the CQSE are facing the decision whether an existing, long-lived system should be newly developed. Often, it is questionable whether these systems can still be maintained efficiently. One of the CQSE customers decided to stop the maintenance of a system because—amongst other reasons—the redundancy within the system created an unbearable overhead per change request. At the point in time of our audit, the system was three years old and had little more than 100,000 lines of code. Even within this short history, the system had a clone coverage of 37.4%. This indicates an extraordinary high amount of redundancy; as a target reference we use a threshold of 10% for all our customer systems in continuous quality control [85].
In this case, the effects of not using static analyses were one reason to stop the existing development. The non-conformance of this system finally resulted in the costs of redevelopment. It remains speculative what would have happened if a clone detection was used repeatedly during initial development. Nevertheless, we have evidence that even in large systems, it is possible to control to redundancy both during development and later maintenance [85].

Case 3.5: Stop of Maintenance due to Cloning

3.5 Limitations of Automated Static Analysis

Even though static analyses are a powerful quality assurance technique, they face certain limitations. In the previous sections, we provided anecdotal evidence about their strengths and first insights about the benefit of removing their findings. However, it is unclear how much of the overall non-conformance of software can or cannot be quantified by static analyses. Static analyses are restricted per definition to the code base. It is barely examined how many problems in the code base can be detected by static analyses and, additionally,

how many problems are in fact unrelated to the code. In this section, we outline the state-of-the-art of current research.

In an empirical study, Yamashita and Moonen addressed this problem and examined to what extent maintenance problems can (or cannot) be predicted by static analysis tools (code smell detectors) [103, 105]. Prior to the study, four different companies were hired to independently develop their version of the case study system, all using the same requirement specification. These four functionally equivalent systems were then used for the maintenance study in which six professional developers had to perform a big maintenance task of replacing the systems' underlying platform. The study kept track of any kind of problem that occurred during maintenance.

The study showed, that about half the recorded maintenance problems were not related to the source code itself. Instead, they related to the "technical infrastructure, developer's coding habits, dependence on external services, incompatibilities in the runtime environment, and defects initially present in the system" (prior to the maintenance phase). Within the code-related maintenance issues, 58% were associated with findings from static analyses. Hence, based on these results, static analyses can partly address the maintainability aspect of software quality, but not completely. They only revealed about a third of the overall maintenance problems.

However, this study is limited to a selected subset of static analyses. Code duplication, for example, is stated to be beyond the scope of the study. Hence, even though the study provides very valuable insights about impact and limitations of static analyses, it cannot make a general claim. Yet, it is very difficult to conduct an all-encompassing study to quantify the limitations of static analyses. The longitudinal study setup of Yamashita and Moonen has been unprecedented— to our best knowledge— and cost about 50,000 Euro. It was the biggest longitudinal study to study the impact of code quality on software maintenance in four functional equivalent systems. Yet, its results provide only first insights. It is out of the scope of this thesis to replicate such a study or to further quantify impact and limitations of static analyses with an scientifically sound approach.

3.6 Summary

In this chapter, we introduced the underlying maintenance model of this thesis, the ISO 25010 standard for software quality, and the concept of non-conformance costs as a measurement of quality. We outlined the variety of quality assurance techniques and, in particular, examined the impact of using static analysis in industry. With several industrial cases, we showed that static analyses are very cost-effective when they can reveal faults of the system. Further, they are a powerful technique to assure maintainability of the code.

Besides their inevitable impact, static analyses also face certain limitations as quality assurance technique. This chapter illustrated them: It is unclear how much non-conformance costs can be saved using static analysis. Further, it is also barely examined how many aspects of software quality cannot be addressed by static analyses at all.

"Quality means doing it right when no one is looking"

Henry Ford

4 A Quantitative Study of Static Analysis Findings in Industry

As a first step in this thesis, we analyze the state-of-the-practice of using static analysis in industry. In this chapter, we provide an empirical study to analyze the number and the nature of findings revealed by commonly used static analyses in practice. The study serves as empirical starting point for our research, from which we derive the steps following in Chapter 5 and 6 to obtain a cost-effective approach for using static analyses as quality assurance technique.

Section 4.1 outlines the study design while Section 4.2 presents the study objects. We show the results of the study in Section 4.3 and discuss them in Section 4.4. Section 4.5 summarizes this chapter.

4.1 Study Design

The case study aims to determine how many findings commonly used static analyses reveal in brown-field software systems. In the study design, we differentiate between analyses that affect maintainability and analyses affecting program correctness. This differentiation is based on the observation in Chapter 3 that static analyses address different quality aspects of a system. In particular to address maintainability and correctness, static analyses are most frequently used. Hence, these analyses are suitable for a state-of-the-practice investigation. As denoted in Chapter 2, we refer to findings of maintainability analyses as *internal* findings, to those from correctness analyses as *external* findings.

The study uses several commonly used static analyses and applies them on a large number of industry and open source systems written in three different object-oriented languages Java, CPP, and CSharp. The study serves as a quantification to reveal how many findings each analysis detects.

4.1.1 Analyses Regarding Maintainability

As common analyses with respect to maintainability, we use a code clone detector to measure redundancy, structural metrics to analyze the organization of the code in files and methods, and a comment completeness analysis that assesses the quantitative code documentation. (see Section 3.3.2).

We selected these maintainability analyses for several reasons. First, because they are commonly accepted in industry and academia [8, 66, 85, 94, 95, 103–105]. Second, because

they can be conducted language-independently. Hence we can provide comparable results for the study which comprises three different object-oriented languages. Third, the findings produced by these analyses require manual effort to be removed appropriately: In contrast to, e. g. formatting violations that can be easily removed with a code auto formatter, code clones or long files cannot be refactored automatically such that the refactoring is semantically meaningful (in the current state-of-the-art). Also code comments cannot be generated automatically with the same quality and information gain as human written comments. Hence, to improve maintainability, comments have to be inserted manually.

On the other hand, we limit the maintainability study to these analyses as they require the least manual interpretation of the results—they can be used fully automated to assess their quality aspect. Other analyses, e. g. to evaluate the exception handling, the naming of the identifiers, or the quality of the comments, require a lot more manual inspection of the code to draw a meaningful comparison and to eliminate the number of false positives. Hence, they are not suitable to be used in a fully automatic analysis.

For each analysis, we explain its configuration and the derived metric in the following. For background information about the quality aspects addressed by a specific analysis, please refer to Section 3.3.2. The analyses were implemented within the quality analysis tool Teamscale [40].

Clone Detection For the study, we calculate the number of clone instances per system. The number of clone instances illustrates the amount of (clone) findings a developer is confronted with. We use the existing code clone detection of Teamscale [42] and configured it to detect type-II clones[1] with a minimal clone length of ten statements.

Long File Detection We count how many overly long files exist in each system. We detect long files by counting source lines of code (SLOC, lines of code excluding comments and empty lines) and use a threshold of 300 SLOC to define a class as overly long. The threshold is an experience-based threshold from CQSE that has been used for many years and in many development teams across Java and .NET domains.

Long Method Detection Besides the detection of long files, we use a second structural metric: we show how many long methods exist – further findings among which a developer needs to prioritize. We define a method to be *long* if it has more than 30 source lines of code. The threshold results from the experience of the CQSE GmbH and recommendations from [66]: it is based on the guideline that a method should be short enough to fit entirely onto a screen.

Nesting Depth As a third structural metric, we measure the nesting depth of the code. We define a nesting finding as a line of code that is located within at least three open scopes. We start counting open scopes at method level, i. e. the scope opened by the class definition is ignored. This threshold is also experience-based.

[1] For definition of type-I, II, and III clones, refer to [57].

Abbreviation	Findbugs Rule
[DefNPE]	Definite Null Pointer Exception
[PossibleNPE]	Possible Null Pointer Exception
[NullcheckPrevDeref]	Nullcheck of value previously dereferenced
[StringComp]	String Comparison with `!=` or `==`
[MisusedEqual]	Calling `equals` on different types or while assuming type of its argument
[UncheckedCast]	Unchecked or Unconfirmed cast
[KnownNull]	Load of known null value
[NullPass]	Method call passes null for non-null parameter
[WriteToStatic]	Write to static field from instance method

Table 4.1: Selected, Error-Prone Findbugs Rules

Comment Completeness We measure the comment completeness for interfaces: For Java and CSharp, we count how many of the public classes/types within a system reveal a comment preceding them and count it relative to the total number of public types. For C/CPP we measure how many of the type or struct declarations are preceded by a comment (relative to the total number of declarations). This is a quantitative measure only. However, compared to a qualitative assessment that can be done at most semi-automatically, the quantitative measure provides an automatic and comparable metric for the study.

4.1.2 Analyses Regarding Correctness

With respect to the functional suitability, we use the Java-specific open-source tool Findbugs (see Section 3.3.2). Findbugs provides a bug pattern detection on byte code with a large variety of different rules. Some of them have high correlation to actual faults in the program [97]. Based on [97], we selected a small subset of Findbugs rules whose correlation to actual programming faults is the highest. We enriched the selection process with empirical evidence we gained through numerous code audits, in which we asked developers about the relevance of Findbugs results on their own code. Hence, we selected those rules that have the highest correlation to faults in [97] and revealed the highest relevance based on developers' opinions in practice. Table 4.1 shows the selected rules.

4.2 Study Objects

As study objects, we selected a broad range of industry and open source systems. However, we did not have both source code and byte code available for all case study systems. Hence, as the maintainability analyses require source code and the correctness analyses require byte code, we use different case study objects for either one.

From all systems, we excluded generated code as it is usually only generated once and not manually maintained after. Hence, the quality of generated code has to be measured

differently than the quality of application code. We further also exclude test code from the study. The amount of test code varies greatly in industry systems and, often, influences the number of findings significantly [94]. Test code often contains a lot more clone findings than application code. Hence, to obtain comparable study results, we excluded test code.

4.2.1 Maintainability Study Objects

Table 4.2 shows the study objects for which we had the source code available. Amongst them, 14 are written in Java, 4 are written in C/C++ and 20 are written in CSharp. Due to non-disclosure agreements, we use anonymous names for the industry systems. In the table, the systems are sorted by system size, which ranges from about 20 KSLOC to little over 2.5 MSLOC. We selected open source and industry system that have grown over the years and span a variety of different domains—from insurance over avionics, energy, and general manufacturing to software product development. Hence, we believe that they are a representative sample of brown-field software development.

4.2.2 Correctness Study Objects

Table 4.3 shows the systems in the correctness study, for which we had the byte code available. Note that System D and E appear in both parts of the study because for them, we had both source and byte code. Unfortunately, we were able to include only seven systems in the study. Due to the low number of case study systems and the Java dependency, the correctness study is much less representative than the maintainability study.

Table 4.3 also indicates how the teams were using Findbugs before their code was analyzed within this study. Only one team (System F') claimed that Findbugs was integrated into their nightly build. Hence, developers had the warnings available. However, Findbugs rules were not strictly enforced, i. e. adding new Findbugs findings could not cause the build to break. One other team (System H') indicated that they were running Findbugs occasionally, but not as part of their continuous integration system. To our best knowledge, all other teams were not using Findbugs.

4.3 Results

In the following, we present the quantitative number of findings separately for maintainability and correctness analyses.

4.3.1 Findings From Maintainability Analyses

Table 4.4 shows for each language the aggregated results over all five maintainability analyses: On the one hand, we calculated the absolute internal findings count as the sum of the five analyses' findings; we show the average and the median over all systems as well as the minimum and maximum. On the other hand, we normalized the absolute findings

	Name	Domain	Development	Size [SLOC]
Java	System A	Energy	Industry	22,911
	System B	Energy	Industry	29,577
	System C	Insurance	Industry	34,164
	System D	Avionics	Industry	63,081
	JabRef	Software Product Development	Open Source	68,549
	Subclipse	Software Product Development	Open Source	84,467
	ConQAT	Software Product Development	Open Source	92,635
	System E	Business Information	Industry	100,219
	JEdit	Software Product Development	Open Source	111,035
	System F	General Manufacturing	Industry	147,932
	Tomcat	Software Product Development	Open Source	190,202
	ArgoUML	Software Product Development	Open Source	195,670
	System G	Insurance/Reinsurance	Industry	240,512
	System H	Mobile Telecommunication Services	Industry	263,322
CPP	System I	Software Product Development	Industry	94,687
	Tortoise	Software Product Development	Open Source	158,715
	System J	General Manufacturing	Industry	550,243
	Firefox	Software Product Development	Open Source	2,634,278
CS	System K	Insurance/Reinsurance	Industry	20,851
	System L	Insurance/Reinsurance	Industry	23,632
	System M	Business Information	Industry	30,495
	System N	Insurance/Reinsurance	Industry	31,969
	System O	Insurance/Reinsurance	Industry	42,771
	System P	Insurance/Reinsurance	Industry	51,339
	System Q	Insurance/Reinsurance	Industry	57,118
	System R	Insurance/Reinsurance	Industry	67,522
	System S	Insurance/Reinsurance	Industry	85,775
	System T	Insurance/Reinsurance	Industry	89,287
	System U	Insurance/Reinsurance	Industry	96,510
	System V	Insurance/Reinsurance	Industry	102,072
	System W	Insurance/Reinsurance	Industry	130,253
	System X	Insurance/Reinsurance	Industry	182,399
	System Y	Insurance/Reinsurance	Industry	189,540
	System Z	Insurance/Reinsurance	Industry	192,170
	System A'	Insurance/Reinsurance	Industry	281,968
	System B'	Insurance/Reinsurance	Industry	329,667
	System C'	Insurance/Reinsurance	Industry	420,558
	System D'	Insurance/Reinsurance	Industry	622,407

Table 4.2: Maintainability Study Objects

	Name	Domain	Development	Findbugs Usage	Size [LOC]
Java	System D	Avionics	Industry	No	101,615
	System E	Business Information	Industry	No	153,735
	System E'	Business Information	Industry	No	84,248
	System F'	Business Information	Industry	Regularly	396,225
	System G'	Business Information	Industry	No	459,983
	System H'	Business Information	Industry	Unregularly	573,352
	System I'	Insurance	Industry	No	953,248

Table 4.3: Correctness Study Objects

count with the system size and denote the number of findings per thousand SLOC. Again, we show average, median, minimum and maximum.

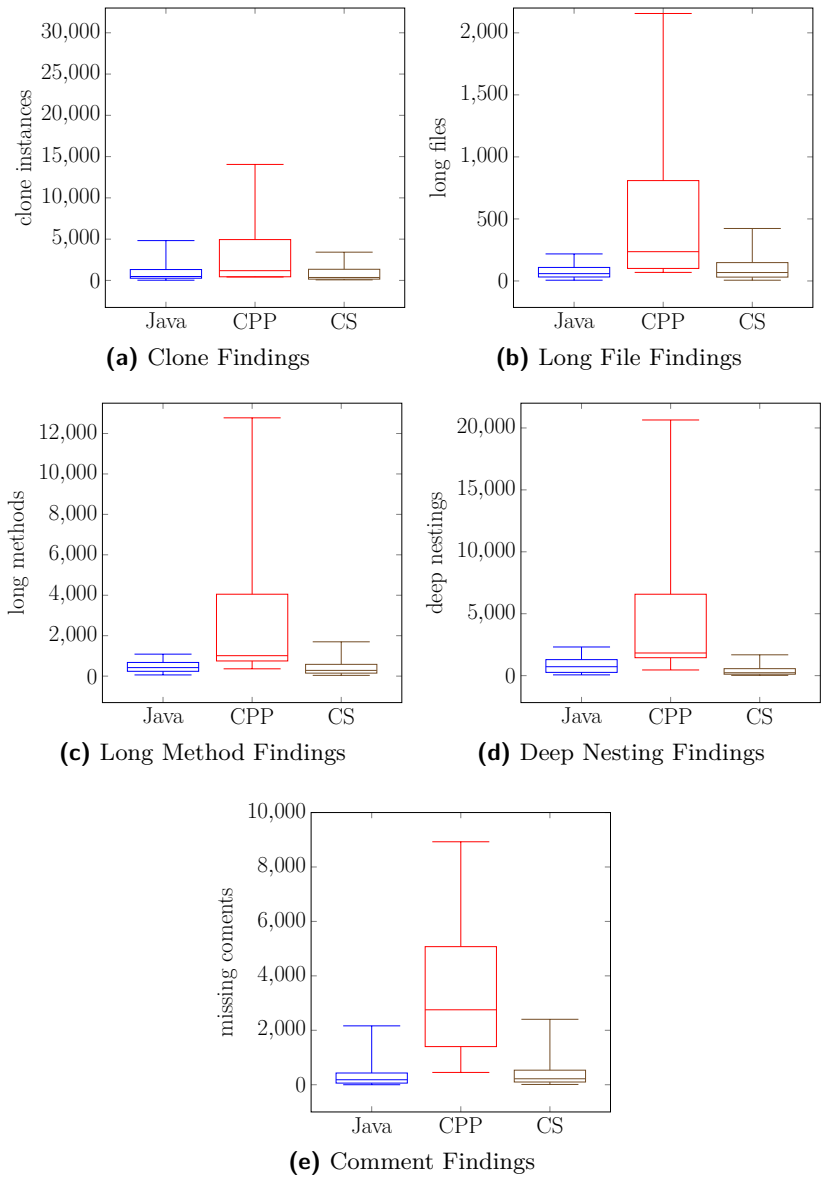
The table shows that we have on average 2,355 internal findings in CSharp, 3,103 in Java, and above 18,000 in CPP. The median is lower with between 1,207 internal findings in CSharp and 7,010 in CPP. Normalizing the absolute findings count, we encounter between 17 findings per KSLOC (CS) and 24 (Java) on average.

Figure 4.1 shows the number of internal findings per analysis and per language. We aggregated the results of the different systems with box-and-whisker plot: the bottom and top of the box represent the first and third quartiles, the band inside the box the median. The ends of the whiskers denote the minimum and maximum of the data.

The redundancy measure produces the most findings, with a median of about 350 clone instances in CS, 500 in Java and 1200 in CPP. The maximums even reveal more than 14,000 clone instances in one CPP system. Among the structural metrics, the long file analysis causes the least number of findings, with a median of about 59 in Java, 68 in CS, and 236 in CPP. The number of findings increase for the long method analysis (medians 238 for CS, 421 for Java, and 1011 for CPP) and even further for the nesting findings. Finally, there are 183 missing comments in Java (median), 219 in CS, and 2753 in CPP.

	absolute [all findings]				relative [per KSLOC]			
	∅	median	min	max	∅	median	min	max
Java	3,103	2,109	177	8,123	24.3	23.9	1.9	61.5
CPP	18,582	7,010	1,756	58,553	22.1	20.3	16.5	31.9
CS	2,355	1,207	158	9,107	17.1	14.8	7.5	31.7

Table 4.4: All Maintainability Findings



(a) Clone Findings

(b) Long File Findings

(c) Long Method Findings

(d) Deep Nesting Findings

(e) Comment Findings

Figure 4.1: Findings Separated By Analysis

4.3.2 Findings From Correctness Analyses

Table 4.5 shows the results and reveals a much smaller number of external than internal findings. We found only two definite null pointer exceptions (in all systems) and not more than 20 possible null pointer exceptions per system. Overall, a system revealed between six and 115 critical external findings, depending on the system size.

Based on our results, the system whose team was using Findbugs regularly (System F') still reveals 14 critical findings. Even though developers had the Findbugs results available, they did not remove all critical warnings. We assume that this is due to the team not strictly

System	D	E	E'	F'	G'	H'	I'
[DefNPE]	–	1	–	–	–	1	–
[PossibleNPE]	15	15	–	1	–	20	2
[NullcheckPrevDeref]	6	–	–	–	–	14	–
[StringComp]	–	6	–	1	–	9	–
[MisusedEqual]	10	–	–	2	2	1	4
[UncheckedCast]	–	–	–	5	–	–	–
[KnownNull]	–	–	1	–	2	9	–
[NullPass]	7	3	3	–	7	1	–
[WriteToStatic]	–	–	1	5	3	60	–
Sum	38	25	5	14	14	115	6

Table 4.5: Number of Error-Prone Findings

enforcing Findbugs warnings—the build would not fail even if developers created critical findings with their commit. The team developing System H' was claiming to use Findbugs as well, but not regularly. System H' reveals the most findings. However, it was also the second biggest system in the correctness study. In this case, the number of findings might related stronger to size than to the unregular usage of Findbugs. Overall, the two teams that had used Findbugs before our analysis were not performing significantly better than other teams. However, again, the small size of our study permits to generalize this result.

4.4 Discussion

The results show that the number of internal findings revealed by only five commonly used maintainability analyses is large enough to require a prioritization mechanism: Even the minimum of the absolute findings count in the case study (158 findings on a CS system) is difficult to prioritize manually for removal. The study further shows that the maximum of the absolute findings count (58,553 findings on a CPP system) is beyond of what a single developer or quality engineer can cope with.

The study's results show further that the number of internal findings is considerably higher than the number of external findings. We assume the small amount of external findings is due to their criticality motivating developers: if a finding can cause a costly system failure, developers are often willing to remove it immediately.

The small number of study systems in our correctness study threatens the validity of our result. However, the result of existing work matches our finding that external findings occur much less frequently: Ayewah et al. used Findbugs on all 89 publicly available builds of JDK (builds b12 through b105) and reveal a similar order of magnitude in terms of quantity [7]: Overall, they found, for example, 48 possible null pointer dereferences and 54 null checks of values previously dereferenced.

In addition, our results from the correctness study match our gut feelings from several years of experience as software quality consultants as well: Beyond the study, we have used similar correctness analyses earlier during our consultancy on more than twenty other industrial systems. The study results match the results obtained earlier.

4.5 Summary

In this chapter, we analyzed how many findings commonly used automated static analysis reveal. We differentiated between analyses addressing maintainability and correctness. We conducted an empirical study on 38 industrial and open-source projects from three different domains to investigate results from maintainability analyses. To examine findings from correctness analyses, we had seven study systems available. The results show that the number of external findings is quite small for our study systems. However, already a few maintainability analyses result in thousands of internal findings. Hence, as developers likely cannot remove all of them at once and quality assurance budget is limited, they require a sophisticated prioritization approach.

"It is always cheaper to do the job right the first time."

Philip Crosby

5 A Cost Model for Automated Static Analysis

In the previous chapter, we showed that already few maintainability analyses result in thousands of internal findings. While the number of external findings is much smaller, the number of internal findings requires a prioritization. To effectively prioritize findings for removal, an estimation of the cost-benefit ratio of their removal is required.

For external findings, the benefit of removal is quite obvious: A potential system failure is eliminated. For internal findings, the benefit—which is not immediate, but long-term—is much harder to quantify. Even though we have evidence that using static analyses can be cost-effective to reveal faults (Section 3.4.1) and is beneficial to assure maintainability (Section 3.4.2), the overall reduction of non-conformance costs can generally not be quantified yet. Additionally, little is known about how much it costs to remove a specific finding (and this might be very finding-specific)

In this chapter, we provide a conceptual cost model that formalizes a return-on-invest calculation (Section 5.1). Based on our model of software maintenance, this model describes costs and benefits of removing findings. As the model is parameterized with several variables, we conduct an empirical study to gather industrial evidence for some of the variables (Section 5.2). This data allows us to provide an exemplary instantiation of the model and calculate costs and benefits of removing a *large class* (Section 5.3). We discuss the implications of the instantiation in Section 5.4. In Section 5.5, we examine how well the model can be applied to predict the cost-benefit ratio. As the model and its instantiation are based on several assumptions, we discuss the resulting threats to validity in Section 5.6. Section 5.7 summarizes this chapter.

5.1 Conceptual Model

The model serves the purpose to describe when (or if) removing a finding from the code base pays off. Therefore, it models costs and benefits of finding removal. An instantiation of the model ought to help convince project management of cost-efficiency of using static analyses as quality assurance technique. The model is based on an activity-based model of software maintenance (Definition 1, Chapter 3), and combines it with the ISO definition of software product quality (Definition 2, Chapter 3).

5.1.1 Core Condition

The fundamental principle of our cost model is that a finding is worth to be removed if the benefits outweigh the removal costs. However, both removal costs and benefit depend on the

finding's category and the finding itself: The benefit of removing code clones, for example, might differ from the benefit of removing coding guidelines violations. Even among all code clones, some may hamper maintainability more than others, revealing a bigger benefit of removal. Additionally, in terms of removal costs, some code clones can be removed significantly faster than others [86]. The same applies for other findings and findings categories. Hence, all variables in our model take the specific finding f as a parameter. (As a finding is always uniquely associated with its finding category, the latter is not modeled explicitly.)

The removal costs for a finding depend not also on the finding itself, but also, e. g. on the quality of surrounding source code or the testing environment: In cleaner code, a single finding might be easier to remove and, similarly, with a better test suite, the removal of a finding might be tested faster. For reasons of simplicity, we model this by parameterizing the removal costs with a time parameter t.

Whereas the removal costs occur at time t when the finding is removed, the benefit is distributed over a time interval after the removal [4]. For external findings, costs, e. g. through loss of reputation, can be caused days or months after a system failure occurred. For internal findings, maintenance can be eased not only for the next modification, but for all subsequent ones. For this time interval, we use the notation Δt that describes an interval $]t, t_1]$.

In addition, the benefit for removing an internal findings also depends on the included *scope* of the consideration: For example, when a long method is split up into multiple shorter methods, this can be beneficial for subsequent maintenance activities affecting the method itself—the method might be easier to read. But it can also be beneficial for subsequent activities affecting the surrounding class–the class might be easier to be over-viewed. Consequently, we take the considered scope as a parameter into account when modeling benefit. The scope depends on the finding's category. For reasons of simplicity and as above, we formalize it with a function $\text{scope}(f)$ that depends on the finding, because the finding is uniquely associated with its category.

Consequently, a finding is worth to be removed, if there is a feasible, i. e. sufficiently small time interval, after which cost savings are greater than the removal costs for the finding. In our model, a cost function returns a positive value if costs are *created* and a negative value if costs can be *saved*. Hence, we obtain the following core condition or our model for when a finding is worth to be removed:

$$\exists t_1, \Delta t =]t, t_1] : \text{removal\_cost}(f, t) < -\text{cost\_savings}(f, \Delta t, \text{scope}(f)) \tag{5.1}$$

If such an interval Δt exists depends on the expected duration of a software's life cycle. Software that is expected to be replaced in the near future is likely not worth the effort to remove findings from its code base. In contrast, software with an expected maintenance of several decades might be well worth the quality assurance. The final decision whether it is worth to remove a finding is very system-specific.

In our model, the removal costs as well as the benefits are defined for an *individual* finding. Hence, it does not take correlations or side-effects between multiple findings into account. We discuss this threat to validity in Section 5.6.

5.1.2 Cost Model

In the following, we model the removal costs of a finding. In our model, we use time as unit of cost–removing a finding costs time of the developer which then can be converted into money. Based on our definition of maintenance (Definition 1, Section 3.1), the finding removal comprises three activities: reading the corresponding parts of the code to understand the change, performing the change by removing the finding, and testing the change. All three activities come at a certain cost. The removal costs depend on the current code base and the current testing environment. Hence, their cost functions take the parameter t as input as well as the finding f itself (see Table 5.1):

$$\begin{aligned} \text{removal\_cost}(f,t) = \ & \text{c\_reading}(f,t) + \text{c\_modification}(f,t) + \\ & \text{c\_testing}(f,t) \end{aligned} \tag{5.2}$$

5.1.3 Benefit Model

In the following, we model the benefit of removing a finding from the code base. We define a boolean function is_external which returns *true* if a finding is external and *false* if it is internal. For both cases, the benefit is modeled differently:

$$\text{cost\_savings}(f,\Delta t) = \begin{cases} \text{cost\_savings\_e}(f,\Delta t) & \text{if is\_external}(f) = \text{true} \\ \text{cost\_savings\_i}(f,\Delta t, \text{scope}(f)) & \text{if is\_external}(f) = \text{false} \end{cases} \tag{5.3}$$

The benefit of removing an external finding, *cost_savings_e* (for example for a correctness finding), is quite obvious; the developer eliminates a potentially costly system failure which can create additionally failure-follow-up costs, costs for debugging, hotfixes, retesting or reputation loss. Whereas failure follow-up costs and costs of reputation loss are usually directly given as a money value in some currency, costs for debugging and retesting usually first require development time, which, in turn, can be converted to money.

For external findings, we rely on repeatedly experienced, anecdotal evidence from our customers in various quality control projects that system failures in production are very costly and, hence, the costs of static analyses quickly redeem them (see Section 3.4.1). Hence, we abstain from modeling *cost_savings_e* in more detail. Instead, we assume for any external finding f_1 and internal finding f_2:

$$\text{cost\_savings\_e}(f_1,\Delta t) \gg \text{cost\_savings\_i}(f_2,\Delta t, \text{scope}(f_2)) \tag{5.4}$$

and

$$\text{cost\_savings\_e}(f_1, \Delta t) \gg \text{removal\_costs}(f_1, t) \tag{5.5}$$

For internal findings, in contrast, the benefit is much harder to quantify. While studies exist that investigate the negative impact of findings on maintainability in general [3, 8, 23, 26, 29, 48, 49, 52, 53, 66, 94, 95, 103–105], the benefit of removing a single finding is yet very hard to predict. We model the benefit for removing an internal finding in terms of time—and, hence, cost—savings for the developer in the time interval after the finding removal. However, removing a maintainability finding is beneficial only when the affected code region is maintained again. We assume that removing a finding alleviates each subsequent maintenance activity as cleaner code can be read and modified faster [66] and tested easier [93, 94].

To quantify how much time a developer saves by reading, modifying, or testing cleaner code, we have to take the *scope* into account. The definition of the scope in our model influences how much code is read or modified: A file, for example, might be read and modified much more often than a single method. In our model, we consider both method and file level as possible scopes: $\text{scope}(f) \in \{\text{FILE}, \text{METHOD}\}$. Hence, we obtain the following model of benefit with scope and time interval as parameter:

$$\begin{aligned} \text{cost\_savings\_i}(f, \Delta t, \text{scope}(f)) = \; & \text{s\_reading}(f, \Delta t, \text{scope}(f)) + \\ & \text{s\_modification}(f, \Delta t, \text{scope}(f)) + \\ & \text{s\_testing}(f, \Delta t, \text{scope}(f)) \end{aligned} \tag{5.6}$$

The savings in code reading of a scope in a time period can be approximated with the average number of reads per interval multiplied with the average save per read. Using average as approximation of courses introduces fuzziness to our model. We discuss this threat to validity in Section 5.6. In this approximation, the average save per read depends on the finding and the considered scope. The average number of reads, in contrast, are independent from the finding as they rather depend on the performed change requests. Yet, they do depend on the considered scope as the number of reads may vary when it is measured on file or method level, for example. As the same holds for code modification, we obtain the following approximation:

$$\begin{aligned} \text{cost\_savings\_i}(f, \Delta t, \text{scope}(f)) = \; & \text{avg\_s\_read}(f, \text{scope}(f)) \cdot \text{avg\_n\_read}(\Delta t, \text{scope}(f)) + \\ & \text{avg\_s\_mod}(f, \text{scope}(f)) \cdot \text{avg\_n\_mod}(\Delta t, \text{scope}(f)) + \\ & \text{s\_testing}(f, \Delta t, \text{scope}(f)) \end{aligned} \tag{5.7}$$

In contrast to reading and modification activities, savings for testing activities are more difficult to model. First, while there is, for example, only one concept of code reading,

code testing comprises the different concepts of unit testing, system testing, or integration testing. Second, the benefit of finding removal (or code refactoring) for the testing activity of subsequent modification has multiple dimensions: the execution of existing tests might be *faster* (less time for the tester), the addition of new tests might be *easier* (less time for the developer), the tests might reveal more programming faults (less failure-follow-up costs). For example, the reduction of redundant implementation of the same functionality might decrease the number of necessary manual system tests. As the execution of manual system tests is usually quite expensive, less executions result in less time spent by the testers. As another example, removing long methods with multiple functionalities by extracting multiple short methods with single functionality can ease the addition of new unit tests because the fewer functionality a method has, the simpler the unit test.

Athanasiou et al. tried to model test code quality and evaluated the usefulness of their model based on expected benefits in code testing with respect to issue handling performance [5]. The authors found positive correlation between test code quality and throughput and productivity, but not between test code quality and defect resolution speed. The study shows that modeling the benefit of cleaner code for testing activities is quite complex. As our work focuses on static analysis and not on testing as quality assurance technique, we abstain from modeling the benefit for testing but leave it for future work (Chapter 8).

Aside from the positive impact on code reading, modification, and testing, removing findings is beneficial in other dimensions as well, e. g. increased quality awareness or higher developer motivation. These dimensions are currently not included in the model. We discuss this threat to validity in Section 5.6.

5.2 Empirical Studies

Based on the conceptual model, we strive to obtain empirical data that allow us to make realistic assumptions about the removal costs and cost savings (Equation 5.2 and 5.7). These assumptions will help us to instantiate the model in Section 5.3. Hence, we conducted a case study to enrich our conceptual model with evidence from software engineering in practice.

The case study serves to provide a very first reference point about the dimensions of removal costs and cost savings. While the results are certainly not generalizable and cover only individual aspects of the model, they do help to formulate realistic assumptions about the model. However, obtaining a suitable industrial case study object is quite a difficult task. It requires a development context, in which

- the source code is continuously analyzed with static analysis,
- clean code is a quality goal of the team,
- developers have enough resources to remove findings regularly,
- developers keep track of the costs of finding removal,
- the system, its history, or its team can provide some measure of benefit.

Variable	Explanation
c_reading(f, t)	Costs for reading code with a finding at a time-dependent version of the code
c_modification(f, t)	Costs to remove a finding at a time-dependent version of the code
c_testing(f, t)	Costs for code testing after removing a finding at a time-dependent test set-up
s_reading$(f, \Delta t, \text{scope}(f))$	Savings for reading a scope within a time period if a finding is not present
s_modification$(f, \Delta t, \text{scope}(f))$	Savings for modifying a scope in a time period if a finding is not present
s_testing$(f, \Delta t, \text{scope}(f))$	Savings for testing a scope within a time period if a finding is not present
avg_s_read$(f, \text{scope}(f))$	Average savings per read in a given scope if a finding is not present
avg_s_mod$(f, \text{scope}(f))$	Average savings per modification in a given scope if a finding is not present
avg_n_read$(\Delta t, \text{scope}(f))$	Average number of reads of a scope in a time period
avg_n_mod$(\Delta t, \text{scope}(f))$	Average number of modifications of a scope in a time period

Table 5.1: Variable Definition for Cost Model

Due to these requirements, we only had one case study object available. The single study object limits the results' generalizability significantly. Existing work, however, has shown that access to only one industrial system is valuable as much of state-of-the-art research has no access to industrial data.

5.2.1 Study Design

The case study is based on a long-running project together with one of our customers, a German insurance company. For this project, we monitored an established quality improvement process over several years: Since March 2013, the developers have performed a structured quality-control process with our support, similar to the one described in [85]. In monthly meetings, we as external experts have discussed the latest quality improvements and new findings together with the development team.

Taking into account the developers' opinions on the new findings, we selected a given number of findings based on the estimated cost and benefit of removal: The finding selection is guided by the finding's introduction date, probability of being a programming fault, ease of removal, and the change frequency of the code that contains the finding. For each selected finding, we created a "task" (Scrum story) in the issue tracker that addresses the removal of the given finding. The tasks have been scheduled together with other stories in the sprint

planning and the findings were removed during the sprint. Each developer recorded the time needed for solving a given task in the issue tracker.

The subject system is written in Java and currently comprises roughly 260 kLOC. It has a history of more than seven years and is developed using Scrum. To analyze its source-code quality, we use the static code-analysis tool Teamscale [40]. To assess maintainability, Teamscale analyzes cloning, code structuring, commenting, and naming. To assess the functional suitability, Teamscale utilizes selected FindBugs patterns. Table 5.2 comprises all findings categories. Developers of the team chose these categories as they considered them to be relevant to improve functional suitability and maintainability.

5.2.2 Data Collection

On the cost side, we extracted the time developers recorded for each finding removal from the issue tracker. This data provides a first reference point for *removal_cost*(f, t) in Equation 5.2. However, it serves as reference point only for the overall sum in Equation 5.2, not for its individual summands. The available data from the issue tracker did not allow a more fine-grained evaluation. We will discuss this threat to validity in Section 5.6.

As modeled in Equation 5.2, we measure the costs in terms of the time needed to determine an appropriate solution to remove the finding, perform the removal, and test the changes. From the issue tracker, we were able to extract the removal times for each finding. As each ticket contains the findings category, we can exploit the data per category.

On the benefit side, we focus only on internal findings (Equation 5.7) as explained above. However, based on the available historical data from the version control system, we can provide reference data only in terms of modification data, i. e. $avg\_n\_mod(\Delta t, \text{scope}(f))$. Approximating the other variables in Equation 5.7 was not feasible based on this case study set up. We discuss this in Section 5.6.

We measure how often the surrounding scope—a method or a file—was changed after one of its findings had been removed. In particular, we count the modifications per month within the period in which the surrounding scope existed in the system. For most findings, we use its file as surrounding scope. Findings of the category “Large Class” obviously affect the entire file. As clones may comprise multiple methods and also the attributes of the class, we use the surrounding file as well. The same holds for task tags and unused code because they can be located either inside or outside a method. Only for the categories “Long Method” and “Deep Nesting”, we can map all findings to a surrounding method as scope.

To obtain accurate results, it is important to track the method or file during all refactoring activities performed in the history like moves and renames. For files, we use the tracking algorithm as presented in [89]; for methods, as in [84].

To evaluate the modification frequency, we extended the set of removed findings from the set in Table 5.2 to all findings that were ever removed in the history of the system. However, we excluded findings that were removed due to their surrounding code entity being deleted for two reasons. First, these findings were not deliberately removed. Instead, their removal is only a side effect of the code deletion. Second, we cannot measure the benefit of the code

removal by subsequent changes as the removed code cannot be changed any more. Even though the code deletion might have benefits (e. g. in terms of reducing the overall system size), we ignore it within the scope of this case study. Please note that we exclude removed findings only if the surrounding entity was simultaneously deleted, not if it was deleted at least one revision after the finding removal.

5.2.3 Results

In the following, we present the results first for the removal costs and then for the average number of modifications.

Removal Costs Table 5.2 shows per category, how many findings were removed and how long it took the developers on average and as median. Over the course of two years, the team removed 215 findings. On averages it takes them between ten minutes (to remove commented out code), 30-74 minutes (to remove large classes, deep nestings, or long methods) up to 118 minutes (to removed redundancy, i. e. duplicated code). Based on the information from the developers, removing redundant code is particularly troublesome because it can cause unforeseen side-effects that make the removal more complicated than at first glance. The average over all categories is about an hour. The median resides at 30 minutes, indicating that some findings take significantly longer to be removed than others.

The results show that there is a notable difference between the mean time to remove a correctness finding compared to a maintainability finding: Removing a maintainability finding takes roughly half an hour longer compared to a correctness finding. A possible explanation is that bug fixes often require only few tokens to be changed whereas refactorings mostly involve larger code blocks or multiple locations to be changed. The median, however, is identical for both categories.

Average Number of Modification Table 5.3 shows the results. For each category, it shows the number of findings that were removed, the mean and the median of the changes per month to the code entity after the finding's removal, as well as the absolute and relative number of code entities that remained unchanged after the finding's removal. It shows that the corresponding file or method for a finding is changed on average 0.44 times per month. This corresponds to almost once every second month. The median is less than half of the mean, indicating that some entities changed significantly more often than others.

For all categories measured on file basis, the mean varies per month between 0.35 and 0.63. The only exceptions are empty classes with only 0.08 changes per month. For most newly documented empty classes, the benefit is rather expected on the readability side. Among the rest, the change frequency is the lowest for removed naming violations and code clones. For clones, this might be due to counting subsequent changes only in the files from which the clones were removed but not in the files to which they were extracted to, e. g. if clones were unified in a common base class.

		Time [Minutes]			
Finding Category	#	*Min*	*Mean*	*Median*	*Max*
All	215	1	59	30	870
Correctness (FindBugs)	100	1	47	30	660
Maintainability (¬FindBugs)	115	2	71	30	870
Cloning	43	10	118	60	870
Unused Code	6	30	90	45	300
Long Method	10	15	74	53	240
Empty Class	9	30	47	60	60
FindBugs	100	1	47	30	660
Deep Nesting	2	20	40	40	60
Large Class	2	30	30	30	30
Task Tags	18	5	29	15	120
Naming	19	5	24	30	60
Commented-Out Code	6	2	11	5	30

Table 5.2: Time Spent for Removing Findings

In contrast to files, the surrounding methods are changed less frequently; only 0.1 or 0.15 times per month and every third method remained unchanged after the finding's removal. This is not surprising since methods naturally change less frequently than files. Still, we expect the benefit of an individual change to a method higher compared to files, because we can be certain that the change happens exactly where the code quality was improved. For files, there is the probability that changes happen in parts different from those where the code quality was improved. Future work might improve this.

It is worth noting that besides saving time, there are other benefits of quality improvement that we identified within this case study. However, these benefits are currently not included in our model. We discuss this in Section 5.6.

5.2.4 Conclusion

On the one hand, the study provides empirical data for costs of finding removal. On the other hand, it provides evidence for one of the variables in estimating the benefit of finding removal, i. e. the modification frequency. On the cost side, the results indicate that external findings are on average significantly faster to remove than internal findings. On the benefit side, the study shows that a file is changed on average about once every second month after a finding was removed.

			Changes / Month		Unchanged	
Finding Category	*Entity*	#	*Mean*	*Median*	#	%
All	Both	5,259	0.44	0.20	621	11
Task Tags	File	704	0.63	0.39	31	4
Unused code	File	889	0.60	0.28	37	4
Commented-Out Code	File	810	0.59	0.31	71	8
Large Class	File	50	0.48	0.33	3	6
Cloning	File	1,762	0.35	0.18	183	10
Naming	File	410	0.35	0.12	108	26
Long Method	Method	335	0.15	0.05	97	28
Deep Nesting	Method	239	0.10	0.05	70	29
Empty Class	File	60	0.08	0.03	21	35

Table 5.3: Number of Modifications (Changes) after Removing Findings

5.3 Model Instantiation

Based on the conceptual model, first industrial evidence, and existing work, we exemplary instantiate the model. The instantiation serves three purposes. First, it tests whether the conceptual model is applicable in practice. Second, it provides an order of magnitude of costs and benefits of finding removal. Third, it shows current limitations of the instantiation due to missing reliable data. Hence, it also provides direction for future research.

To instantiate the model, we strive to provide reference points for each variable in Equation 5.2 and 5.7. When providing reference points, we clearly mark how reliable our assumptions are (see Table 5.4). Some assumptions are based on empirical or anecdotal evidence, some are even based on pure gut feelings obtained from personal three-year experience as software quality consultant in the insurance, avionic, mobile, and banking domain. As we outline the reliability of each assumption, the instantiation can serve as reference for future work to close existing gaps. Based on current state-of-the-art, our instantiation cannot serve as a profound scientific cost-benefit evaluation. Hence, the results are not to be seen as a thorough return-on-invest calculation that can prove managers in industry how much money they will save per cent spent on quality improvement. Rather, the instantiation is to be seen as a *gedankenexperiment*[1].

Some of the variables in Equation 5.2 and 5.7 depend of the finding's category, as described in Section 5.1. To show the practicability of our model, we chose one exemplary category, namely large classes. For these categories and to our best knowledge, we can rely on the most existing work, making our instantiation more plausible and reliable. Nevertheless, our instantiation remains incomplete, neglecting other categories. We discuss this threat to validity in Section 5.6.

[1] This is a German expression describing a "thought experiment which considers some theory for the purpose of thinking through its consequences. Given the structure of the experiment, it may or may not be possible to actually perform it" [1].

Variable	Instantiation	Rationale / Assumption	Reliability
cost_removal(t)	30 min	Empirical evidence (Section 5.2)	🟨
s_testing(Δt, scope)	0 min	No evidence available (*unknown*)	🟥
avg_s_read(FILE)	2.4 $\frac{\text{min}}{\text{read}}$	Relative speed up: 20% based on [3], absolute reading time per task: 12 min based on [56]	🟧
avg_s_mod(FILE)	3.6 $\frac{\text{min}}{\text{mod}}$	Relative speed up: 33% based on argumentation, absolute modification time per task: 11 min based on [56]	🟥
avg_n_read(1m, FILE)	5 $\frac{\text{read}}{\text{month}}$	avg_n_read = 10 · avg_n_mod based on [66]	🟧
avg_n_mod(1m, FILE)	0.5 $\frac{\text{mod}}{\text{month}}$	Empirical evidence (Section 5.2)	🟨

Table 5.4: Instantiation of the Model For a Large Class with the *Scope* of a File and $\Delta t = 1$ month

We use the finding category *Large Class*, because existing work can provide data additional to our own previous empirical study [3, 4, 29]: To our best knowledge, this existing work is the only one that tries to measure the negative impact of a *God Class* [4, 29] or the anti-pattern *Blob* [3] on program comprehension. While the concept of *God Class* directly maps onto our finding *Large Class*, the anti-pattern *Blob* considers the class' complexity and cohesion in addition to size. We discuss this threat to validity in Section 5.6.

In the following, we instantiate the variables in Equation 5.2 and 5.7. For each variable, we discuss our assumptions and their reliability. Table 5.4 summarizes the assumptions. A colored square in Table 5.4 indicates how reliable the assumptions are based on our estimation. With no existing case study being large-scale or even generalizable, no assumptions receives a green reliability rating. Instead, they are classified as yellow (empirical evidence based on few data points), orange (questionable combination of empirical evidence), and red (no empirical evidence, gut feeling).

cost_removal To approximate the removal costs, we rely on the data from our own empirical study in Section 5.2. Based on two data points, we showed that refactoring a large class took 30 minutes. The low number of data points limits the generalizability of the data. However, considering all 215 data points of the previous study, the mean removal costs were 30 minutes as well. Hence, we assume that an increased number of data points for large classes would not change our variable instantiation significantly.

s_testing As discussed in Section 5.1, modeling the savings of finding removal in terms of testing is very difficult. We have neither scientific nor anecdotal evidence of the order of magnitude and, additionally, we can not even provide reasonable gut feelings. Hence, for our instantiation, we assume the worst case, i. e. that the savings in testing are zero. Consequently, our model will state that the benefits outweigh the costs later than in reality given the assumption that the savings in testing are in fact greater than zero.

avg_s_read To our best knowledge, the work in [3] is the only work that quantifies the speed up gained by cleaner code specifically for the code reading activity. The authors measure how much faster participants can answer questions with respect to programs which do or do not contain *Blobs*. They cannot find evidence for or a positive reading speed up on all case study systems—one system actually reveals a negative impact. Similarly, the study in [4] also could not prove a speed up for solving tasks on refactored code (including code reading and modification) for all their case study systems. The authors in [4] argue that this might be due to developers not being used to the style of refactored code. As they found the tasks on refactored code yet to be completed cleaner, they conclude that the benefit of cleaner code may not occur immediately but only after a certain period of time.

Both studies show how difficult it is to obtain accurate data as the studies are heavily influenced by the developers skills, for example. Additionally, the task completion time as well as the reading speed up can be influenced not only by the developers, but also by other factors—size of the task, size of the code base, personal developers' preferences, or development tooling to name just a few.

In the absence of accurate data, we instantiate our model with data from the two study systems in [3] that revealed a positive speed up. Unfortunately, the provided data does not match our variable exactly: As the given questions were artificial, they do not match actual maintenance tasks but were significantly smaller. Hence, while we can derive a relative speed up in terms of code reading, we cannot yet approximate the absolute time savings for a real maintenance task. For the relative speed up, we use the minimum of both results to minimize the error in our model instantiation. The minimum relative speed up of the study was 20%.

The study in [56], in contrast, provides data how much time developers require to perform a maintenance task. The authors provide a fine-grained differentiation between different maintenance activities and conclude that, on average, a maintenance task takes about 70 minutes (including simulated interruption time by coworkers). For each task, developers spent about 12 minutes on code reading (which corresponds to a fifth of their non-interrupted time per task).

Combining the evidence from both studies, we estimate the average speed up for reading cleaner code with the relative speed up taken from [3] and the absolute reading time per task from [56]: $0.2 \cdot 12 \text{ min} = 2.4 \text{ min}$. However, this assumption has no reliable scientific evidence and needs to be treated with caution. At least, the result does not violate our gut feelings and sounds plausible.

avg_s_mod To our best knowledge, there is no empirical evidence how much faster a refactored class can be modified. Similar as to the speed up in reading, we expect the speed up in modification to be very difficult to investigate scientifically sound. In addition to the large number of parameters regarding reading activities, we expect even more parameters to occur with respect to modification activities [41]: How fast a developer can modify code depends additionally on how fast he can type, how well he uses IDE support (e. g. auto completion) and how much he uses the mouse versus the keyboard interaction.

In the absence of data, we assume the speed up in modification to correlate to the size of the code affected by the modification: If a large class C with z lines of code was refactored and split into two classes A and B with x and y lines of code, we assume that class A can be modified faster with a relative speed up of $(1 - \frac{x}{z}) \cdot 100\%$. This assumption is based on pure reasoning. Currently, there is no evidence for it.

Based on anecdotal evidence from our customers, developers usually consider a class with more than 400 lines of code to be worth to refactor. Based on our experience, not every developer agrees to the threshold of 400 lines of code but a majority does. Additionally, it depends on the content of the class; a class containing only UI code, for example, might still be easy to understand even it has more than 400 lines of code. Hence, even though the threshold is not completely reliable, it provides a first reference point. To complete our gedankenexperiment, we assume that a class with 600 lines of code is refactored in classes containing roughly 400 and 200 lines of code. Hence, the relative speed up would be 33% for the larger, newly created class. However, this instantiation is completely built up on gut feeling and not scientifically reliable.

The study in [56] provides data how much time developers spend on modifying code while performing a maintenance task: The study claims that on average, a developer spends 11 min modifying code. Again, this data cannot be generalized but provides first empirical evidence. Combining the absolute time for code modification from [56] with our hypothetical relative speed up of 33%, the overall speed up in modification is instantiated with $0.33 \cdot 11 \text{ min} = 3.6 \text{ min}$.

avg_n_read While we have evidence of how much code reading is involved in performing a maintenance task [56], we have less scientific evidence for how many tasks a single class is read in a given period of time. In the absence of scientific data, we use anecdotal evidence from [66] stating that code is read ten times more than written. As we have empirical evidence from our own study how often code is written (see next bullet), we can extrapolate this to approximate the number of reads.

avg_n_mod Based on our study in Section 5.2, large classes are modified approximately one time within two months after they were refactored. The result from the study is not generalizable but provides a first reference point based on empirical evidence.

Given the instantiation of all variables in Equation 5.2 and 5.7, we can now calculate the time interval Δt in which the refactoring of a large class would pay off based on our assumptions. Therefore, we instantiate Equation 5.1 as follows:

$$
\begin{aligned}
\exists t_1, x, \Delta t = [t, t_1] = x \text{ months}: \\
30 \text{ min} < \; & 2.4 \frac{\text{min}}{\text{read}} \cdot 5 \frac{\text{read}}{\text{month}} \cdot x \text{ month} + \\
& 3.6 \frac{\text{min}}{\text{mod}} \cdot 0.5 \frac{\text{mod}}{\text{month}} \cdot x \text{ month} + \\
& 0\text{min} \\
\iff & \\
30 \text{ min} < \; & 12x \text{ min} + 1,8x \text{ min} \\
\iff & \\
2.17 < \; & x
\end{aligned}
\tag{5.8}
$$

Based on our assumptions, the refactoring of a large class would pay off after roughly three months, taking an error of up to approximately 30% into account. Hypothetically, after a year ($x = 12$ months), a developer would have saved approximately 135 minutes, 2.25 hours working time. With an estimated average salary of 73,000 US Dollars[2] for a developer with five to nine years working experience, one person hour costs about 40 US Dollars.[3] Hence, within a year, the company would save roughly 90 Dollars for a single refactored large class.

5.4 Discussion

Our model and its empirical studies serves as a first starting point towards a return-on-invest calculation for finding removal. We instantiated th model exemplary for the specific finding category of large classes. In the following, we discuss our findings.

Based on the exemplary instantiation, the model reveals an order of magnitude that the refactoring of a large class is likely to pay off within few months. More specifically, we calculated that the refactoring of a large class could save a company about 90 Dollars within a year. Considering the results from our quantitative study in Chapter 4, a Java systems contains about 80 overly large classes on average. Hence, if all classes were refactored, the company could save about 7200 US dollars within a year. However, this is only a gedankenexperiment. In practice, it is not feasible and even risky to refactor all classes at once. Hence, the refactoring and the pay-off would both spread out over a certain time interval.

Nevertheless, the gedankenexperiment reveals that the savings calculated with our model can reach thousands of dollars. However, as explained in Section 5.1, these numbers have to be treated with caution, because they are based on weak assumptions or partly even gut feelings on the one side. On the other side, they do not consider interference effects between multiple findings, for example, and other positive benefits from quality improvement such as

[2] http://www.computersciencedegreehub.com/software-development-salary/

[3] Calculation based on 6 weeks of vacation, hence 46 working weeks per year, and 40 working hours per week.

increased quality awareness, increased developer motivation, or reduced defect probability. Ammerlaan et al. suggested that refactored code is beneficial after a certain learning curve and that "good quality code prevents developers from writing sloppy code" in the first place [4]. In fact, the savings might be significantly larger than modeled in our theory: Case 3.5 showed that if static analyses are ignored, the costs of non-conformance may even result in the redevelopment of the system.

Overall, our instantiation of the model revealed the following: First, it is possible to instantiate the model in practice. Second, gathering scientifically sound evidence to instantiate the different variables is difficult. Third, based on current-state-of-the-art and this thesis, there is still evidence missing for some variables. We further discuss this as future work in Chapter 8.

5.5 Model-based Prediction

With a conceptual model, we analyzed when the removal of a finding would pay off. However, for a prioritization of individual findings, a cost-benefit analysis based on average experience values from the past is not sufficient. Instead, we need to predict the benefit of removing a specific finding in the future. In particular for internal findings, the benefit is harder to predict as it depends on future maintenance of the affected code region. Knowledge about the future maintenance of the system is required.

In this section, we examine in how far static analyses can be used to predict future maintenance. More specifically, we conducted a case study to investigate how the existing code base and its history can be used to predict future changes. It is commonly assumed that "the more changes that have been made in the (recent) past, the likelier it is that this part of the software is still unstable and likely to evolve" [67]. We test this assumption and examine whether the change history of a system can be used to predict the changes in its future. This would give us an approximation for the variable $avg\_n\_mod(\Delta t, \text{scope}(f))$ in Equation 5.7 of our conceptual cost model in Section 5.1.

Problem Statement Estimating the benefit of removing internal findings requires knowledge about the future maintenance: If we can predict the future changes to a piece of code, we can better estimate how much this piece of code is worth to be cleaned up. However, there are currently no precise prediction models available. This study examines whether the change frequency of the past serves as a good predictor of future change frequency.

5.5.1 Data Collection

As data set for the study, we reuse the Java data from Paper **B**: the dataset contains the change history of 114,652 methods from ten case study systems. Nine of them were open-source, one of them was an industrial system. Overall, the data set contains an analyzed history of 72 years. For each method, we extracted its changes from the initial commit to the head revision in the analyzed history.

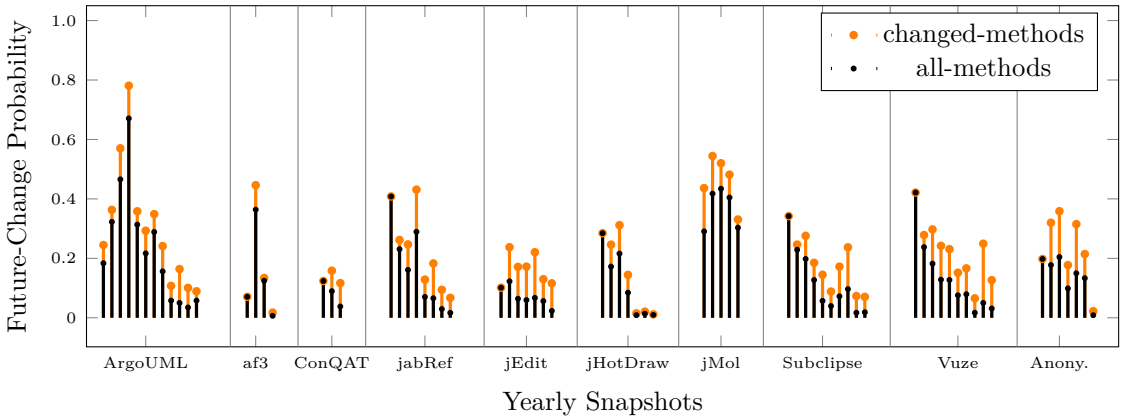


Figure 5.1: Data of Maintenance Prediction Study

5.5.2 Evaluation

To evaluate whether the change frequency in the past is a good predictor for the change frequency in the future, we use yearly snapshots of the history (e. g. May, 1st 2009, May, 1st, 2010 etc.). For each snapshot, we create two method sets: First, the set of methods that were changed in the year prior to the snapshot. We refer to this set as the *changed-methods*. Second, the set of methods that existed prior to the snapshot—referred to as *all-methods*. The latter is thereby a super set of the first set as it comprised both the changed and unchanged methods. For both sets, we calculate the probability that a method is changed in the year after the snapshot and refer to it as *future-change probability*. The future-change probability denotes the relative frequency of the methods that were changed in the year after the snapshot.

Hypothesis The case study is based on the following hypothesis: If the change frequency in the past is a good predictor for the change frequency in the future, then the *future-change probability* should be significantly higher for the *changed-methods* than for *all-methods*.

δ	A.UML	af3	ConQAT	jabRef	jEdit	jHotD.	jMol	Subclipse	Vuze	Anony.	**Sum**
0%	12/12	3/4	2/3	7/8	6/7	6/7	5/5	9/10	9/10	6/7	**65/73**
5%	8/12	1/4	2/3	6/8	6/7	3/7	4/5	7/10	7/10	5/7	**49/73**
10%	3/12	0/4	0/3	2/8	4/7	0/7	2/5	1/10	4/10	3/7	**19/73**
15%	0/12	0/4	0/3	0/8	1/7	0/7	0/5	0/10	1/10	2/7	**4/73**

Table 5.5: Snapshots with Higher *Future-Change* Probability for the *Changed-Method* Set than for the *All-Method* Set

5.5.3 Result

Figure 5.1 and Table 5.5 show the results of the evaluation. Out of 73 snapshots, our hypothesis was true for 65 snapshots. However, the *future-change probability* for the *changed-methods* was often only marginally higher than for *all-methods*. Hence, we also evaluated in how many cases the *future-change probability* for the *changed-methods* was at least 5%, 10%, or 15% higher. With a threshold difference of at least 5%, our hypothesis was still true in 49 out of 73 snapshots (67%). With a 10% threshold difference, the hypothesis only held in 19 out of 73 snapshots (26%) and with 15% difference, only in 4 snapshots (5%).

5.5.4 Discussion

Based on these results, the change history in the past is helpful for the prediction of future changes; however, it does not improve the prediction significantly: Only 26% of the snapshots, the *future-change* probability improved by more than 10%. We suspect that the current code base and its history can only provide limited information for its prediction because future maintenance is rather driven by non-code related influence factors: With about 60%, maintenance is mostly *perfective*, i. e. changes comprise adapting existing or adding new functionality due to changes in the business process or new user requests [24, 63, 71]. Instead of deriving changes from the existing code base, they must be rather derived from changing requirements and new market situations. Hence, we assume that the automated, tool-based prediction of future maintenance is inherently limited based on existing software artifacts. This contradicts previous assumptions that entities that are changed frequently in the past are likelier to change in the future [67].

5.5.5 Conclusion

The results show that static analysis of the existing code base and its history have limited impact to predict future maintenance. We assume that—in general—existing software artifacts cannot provide sufficient information to predict future changes, as these are usually rather derived from changing requirements [24, 63, 71].

5.6 Threats to Validity

As our model and its instantiation are based on a variety of assumptions, it faces a large number of threats to validity. In the following, we discuss them.

In the conceptual model, costs and benefits are formalize for a single finding. In our model, the removal costs as well as the benefits are defined for an *individual* finding. Existing work, however, has shown that while a single finding has moderate impact on program comprehension and software maintenance, a combination of them can have a significantly stronger negative impact [3, 106]. In [3], this effect of interference has been shown for

two findings and cannot yet be generalized even though its existence seems plausible for other findings as well. Additionally, there is first scientific evidence that the occurrence of multiple findings does not only increase the negative effect on maintainability but also on program correctness: The work in [34] provides one industrial case in which classes with more findings are also more error-prone. As it is out of scope of this thesis to prove the interference effect for all findings (or even just a subset), our model does not include them. Consequently, if the core condition of our model is satisfied, it gives only an upper-bound for Δt: Due to possible interference effects, it might pay off earlier to remove a finding.

Modeling costs and benefits for a single finding, the result depends on the nature of the finding's category. Code clones might impact maintainability differently than naming convention violations or large classes. In our empirical study (Section 5.2) we tried to mitigate this threat by providing finding-specific data. However, these data are only a first reference point and neither generalizable nor exhaustive. Hence, our model is to be understood only as a first step towards the formalization of a return-on-invest calculation of finding removal. It is generic enough to be applied on all static analysis results but does not yet comprise an instantiation for each one of them.

The model uses average as approximation for the benefit in reading and modification. When modeling the savings with respect to reading (*s_reading*), we approximated them with the average number of reads multiplied with the average save per read (see Equation 5.7). The same holds for modification, respectively. The approximation with the average introduces a certain fuzziness to the result. Yet, based on existing related work and our own empirical study, we were only able to provide an instantiation for the average approximation. It remains future work to conduct a more fine-grained case study.

The model considers benefit only in terms of reading, modification, and testing. Aside from the positive impact on reading, modification, and testing, removing findings is beneficial in other dimensions as well. Existing work as shown that cleaner code encourages developers to produce cleaner code in the first place [4]. The actual benefit expands to the reduction in non-conformance costs resulting from *not* introducing more future findings. However, modeling this dimension of the benefit would require a prediction model of newly introduced findings in the future. For the purpose of this thesis, we preferred to keep the model as simple as possible to be able to provide a first instantiation as in Section 5.3.

Our own case study (Section 5.2) revealed several other additional benefits. However, they are very hard to quantify. These benefits include the knowledge transfer between developers, monthly internal quality discussions which otherwise would have not taken place, higher quality awareness, higher developer motivation to work on clean code and lower risk of introducing bugs in the first place. Hence, our benefit considerations are to be seen as a lower bound of the real benefit. The costs of removing a finding might in fact be redeemed earlier than stated in our model.

Our own empirical data to approximate benefit is not exhaustive. With our empirical case study (Section 5.2), we provided reference data only in terms of modification data; we approximated $avg\_n\_mod(\Delta t, \text{scope}(f))$. Approximating the other variables in Equation 5.7 was not feasible based on the case study set up. Measuring the number of reads, $avg\_n\_read(\Delta t, \text{scope}(f))$, would have required a tracking mechanism of developers' navigation in the IDE as well as a time tracking tool for the developers' reading activities. Both was not available for installation at our customer's site. Additionally, measuring the average savings per read, per modification, as well as for testing ($avg\_s\_read(f, \text{scope}(f))$, $s\_testing(f, \Delta t, \text{scope}(f))$, $avg\_s\_mod(f, \text{scope}(f))$) was not possible as we had no evidence of what would have happened is a finding was not removed. For our discussion in Section 5.3, we had to rely on existing related work and were not able to contribute data from this case study.

Our empirical data to approximate removal costs is not fine-grained. In Section 5.2, we provided first empirical data on the removal costs for findings in ten different categories. These findings comprised both internal and external findings. This data gave first indication for the overall removal costs in Equation 5.2. However, it does not allow a separation between *c_reading*, *c_modification*, *c_testing*. For an analytical, retrospective calculation to decide whether the core condition in Equation 5.1 was satisfied for a removed finding, having evidence for the sum suffices. For a constructive application of the model, e. g. to predict cost and benefit and prioritize findings based on the best ratio, knowledge about the individual summands would allow a more fine-grained prediction. However, we currently do not have this data. How we use the cost prediction in our prioritization approach will be addressed in Section 6.2.3.

The instantiation is done for only one category of findings. Our instantiation has shown that the model is applicable for the category of *large classes*. Nevertheless, our instantiation remains incomplete, neglecting other categories. Due to the large number of different existing static analyses, we consider it to be infeasible to provide an instantiation for all of them, given the scope of this thesis. Hence, this thesis can only provide a starting point and a reference for future replication for other categories.

Yet, we believe that the instantiation can be done similarly for other categories. For many other categories, we have the corresponding empirical data from our own study in Section 5.2, affecting the instantiation of *avg_n_mod*, *avg_n_read*, and *cost_removal*. Further, our assumption for instantiating *avg_s_mod* can be partially transfered as well: The removal of long methods, deep nesting, or code clones reduces the size of the code affected by a modification as well. For other findings, such as naming convention violations or code comments, the assumption cannot be transferred. This remains future work. Finally, for *avg_s_read*, empirical evidence is missing the most. However, as we outlined the difficulties in obtaining reasonable data, this is out of scope of this thesis.

The instantiation for the finding *Large Class* uses related work from the anti-pattern *God Class* and *Blob*. Our instantiation of the model for large classes relies on empirical data from existing work on god classes [4, 29] or on the anti-pattern *Blob* [3]. While the concept of *God Class* directly maps onto our finding *Large Class*, the anti-pattern *Blob* considers the class' complexity and cohesion in addition to size. As the anti-pattern considers more code characteristics than our finding, it adds fuzziness to our instantiation. Possibly, the negative effect of Blobs on program comprehension is larger than the effect of large classes. We tried to mitigate this threat by taking the minimum measured negative effect in the case study from [3]. To our current best knowledge, the data from [3] is the only scientific evidence that can enrich our model in current state-of-the-art (aside from our own, non-exhaustive empirical study). Yet, this threat to validity needs to be addressed by future work.

The instantiation for the finding *Large Class* is based on a large variety of assumptions. As the model is designed as a gedankenexperiment and scientifically sound evidence is missing and very difficult to gather, our instantiation depends on a large number of assumptions. These assumptions reveal different reliability and influence the outcome of the instantiation. We believe we have outlined the reliability of our assumptions in detail in Section 5.3 and summarized them in Table 5.4. To our best knowledge, more reliable related work does not exist and the required studies for the missing data are out of scope of this thesis.

5.7 Summary

In this chapter, we provided a conceptual model to formalize costs and benefits of finding removal. While the benefit of removing external findings is quite obvious, the model shows that removing internal findings is beneficial only if the corresponding code is maintained. However, the model also shows that static analyses have limited impact to predict future maintenance. Future changes are usually rather derived from (unknown) changing requirements and evolving market situations. Hence, we assume that existing software artifacts generally cannot provide sufficient information for an automated maintenance prediction.

On the cost side, the model revealed that external findings are usually very easy to remove. Internal findings, however, create higher removal costs. As they are also much larger in number, only a small subset of them can be removed within one development iteration. While we cannot reliably prioritize based on the benefit of their removal, we can yet prioritize based on their removal costs. With low removal costs, it is easier to reduce the inhibition thresholds of developers and motivate them to remove internal findings that do not reveal immediate reward. Based on logical reasoning, the model showed that removing internal findings is less costly when aligned to ongoing development: If a finding is removed in code that has to be changed anyway, overhead in code understanding and testing can be saved—performing a change request already comprises both.

"It is not enough for code to work."

Robert C. Martin

6 A Prioritization Approach for Cost-Effective Usage of Static Analysis

Even though many companies employ static analysis tools in their development, they often do not succeed in improving their code quality in the long term [85]. In this chapter, we present a prioritization approach for the effective usage of static analysis. The approach is derived from the results previously gained in Chapter 4 and 5.

After outlining the challenges developer face when applying static analyses, we derive the four key concepts of the approach in Section 6.1. Each of the four key concepts is explained in more detail in Sections 6.2 – 6.5. Section 6.6 summarizes this chapter.

6.1 Key Concepts

Based on personal three-year industry experience as software quality consultants, developers face several challenges when introducing static analyses to a software system developed on the brown-field. These challenges are based on personal, hence subjective experience. The personal experience matches the experience of the quality consulting company CQSE within hundreds of customer projects. However, even though CQSE has a large number of customers, our assumptions might not apply to all software systems in all domains. Yet, we believe they apply for a substantially large subset of long-lived software.

As outlined in Chapter 1, we identified five challenges and repeat them briefly here: First, the number of findings is too large to be fixed at once (Challenge *Number of Findings*). Second, different findings reveal different benefits on removal (Challenge *Benefit of Removal*). Third, findings comprise false positives and non-relevant findings which disturb developers (Challenge *Relevance of Findings*). Forth, while removing findings, new findings can be created that need to be considered as well (Challenge *Evolution of Findings*). And fifth, change requests often dominate finding removal and absorb all resources (Challenge *Management Priority*).

Due to the challenges of applying static analyses effectively in practice, developers require a substantiated approach to effectively use static analyses. In this section, we give an overview of our proposed approach. Generally, the approach is applicable to all static analyses that result in findings in the code—with a finding indicating the risk of increasing future non-conformance costs. Based on our definition of software quality (see Chapter 3), findings address a variety of different quality aspects—from functional suitability over security and usability to maintenance. For example, findings pointing to programming faults are likely to create non-conformance costs resulting from a system failure, vulnerabilities to security

attacks can create significant costs for data losses, and unmaintainable code can increase costs for future change requests.

The approach addresses the current challenges in practice with four key concepts. These are described briefly in the following, and in much detail in subsequent sections.

1. **Automatic Recommendation**—*An automatic recommendation prioritizes the findings.* We provide a fully automatic and tool-supported mechanism which recommends a certain set of findings for removal. This reduces the large amount of findings to a feasible size (Challenge *Number of Findings*). The approach is based on the results from the quantitative study (Chapter 4) as well as on cost and benefit considerations as outlined in our cost model (Chapter 5). Therefore, it differentiates between *external* and *internal* findings to take the developers' motivation/ inhibition threshold into account (Challenge *Benefit of Removal*).

2. **Manual Inspection**—*Manual inspection filters relevant findings.* We require a manual component to inspect the recommended findings and to accept the set (or a potential subset) for removal. By manually accepting or rejecting findings, false positives and non-relevant findings can be eliminated (Challenge *Relevance of Findings*). As a result, automatically recommended and manually accepted findings are prioritized to be removed from the system.

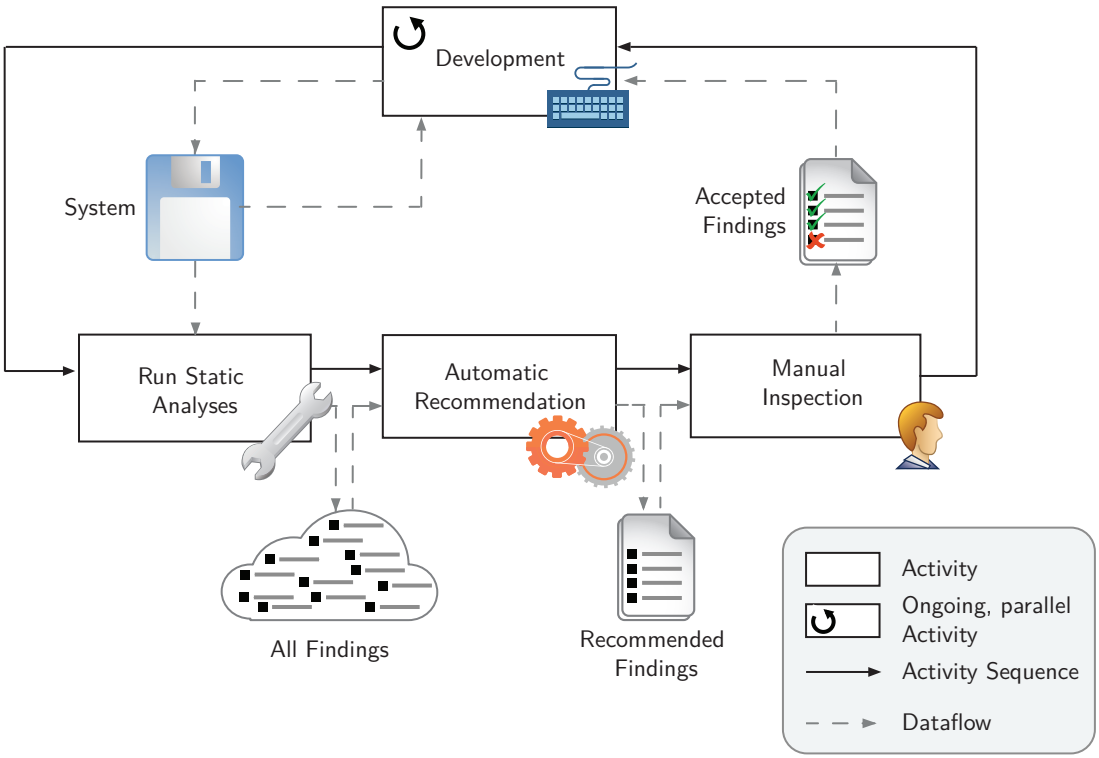


Figure 6.1: Overview of the Approach

		Key Concepts			
		Automatic Recommendation	Manual Inspection	Iterative Application	Quality Control
Challenges	Number of Findings	X			
	Benefit of Removal	X			
	Relevance of Findings		X		
	Evolution of Findings			X	
	Management Priority				X

Table 6.1: Mapping between Key Concepts and Challenges

3. **Iterative Application**—*An iterative application ensures continuity.* Our approach suggests applying the prioritization iteratively during the development process. Consequently, the prioritization takes the ongoing development and maintenance of the code base into account (Challenge *Evolution of Findings*).

4. **Quality Control**—*A quality control process raises management awareness.* To guarantee enough resources for finding removal, the approach embeds the iterative prioritization in a quality control process. The process raises management awareness for code quality and provides transparency about the usage of the provided resources (Challenge *Management Priority*).

Figure 6.1 gives an overview of how the automatic recommendation and the manual inspection are applied iteratively: At the beginning of each iteration, a set of static analyses is ran on the system, resulting in a set of findings. The automatic recommender selects a subset of the findings which are then inspected manually. During manual inspection, findings can be accepted or rejected. The accepted findings are scheduled for removal during the iteration. While development continues and the system is further modified, the static analyses are run again at the beginning of the next iteration and the prioritization repeats. Table 6.1 summarizes how the key concepts address the identified challenges in practice.

In the remainder of this chapter, we describe the automatic recommendation and the manual inspection in Section 6.2 and 6.3. Section 6.4 focuses on the iterative application. The quality control process and its required tool support are described at the end of this chapter in Section 6.5.

6.2 Automatic Recommendation

The main component of our prioritization mechanism is the fully automatic recommendation: Given a set of current findings in a system, the recommender outputs a subset of

them as *recommended* findings. The set of recommended findings should be of feasible size as, in a second step, these findings are be manually inspected for acceptance. In general, the automatic recommendation is based on cost and benefit considerations as outlined in our cost model in Section 5.

Within the large amount of current findings, different findings reveal a different benefit when being removed (see Section 5.1). Our approach takes the different benefit into account by differentiating between *external* and *internal* findings (see Equation 5.3). Based on our experience, developers are willing to remove external findings almost instantly that point, e. g. to a potential programming error or security vulnerability. Removing these findings has an immediate pay-off, e. g. to remove a fault in the system. Contrarily, developers are harder to motivate to remove internal findings impacting the maintainability of the code, e. g. long methods or code clones. Removing these findings does not improve the user-visible behavior of the system. Hence, it has no short-term benefit but only the long-term benefit of achieving a more maintainable code base.

6.2.1 Findings Differentiation

This differentiation between external and internal findings is aligned with the two different kinds of non-conformance costs (see Definition 4): on the one hand, non-conformance can stem from an *external* failure of the system in production (e. g. costs for fixing field defects—debugging, retesting, deploying hot fixes); on the other hand, non conformance can also result from *internal* failure of the system (e. g. costs for restructuring the code to make it more maintainable, or adapting the code to make it portable to a different technical platform).

External Findings These findings present a high risk of leading to a system failure observable by a user. The notion of risk is context and domain dependent: High risk findings point, for instance, to a programming fault or to code weaknesses causing security vulnerabilities. Examples include null pointer dereferences, synchronization problems, or vulnerabilities for SQL injections.

However, not all external findings necessarily cause a system failure in practice. For example, the null pointer dereference might not cause an exception, if the code is not executed. If the exception is indeed thrown, it still could be catched and suppressed during later program execution. Nevertheless—if uncatched—the exception can create a user-visible failure and, hence, the null pointer dereference poses a certain risk.

Internal Findings These findings cannot be noted by the user of the system. Nevertheless, they are likely to increase non-conformance costs resulting from internal failure as they hamper readability, understandability, or changeability of the code. Examples include structural findings like long classes and long methods, redundancy findings like code clones, or documentation problems like missing interface comments.

While most analyses reveal only internal or only external findings, some analyses can reveal both. One example includes the detection of duplicated code that can be configured in many different ways: If configured to detect only identically duplicated pieces of code, the analysis reveals internal findings because these clones represent an increased change effort when changes have to be propagated to all instances. In contrast, if the analysis is configured to detect inconsistent clones [49], differentiating between external and internal findings becomes more complex: In case of an unintentional inconsistency, the clone is likely to represent a programming fault [49]. Due to this risk—existing work shows that the probability of an inconsistent clone representing an actual fault is between 25% [68] and 50% [49]—these clones are classified as external findings. Otherwise, if the inconsistency was created on purpose, it does not represent a fault, but still increases the change effort. In this case, the inconsistent clone represents an internal finding. Using the examples of code clones, we describe in Section 6.2.2 how to classify findings as external or internal.

6.2.2 Recommending External Findings

As external findings reveal a risk of *current* system failure, our approach adds all external findings to the set of recommended findings. However, this poses constraints on the selected analyses: Only those analyses should be run that result in a feasible number of findings that are *risky* enough to be manually inspected. Consequently, the recommendation of external findings comprises two steps: First, the appropriate analyses have to be selected. Second, amongst their findings, the external findings have to be classified.

Selecting Domain-Dependent Analyses

To obtain findings that are *risky* enough to be all manually inspected, the appropriate analyses have to be selected. The notion of *finding's risk* depends on two different aspects: first, the risk of a finding is system-specific and depends on the system domain as well as its context. Second, it can further depend on the *type of code* the finding is located in. As the selection of the domain-dependent analyses requires system-specify context knowledge, our approach provides no automated support for it. This has to be done manually.

Domain and Context Dependency Depending on the domain and context, the risk of a finding can vary. For example, for systems that operate with an online-accessible SQL database, findings representing vulnerabilities for SQL injections are very risky. In contrast, if the data-base is not accessible by a user, employing such an analysis reveals much less failure risk for the system (the number of potential attackers gets reduced to internal users or administrators). Hence, analyses to detect external findings have to be selected depending on the system's domain.

In the domain of Java programs, for example, a variety of bug pattern detection tools exist: Findbugs[1], for instance, performs a data flow analysis on byte code to detect bug patterns that likely point to programming faults. As theses programming faults can lead to a null

[1] www.findbugs.sourceforge.net/

pointer exception that, in turn, might cause an unintended behavior, these findings contain a risk for current system failure and should be consolidated immediately. These findings have a high correlation to actual faults [6, 7, 20, 97]. Usually, they are small in number (as shown by the quantitative study in Chapter 4), and quite fast to remove (see empirical data in Chapter 5.2), Findbugs provides a suitable set of rules to detect external findings in the Java domain [39, 92]. Similar tools exist for other domains as well [39], such as Coverity[2] for C and C++ [11].

Beyond static analyses for correctness, also many static analyses exist to address security. Tools like Coverity[3] or Klocwork[4] offer automated static analyses to identify potential security vulnerabilities [32]. Examples comprise vulnerabilities for SQL injections, cross-site scripting or cross-site request forgery. An SQL injection can, for instance, result in data loss or data corruption and, hence, lead to a user-visible failure of the system.

Code Type Dependency In addition to the domain and context dependency, the risk of a finding can also depend on the type of code it is located in: Static analyses that are suitable for application code may not be suitable for test code at the same time and vice versa [5, 94]: For example, findings in *test code* cannot lead to user-visible system failure by the definition of test code being used only for testing the application, but not for running in production.

For findings in test code, their risk has to be assessed separately from their risk in application code. Some findings may reveal the same risk in application and in test code, others are relevant for only one or the other. For example, a null pointer dereference is likely not desirable neither in test nor in application code. The null pointer dereference will cause the test to fail, maybe even before the intended functionality was actually tested. As another example, inconsistent clones might be often detected in test code, if the same, duplicated test code is enriched with different test data. Whereas inconsistent clones pose a big risk in application code [49], they are likely less harmful in test code.

Consequently, appropriate analyses have to be selected manually depending on domain and context. For each analysis, the decision has to be made if its scope comprises only application or application and test code.

Classifying External Findings

Depending on the analyses, the classification of external findings requires different approaches: For some analyses, the decision whether a finding is external or not depends on the specific finding itself. For others, in contrast, this decision can be made already based on the category of the finding. In the following, we demonstrate the classification with respect to two specific examples—inconsistent clones and Java bug patterns. However, the underlying ideas can be generalized and applied in other domains as well.

[2] ww.coverity.com
[3] https://www.coverity.com/products/code-advisor/
[4] http://www.klocwork.com

Classification based on Findings When a finding category contains both internal and external findings, each finding has to be classified individually. For example, the category of inconsistent clones contains both intentionally and unintentionally inconsistent clones. Hence, the decision which inconsistency is unintentional (representing an external finding) has to be made for each finding separately. For this example, we suggest a classification algorithm in Paper **C**, [87]. Based on machine learning, the provided algorithm classifies clones as unintentional inconsistent and, hence, faulty, or non-faulty (Section 7.3).

Classification based on Category In case an analyses reveals only external findings within one finding category, the finding's risk can be derived from the category. For example in the Java domain, this classification can be done for the checks provided by the tool Findbugs[5] (The same classification can be done for bug pattern tools in other language domains). Findbugs itself provides a categorization of their checks as *correctness*, *bad practice*, *malicious code*, etc. They further assign each check a confidence rank, or—formerly—a priority (high, medium, low). Existing work has shown that engineers' perceptions of the importance of defects generally matched the rankings in Findbugs [6]. With an empirical study, Vetro et al. showed that the correctness checks of high priority are in fact a good predictor for actual faults in the software [97]. Also Ayewah et al. showed that few Findbugs rules account for the majority of defects found [7]. Based on their case study results, the following findings categories are examples representing external findings (for the full list, please refer to [97]):

- Null pointer dereference
- Nullcheck of value previously dereferenced
- Infinite recursive loop
- Impossible cast

For these findings, we have shown in the quantitative study in Chapter 4, that systems usually only reveal few of them. As these findings have a high correlation to actual faults, the corresponding Findbugs rules are suitable analyses to detect external findings.

Whereas some Findbugs check have a high correlation to actual faults, other cannot create a user-visible failure. We can draw this conclusion already based on the category description:

- Uncallable method defined in anonymous classes (this code will not be executed in production)
- Unwritten field (unused code)

However, classifying findings in some other Findbugs categories as external or internal is a grey zone. They could lead to user-visible failure, but this is much less probable. Examples comprise:

[5] http://findbugs.sourceforge.net/

- Method ignores return value (If the return value indicates an error state and is ignored, the untreated error might cause a user-visible failure. If the return value just provides additional information, it may be safely ignored.)
- Invocation of toString on an array (If this string is displayed in a UI, it might be visible for the user. However, if only used for logging, the user will not notice it)

In our approach, we suggest to tackle the grey zone of medium-risk findings not by a black-and-white automated classification as external or internal, but with a manual readjustment of the selected analyses. Further, existing work has tried to classify Findbugs warning as *actionable/actionable* or *severe*, trying to capture the developer's willingness to remove them [38, 54, 69, 78] . Even though research has made substantial improvement, existing approach reveal an ongoing need for manual inspection. We address this with our manual inspection step after the automatic recommendation (see Section 6.3)

Readjusting the Selected Set of External Analyses

It depends on the time and budget how many out of all risky findings should be recommended for manual inspection. With its iterative application, our approach gives the opportunity to readjust the selection of the analyses and the classification of their findings per iteration (see Section 6.4). If budget is low or the initial number of high-risk external findings is unexpectedly large, only the high-priority findings (e. g. the null pointer dereferences) should be recommended for manual inspection. After some iterations, when all null pointer dereferences have been resolved, the set of analyses can be refined to also include the medium-priority findings, e. g. ignored return values.

6.2.3 Recommending Internal Findings

In contrast to external findings, internal findings do not refer to potential user-visible failure of the system, but reveal a risk for non-conformance costs from internal failure. Even though the removal of internal findings is likely less urgent than the removal of external findings, they are yet highly relevant for maintenance. However, the benefit of their removal is hard to quantify for several reasons—as formalized in our cost model in Section 5.

While it is possible to some degree to analyze the benefit of an internal finding that was removed in the past (see Chapter 5), prioritizing individual findings requires the prediction of the benefit of removing a *specific* finding (see Equation 5.6). However, this benefit is hard to predict. The benefit mainly depends on future maintenance: only where future changes occur, the removal of internal findings can pay off. However, our study showed that static analyses have a very limited impact of predicting future maintenance (Section 5.5). We assume that the existing code base (and its history) generally cannot provide sufficient information to predict future changes, as these are usually rather derived from changing requirements.

Additionally, numerous developer interviews during our consultancies for long-lived software show that developers are harder to convince that internal findings should be removed:

the missing short-term benefit of removing a current system failure reduces developers' motivation; the long-term reward of better maintainability seems to be less convincing (see Equation 5.4). This impression is based on the conducted interviews during our consultancies and cannot be generalized to all software engineering. However, the interviews were conducted in several different domains and no bias in the selection of the interviewees was applied. We believe, that the results are, hence, representative for the development of long-lived software. On top, the number of internal findings is usually much larger than the number of external findings (see quantitative study in Chapter 4) and, frustrates the developers.

While we cannot reliably predict the benefit of removal for internal findings, we can yet optimize their costs of removal (see Equation 5.2). With low removal costs, it is easier to reduce the inhibition thresholds of developers and motivate them to remove internal findings which does not offer immediate reward. Our cost model in Chapter 5 showed that removing internal findings is less costly when aligned to ongoing development: If a finding is removed in code that has to be changed anyway, overhead in code understanding and testing can be saved—performing a change request already comprises both. To better understand how developers perform changes and how software evolves, we studied the structural software evolution and derived a change-based prioritization from it.

Changes during Software Evolution In Paper **B** [84], we studied the evolution of code structure, i. e. the organization of code in methods and files, to better understand how developers perform changes. The study shows that while files are changed frequently, many methods remain unchanged after their first introduction to the system. Removing internal findings in methods that will never be modified again does not create an expected reduction of non-conformance costs for modifiability (see Equation 5.7). Instead, it limits the expected cost reduction to better readability (a method might still be read for a change even if that change does not affect it directly).

Change-Based Prioritization As a consequence from the study results, we suggest to remove findings in code when it is actually being modified (or newly added). We suggest this for several reasons: First, code being modified has to be understood, changed, and tested anyway. Removing findings in this code creates less overhead than removing findings in unchanged code and corresponds to the boys scout rule—"Always leave the campground cleaner than you found it" [66]. We assume the removal costs as modeled in Equation 5.2 are reduced. While we have no scientific evidence for the concrete savings in removal costs with this change-based prioritization, we yet believe them to be very plausible and based on common-sense.

Second, the prioritization of findings in changed code has a slightly higher estimated benefit, because a changed method is slightly more likely to be changed again in the future than an arbitrary method (see study result in Section 5.5). This increase in the number of subsequent modifications contributes to the benefit of finding removal as depicted earlier in Equation 5.6.

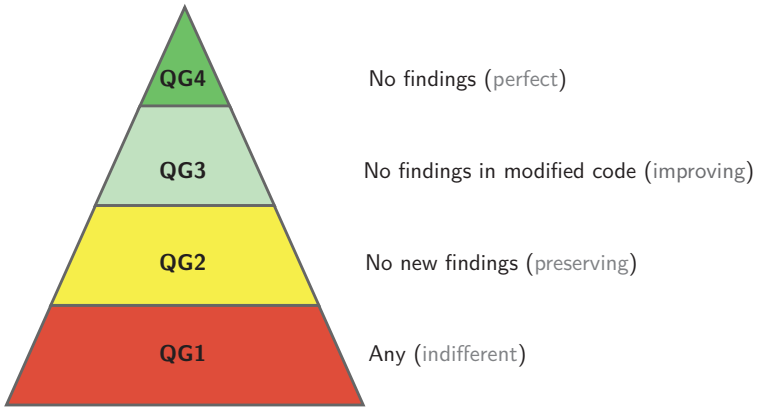


Figure 6.2: Four Quality Goals

Quality Goals To apply the change-based prioritization, our approach introduces the definition of an underlying quality goal to match different system's needs: As systems differ e. g. in their current active development, their criticality, or their expected life span, the quality goal and the derived budget for quality assurance measures differs as well. Based on our industry experience, we believe that the four quality goals *indifferent* (QG1), *preserving* (QG2), *improving* (QG3), and *perfect* (QG4) are sufficient to cover the broad range of different systems' needs.

In our approach, the quality goal has to be defined for each system. If subsystems have different quality assurance budgets, the quality goal can be defined for each subsystem as well and can be adapted over time (see the Process Section 6.5). Based on the quality goal, we define the changed-based prioritization as follows (see also Figure 6.2.3):

QG1 (*indifferent*) Any quality sufficient—No quality monitoring is conducted at all and findings do not have to be removed.

QG2 (*preserving*) No new quality findings—Existing quality findings are tolerated but new quality findings must not be created (or consolidated immediately).

QG3 (*improving*) No quality findings in modified code—A modified method ought to be without findings, leading to continuous quality improvement during further development, similar as the boy scouts rule also used for extreme programming ("Always leave the campground cleaner than you found it“ [66]).

QG4 (*perfect*) No quality findings at all—Quality findings must not exist in the entire system.

To be precise, the concepts of *new* findings or *modified* code introduce the dimension of time to the recommendation process. To define when a finding is *new* or for how long code is considered to be *modified*, we require a time interval. As we apply the finding prioritization *iteratively* into the development process, we obtain iteration intervals. We will address this in more detail in Section 6.4.

QG1 (indifferent) targets running systems for which no maintenance is conducted. Examples include prototypes, migration tools, or systems that will be replaced in the future. Hence, although the new system does not run in production yet, no maintenance (or quality improvement) is performed on the old system to be replaced. QG2 (preserving) and QG3 (improving) are designed for grown systems developed on the brown-field. QG4 is designed for a new system developed on the green field. When quality is monitored continuously from the beginning, a quality goal of no findings is realistically achievable (see Section 6.5).

The idea of those four quality goals itself has been used by the quality consulting company CQSE since several years. Hence, this key concept is not a major contribution of this thesis. Instead, the contribution of this thesis comprises its formalization, evaluation, and integration into an overall finding prioritization approach.

Highly Dynamic System Development In case of very active and highly dynamic development, i. e. frequent modification of the source code, the number of new findings or the number of findings in modified code can still be too large to schedule manual inspection. In this case, the approach recommends only these findings that are very easy to eliminate, i. e. reveal low costs for the refactoring of the code. In our cost model, this corresponds to minimizing the modification costs *c_modification* in Equation 5.2. However, focusing on these findings is only a temporary, less-than-ideal solution to get started. To achieve the long-term quality goal, all findings have to be considered in the recommendation.

Focusing on the easy-to-remove findings has the benefit of providing developers a first starting point for the quality preserving or improving process: In case the change-based recommendation still leads to a very large number of findings, developers are likely to be overwhelmed or even frustrated by the amount of findings in their system. The low-cost-removal recommendation provides them an opportunity to start with the finding removal process in an easy and comfortable, yet useful manner.

In particular, we evaluated the low-cost-removal concept of code clones and long methods in Paper **D**, [86]: The study shows how to detect findings that are very easy to remove, i. e. removable with a fast refactoring. To mitigate the threat that we prioritize findings that are indeed easy to remove but provide no gain for the system's maintainability, we also evaluated whether the developers consider the removal of these findings to be useful. The evaluation showed that–in most cases–developers would in fact perform the refactoring because they considered it to be an improvement for readability or changeability of the code. Hence, the removal of the prioritized findings is not only easy to perform, but also useful to improve maintainability.

6.2.4 Summary

To summarize, Figure 6.3 gives an overview. The figure refines the automatic recommendation process from Figure 6.1 and displays its three steps. Subfigure 6.3a shows the recommendation process from an activity perspective. Subfigure 6.3b shows the same recommendation steps, but from the cost model perspective.

1. External findings recommended for removal with highest priority: regardless of the quality goal, we assume external findings are worth to be manually inspected immediately due to the high benefit of their removal—they reveal a risk of current system failure. We assume removing a potential system failure outweighs the removal costs and has a much higher benefit than removing an internal finding (Equation 5.4 and 5.5). External findings are classified either based on their category (manually) or—in certain cases—based on a single finding occurrence (e. g. with an automatic classifier for inconsistent clones). As we have shown that external findings are usually small in number (Chapter 4) and easy to remove (Chapter 5), this prioritization is feasible.

2. While the benefit for removing internal findings is much harder to predict (see Chapter 5), the prioritization of internal findings aims to minimize removal costs. Internal findings are prioritized in a change-based manner depending on the quality goal: Under the preserving quality goal, the approach recommends removing the new findings. Under the improving quality goal, findings in lately modified code are additionally recommended. In the Figure 6.3a, we refer to findings prioritized based on the quality goal as *QG findings*. The changed-based prioritization reduces the removal costs as modeled in Equation 5.2.

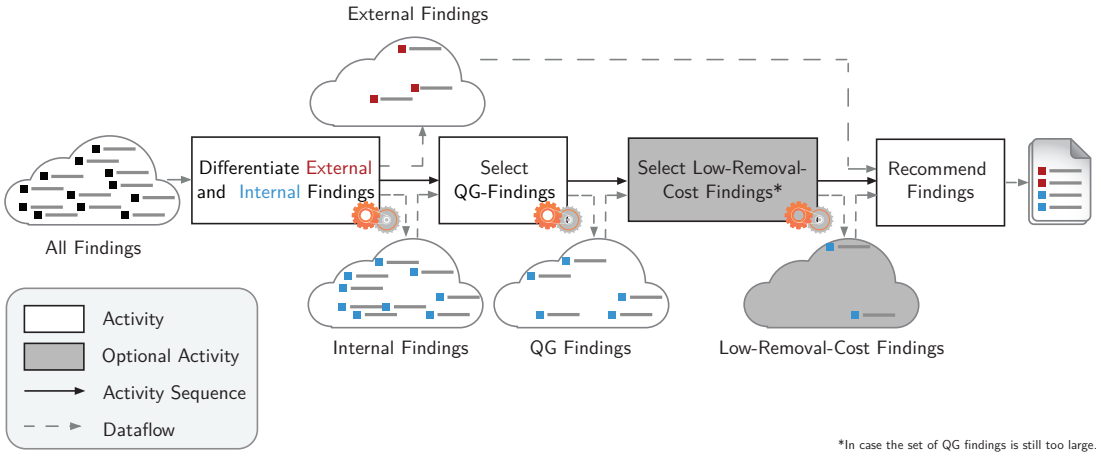
3. For highly dynamic systems with an extensive number of changes to the code, the number of findings in modified code may still be too large to be inspected manually. In this case, the approach classifies in a third step the findings based on the cost of their refactorings and recommends those that are easily removed with a fast refactoring. This minimizes the cost of modification, *c_modification*, as expressed in Equation 5.2.

6.3 Manual Inspection

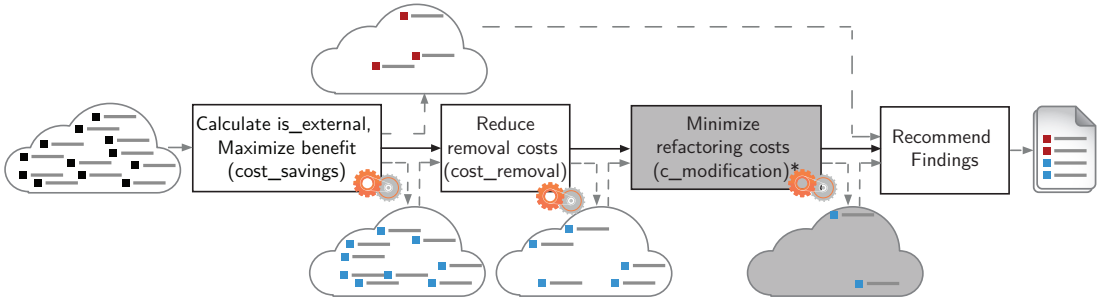
As false positives and non-relevant findings disturb and distract developers, they have to be eliminated from to the set of automatically recommended findings. Therefore, our approach provides a manual component to accept or reject the recommended findings for removal.

For the elimination of the rejected findings, there exists different mechanisms. As we manually inspect the findings individually, we suggest to use *blacklisting* to eliminate single findings. Blacklisting denotes the opportunity that many modern quality analysis tools provide to select a specific finding and remove it from the result to not show it to the developer in the future.

However, sometimes, manual inspection reveals that entire groups of findings have to be blacklisted because they are located, for example, in unmaintained code that should not have been analyzed in the first place. This requires a reconfiguration of the analyses to exclude certain code regions. As our approach is applied iteratively, reconfiguring analyses is possible in-between two iterations. We address this in Section 6.4.



(a) Activity Perspective



(b) Cost Model Perspective

Figure 6.3: Three Steps of the Automatic Recommendation

6.3.1 False Positives

Many automatic static code analyses come along with a certain false positive rate and are, hence, not 100% *precise*. Repeating the example from Section 2.2, a data flow analysis might not be able to parse customized assertions: Developers use customized asserts to ensure that variables are not null. The assert will handle the null-case by, for example, throwing a customized exception. After the assert, the variable can be dereferenced. However, if the data flow analysis cannot parse the assert, it will mark the subsequent dereference as a possible null pointer dereference. This will be a false positive, as it is asserted that the variable is never null.

Even though analysis tools have improved significantly over the last years, false positives probably cannot be eliminated completely. In case of the above example, the data flow analysis could be extended and adapted to parse the customized exceptions. However, as assertion frameworks evolve over time and vary greatly between different development teams, static analyses are likely to be a step behind during the evolution. Hence, the produced false positives require human inspection to be filtered out.

6.3.2 Non-Relevant Findings

In contrast to false positives which are a problem of the analysis, the relevance of a finding depends on external context information: Even though a finding is technically a true positive, it might still not be relevant for removal under certain circumstances (as explained in Section 2.2).

In Paper **D**, [86], we quantified which external context information developers consider when deciding if a finding is relevant enough to be removed—we ran the study exemplary on long methods and code clones. For long methods, developers primarily considered the semantics of the code and its complexity. For code clones, they took the context reasons into account, why certain redundancy was created on purpose.

As much of this context information (e. g. code semantic or historical reasons for redundancy) cannot be automatically measured, a human factor is necessary to decide whether a finding is relevant for removal. Our approach takes this into account as our prioritization is a two-step mechanism with an automated recommendation and a manual acceptance.

6.4 Iterative Application

To take the ongoing development and maintenance of the code base into account and avoid ad-hoc clean up sessions, our approach integrates the automatic recommendation and manual inspection iteratively into the development process (see Figure 6.1). Each iteration aims at prioritizing and removing a certain number of findings to preserve (QG2) or step-wise improve (QG3) the code quality.

6.4.1 Parameters

The iterative application of the proposed prioritization can be adapted to each system with four parameters. All four parameters can be readjusted between iterations, see Section 6.4.3.

Analyzed Code The code being analyzed is the fundamental basis and has to be carefully selected: Different types of code are maintained differently and, hence, have a different impact on the non-conformance costs of a system:

- *Application code* Our approach focuses on the application code: As application code is running in production, its findings are likely to create non-conformance costs: External findings in the application code can cause a system failure and internal findings are likely to hamper future change requests.
- *Test code* The influence of test code on the system's non-conformance is different from the influence of the application code. As test code is not running in production, its external findings, for example, cannot create a system failure. Its internal findings may, however, impact the costs of a change request as modifying application code usually implies also modifying test code. For example, Athanasiou

et al. showed that test code quality is positively correlated with throughput and productivity issue handling metrics [5]. Whether the same quality criteria hold for test code as for application code is frequently debated amongst developers. Recent research as shown that static analyses as quality assurance technique are important for test code as well: Yet, different analyses are suited better [5, 94]. Our approach offers a flexible mechanism to handle test code: For each system, it can be decided individually whether test code can be included. Additionally, on test code different static analyses can be applied than on application code. Yet, the change-based prioritization holds for all of the revealed findings.

- *Generated code* This code should be excluded in our approach: if the quality of generated code is to be assessed, the input of the generator should be analyzed rather than its generated output.

Additionally, both in application and test code (if included to the approach), *unmaintained* code should be identified and also excluded from the analysis—findings in unmaintained code disturb and distract developers unnecessarily.

Analysis Set Before the first iteration, the set of static analyses has to be defined. The set of chosen analyses defines the quality aspects of the system that are addressed (see Chapter 3). In particular, to detect external findings, choosing analyses appropriate for each domain is crucial (see Section 6.2.2).

Iteration Length The length of an iteration and the resulting number of iterations per year depend on the system's current development activity and, hence, on the number of changes made to the code base: Highly dynamic systems with lots of development activities should have shorter iteration periods (e. g. an iteration length of one or two months) than systems with very little development (e. g. with only one or two iterations per year). Further, the iteration length can be adapted to the underlying development model: if the team uses scrum, for example, finding prioritization can be aligned to sprint planning.

Quality Goal The prioritization is based on the fundamental concept of a *quality goal* as explained in Section 6.2. The quality goal has to be chosen for each system before the first iteration. Both QG1 and QG4 require no prioritization as either no (QG1) or all (QG4) findings have to be removed. Hence, the iterative prioritization targets at the preserving and improving quality goal (QG2, QG3).

6.4.2 Prioritization per Iteration

The iterative application starts with running the predefined set of static analyses on the current version of the system, resulting in a set of current findings. At the beginning of the first iteration, all detected findings will be marked as *existing*. During the first iteration, developers are only recommended to follow the underlying quality goal, i. e. consolidate all new findings (preserving) or clean up modified code (improving). As the recommendation

depends on the concept of findings in *new* and *modified* code, the recommendation cannot be performed till the second iteration.

At the beginning of the second (and any further) iteration, the set of static analyses is run again and the prioritization mechanism is applied: External findings are all recommended for inspection, internal findings are recommended based on the quality goal: All findings that were *new*, i. e. introduced within the previous iteration and/or all findings in *modified* code, i. e. code that was modified during the last iteration are recommend for manual inspection. If the set of findings in modified code is still too large to be handled within this iteration, then a classification of low-removal-cost findings is applied (see Section 6.2.3).

6.4.3 Adjusting the Parameters

Inbetween two iterations, the parameters can be readjusted to take different circumstances into account, e. g. a new static analysis, an increased development speed, or a cut in the quality assurance budget.

Analyzed Code In a well structured system that clearly separates application, test, and generated code, and has no unmaintained code, usually the analyzed code base should not have to be readjusted. However, based on our industrial experience, manual inspection does occasionally reveal unmaintained code that developers have forgotten about. This code can be removed from the analyzed code during iterations.

Analysis Set If a new static analysis is available or an existing analysis is discarded by the developers, the set of analyses can be expanded or reduced: In case an analysis is discarded, all its corresponding findings will be neglected in future iterations. In case an analysis is added at the beginning of an iteration, all of its newly detected findings are marked as *existing* for the given iteration. Starting with the subsequent iteration, findings from this analysis can be treated as all other findings—they can be marked as a *new* finding or a finding in *modified* code and recommended accordingly.

Iteration Length Occasionally, development circumstances change and budget for quality assurance is reduced, e. g. when developers leaving the team reduce the development capacity or team transitioning absorbs management attention. The iteration length provides the opportunity to readjust the effort spent on finding removal: For example, when a system undergoes a team transition, the iteration length can be increased during the transition period and decreased after the new team settled down.

Quality Goal The quality goal can be readjusted at system-level: Increasing the quality goal (e. g. from QG2 to QG3) will result in a larger set of prioritized findings, decreasing it will have the opposite effect. It can also be redefined for each system component: For example, a newly developed component can receive QG4 (perfect) while critical components are put under GQ3 and minor components under QG2.

6.5 Quality Control Process

The main contribution of our approach lies in the finding prioritization and its iterative integration. However, based on our industry experience, prioritizing findings for removal alone is not sufficient to lead to long-term quality improvement [85]. Under increasing time pressure, the implementation of new features often dominates quality assurance and static analyses are often omitted. Developers require not just the finding prioritization but also the necessary time and budget to perform refactorings. Hence, additional management awareness is needed. To get developers, quality engineers, and management on board, we describe a continuous control process to achieve the necessary management attention.

6.5.1 Task Management

First, we propose a *task management* to ensure the prioritized findings are scheduled for removal within the development process. For each prioritized finding, a task is created to keep track of its removal status. If finding removal is organized within the same issue tracking system as change requests, chances are smaller than the usage of static analyses is forgotten. Tasks can be easily integrated into planning for agile development (e. g. scrum), but also be useful in non-agile processes (e. g. spiral model).

Figure 6.4 shows the task management combined with the iterative prioritization mechanism: For each recommended and manually accepted finding, a task is created for the developers that contains the finding's location and optionally a possible solution how to remove the finding. The solution can be, for example, a refactoring suggestion in case of a maintainability finding or a bug fix in case of a correctness finding pointing to a programming fault. A dedicated quality manager or the development team itself may, for example, create these suggestions. Before the beginning of the next iteration, the implemented solution for each task is checked: the task is marked either as *solved* or moved to the task backlog to be rescheduled if it was not solved sufficiently.

6.5.2 Management Awareness

Second, to ensure that the tasks for finding removal get enough attention and budget during the daily pressure of new incoming change requests, management awareness is required. Even with prioritized findings and manageable tasks, developers cannot work off the tasks if they are not provided the necessary resources.

To decrease the number of findings in the long term, a continuous quality control process is necessary. The process must combine project managers, quality engineers, and developers. It enhances the proposed findings prioritization and the task management with regular quality reports after each iteration that are forwarded to management. The reports create awareness for the relevance of quality assurance activities at management level as well as transparency—transparency if budget is missing to actually remove findings or if the provided budget actually leads to a decreasing findings trend. More details about the task management and the control process can be found in Section 7.5, containing Paper **E**, [85].

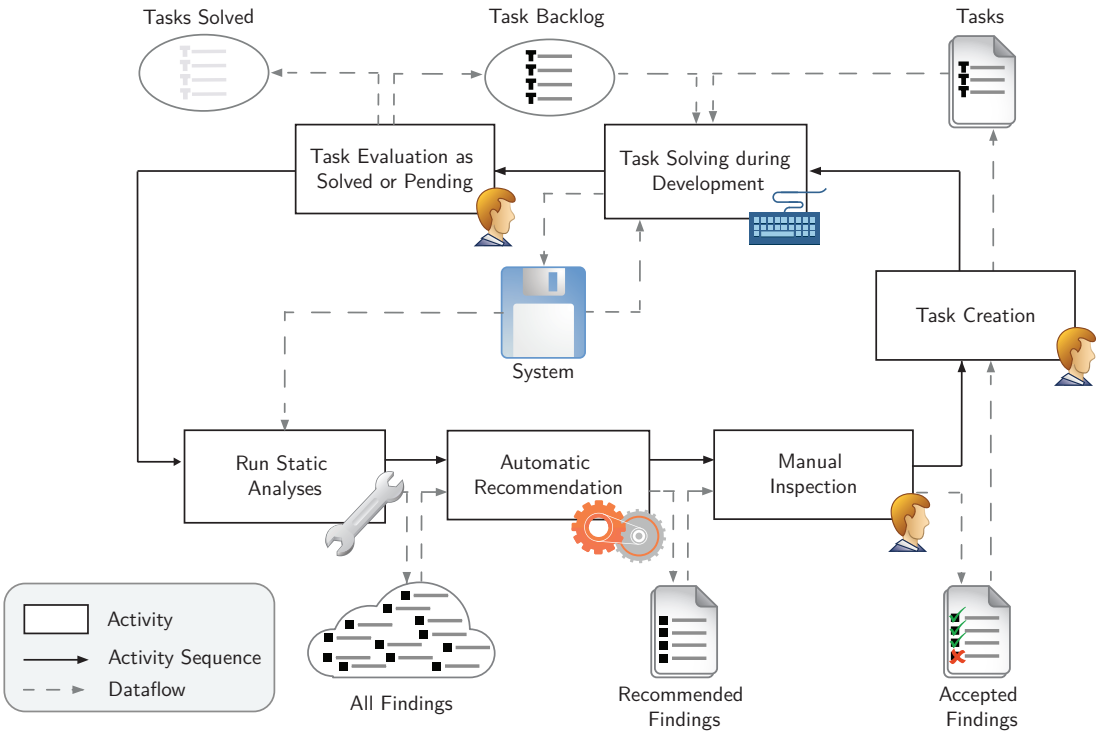


Figure 6.4: Complete Approach with Task Management

6.5.3 Tool Support

Third, to make the process applicable in reality, efficient tool support is crucial. In particular, to apply the prioritization iteratively, a *finding tracking* mechanism is necessary to track findings during evolution: the finding tracking should reliably mark a finding as *new* only if it was newly introduced to the system, not if it changed its location—for example, because a method was extracted, a method was moved to a different file, or a file was renamed or moved. Otherwise, if existing findings are falsely marked as new, they will disturb and distract developers and make them more reluctant to use static analyses routinely.

Many existing tools analyze only a snapshot of a system. Thus, they are not designed to provide a distinction between new and old findings. Teamscale, a tool developed by our company, however, analyzes the full history and, thus, provides the fundamental, time-based data for such a distinction. In Teamscale, we implemented a fundamental technical contribution for a finding tracking algorithm (Section 7.1): In Paper **A** and **B**, we provide an *origin analysis* to track methods and files during evolution. If methods and files can be tracked during refactorings such as moves, renames, and extractions, tracking the findings contained in them becomes easier. It enables Teamscale to accurately differentiate between new and old findings.

Section 7.6 addresses the tool support additionally from the usability side. In Paper **F**, [40] we evaluated in how far the tool Teamscale supports the quality control process and if the users are satisfied with the provided tool support.

6.6 Summary

In this chapter, we presented a prioritization approach based on our cost model (Chapter 5) to effectively use static analyses. In the following, we briefly summarize the approach.

Our approach automatically recommends a subset of findings revealed by static analyses for removal. Hence, it reduces the number of findings to a feasible size. For the recommendation, the approach differentiates between external and internal findings, taking into account that these two kinds of findings have different benefit of removal: As external findings reveal the short-term benefit of addressing a current, potentially user-visible and costly system failure, they are prioritized the highest. As they are usually small in number (Chapter 4) and easy to fix (Chapter 5), their removal is feasible in each iteration.

In contrast, the benefit of removing internal findings is much harder to predict. As future maintenance is usually rather derived from changing requirements, we assume that existing artifacts provide limited input for the automated prediction of future maintenance (Chapter 5). While we cannot reliably automatically quantify the benefit of removal for internal findings, we can yet estimate their costs of removal: With low removal costs, it is easier to reduce the inhibition thresholds of developers and motivate them to remove internal findings that do not reveal immediate reward. Hence, we closely align the prioritization with ongoing development and recommend removing internal findings only in code developers are changing (or adding) anyway. This saves overhead costs of finding removal in terms of code understanding and testing as modifying code due to a change request already comprises both. As our study on software changes showed, this prioritization, in fact, reduces the number of internal findings significantly as many existing methods remain unchanged throughout their life cycle. If the number of internal findings in new code is still too large, our approach focuses initially only on these findings that are very easy to refactor.

After the automated recommendation, the approach contains a manual inspection for all recommended findings. Hence, false positive and non-relevant findings are filtered out. Only the manually accepted findings are prioritized for removal from the system.

As removing findings as a one-off event does not reduce non-conformance costs in the long term, the finding prioritization is applied iteratively. Hence, it closely integrates into the development process and takes into account that new findings can be created. Between two iterations, the prioritization can be adapted with various different parameters to match the project's time and budget planning.

To allocate enough resources for finding removal even when time pressure increases, we integrate the finding prioritization in a quality control process. The process contains a task management for each prioritized finding to reduce the chance that removing findings is omitted. Further, it suggests writing quality reports and regularly send them to management. The reports provide transparency about the usage of the provided quality assurance budget. Finally, the approach also considers adequate tool support to make the process applicable in practice.

"To get to know, to discover, to publish—this is the destiny of a scientist."

Francois Arago

7 Publications

In this chapter, we present the publications included in this thesis. For Papers **A**–**F**, each paper is briefly summarized within one page. The summary shows how each paper contributes to the approach in Chapter 6, see Figure 7.1. Papers **A**–**E** are first-author papers with the first-author claiming full contribution of study designs, implementations, evaluations, and writing of the paper. Paper **F** is a second-author publication with contribution of the study evaluation, the creation of the demonstration video, and most part of the writing, but only minor parts of the implementation.

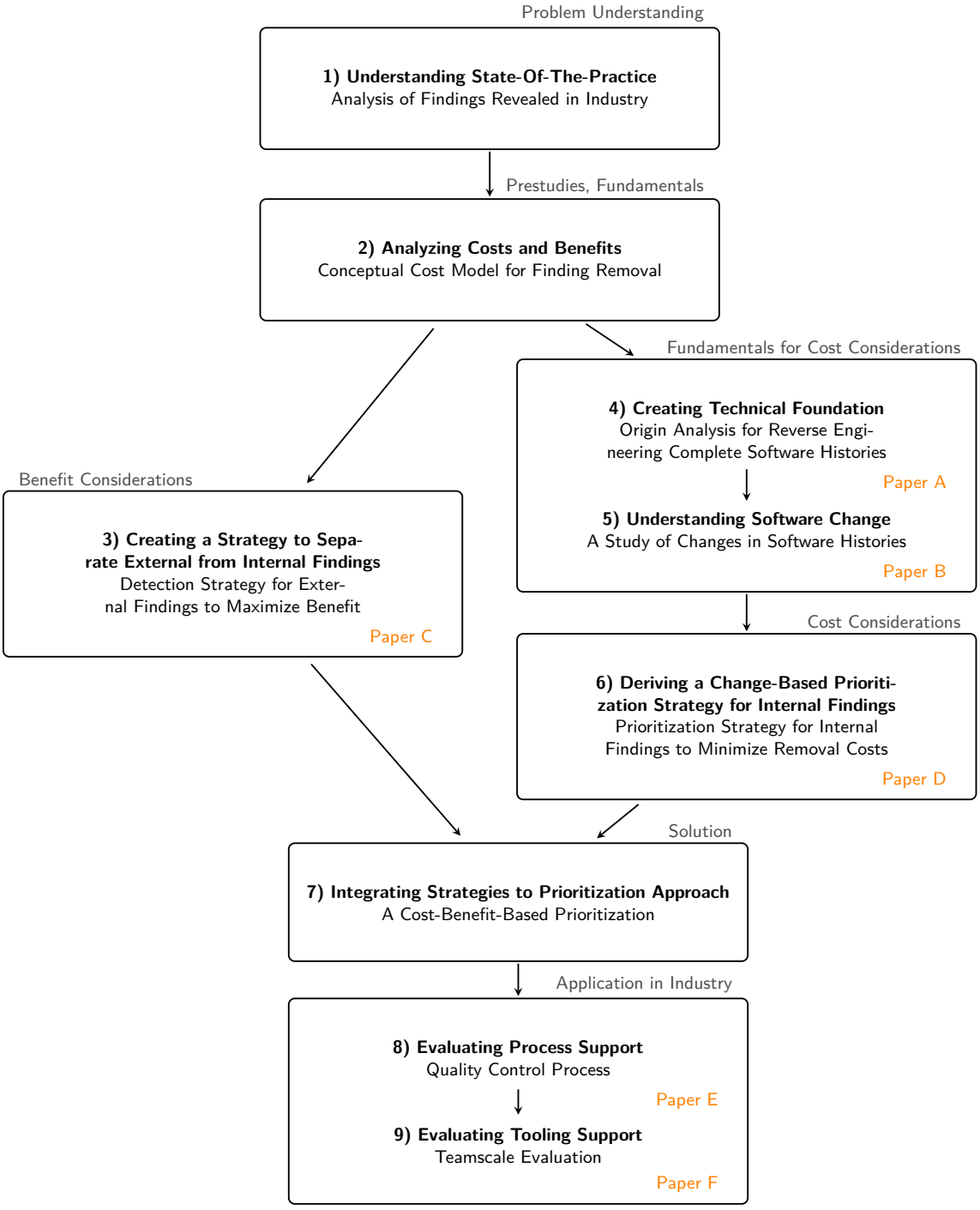


Figure 7.1: Integration of Publications

7.1 Origin Analysis of Software Evolution

The paper "Incremental Origin Analysis of Source Code Files" (Paper **A**, [89]) provides the technical foundation for the evolution case study in Paper **B** and the tool support described in Paper **F**. Even though version control systems have been widely used, the history of software is often not accurately recorded: The paper shows that up to 38% of the files have at least one move in its history that was not recorded in the version control system. However, a complete file history is the fundamental basis for any history analysis: It enables the calculation of accurate quality metric trends and precise findings tracking. With the origin analysis of this paper, we are able to reconstruct the complete history of a source code file, even if the file was refactored, e. g. renamed, copied, moved, or split or if parts of the repository were moved.

Goal The paper aims to provide an incremental, i. e. commit-based origin analysis for source code files to reconstruct the complete history of a file.

Approach For every commit in the system's history, our approach determines for an added file whether it was moved or copied from a file in the previous revision. To detect moves and copies which are not explicitly recorded in the version control system, we provide a clone-based and name-and-location-based heuristic.

Evaluation We evaluate the approach in terms of correctness, completeness, performance, and relevance with a case study among seven open source systems and a developer survey.

Results With a precision of 97-99%, the approach provides accurate information for any evolution-based static code analysis. A recall of 98% for almost all systems indicates that the heuristics can reliably reconstruct any missing information. The case study showed that our approach reveals a significant amount of otherwise missing information. On systems such as ArgoUML for example, 38.9% files had an incomplete history as recorded in the repository. Without our approach, current static code analyses that rely on file mapping during evolution would have made an error on more than every third file. In contrast to many state-of.the-art tools, our approach is able to analyze every single commit even in large histories of ten thousand revisions in feasible runtime.

Thesis Contribution The origin analysis is a fundamental technical basis for this thesis: Its implementation is part of the tool Teamscale that was evaluated in Paper **F**. It enables an accurate finding tracking and, hence, helps developers to accurately differentiate between new and old findings. This differentiation is necessary for the change-based prioritization of our recommendation approach as described in Section 6.2.3. Further the file-based origin analysis was adopted and extended to a method-based origin analysis for Paper **B**.

Incremental Origin Analysis of Source Code Files*

Daniela Steidl
steidl@cqse.eu

Benjamin Hummel
hummel@cqse.eu

Elmar Juergens
juergens@cqse.eu

CQSE GmbH Garching b. München Germany

ABSTRACT

The history of software systems tracked by version control systems is often incomplete because many file movements are not recorded. However, static code analyses that mine the file history, such as change frequency or code churn, produce precise results only if the complete history of a source code file is available. In this paper, we show that up to 38.9% of the files in open source systems have an incomplete history, and we propose an incremental, commit-based approach to reconstruct the history based on clone information and name similarity. With this approach, the history of a file can be reconstructed across repository boundaries and thus provides accurate information for any source code analysis. We evaluate the approach in terms of correctness, completeness, performance, and relevance with a case study among seven open source systems and a developer survey.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics

General Terms

Algorithms

Keywords

Software Evolution, Origin Analysis, Clone Detection

1. INTRODUCTION

Software development relies on version control systems (VCS) to track modifications of the system over the time. During development, the system history provides the opportunity to inspect changes in a file, find previous authors, or recover deleted code. For source code analyses, such as quality or maintenance assessments, the history of a system makes it possible to evaluate the software evolution and to discover metric trends. Many analyses depend on the complete history of source code files. For example, the clone community evaluates the impact of clones on maintenance by measuring the stability of cloned code in terms of age since the last change [21] or as change frequency [8, 20]. The defect prediction community also relies on history information: Prominent approaches use code churn, code stability, change frequency, or the number of authors per file to predict software bugs [15, 23, 24]. Other analyses such as type change analysis, signature change analysis, instability analysis [16], or clone genealogy extractors [17] also rely on accurate program element mapping between two versions.

Unfortunately, the source code history of a file is often not complete because movements or copies of the file are not recorded in the repository. We refer to those moves as *implicit* moves in contrast to recorded, *explicit* moves. Lavoie [22] has shown that a significant portion of file movements are implicit. This is due to several different reasons: When moving a file, programmers might not use the commands as provided by VCSs such as subversion (SVN), *e. g.*, *svn move*. Sophisticated development environments such as Eclipse offer the refactoring *rename* which automatically records the rename as a move in the repository. When developers rename the file manually, though, or copy and paste it, the move/copy will not be recorded in the VCS. Furthermore, when projects are migrated from one type of VCS to another, move/copy information will be lost. For example, many open source projects have been previously managed with CVS (where no origin information was stored at all) and were later migrated to SVN, where the moves and copies in the history of CVS time periods remain missing. On top, if parts of a repository are moved across repository boundaries, this event is not recorded in the history and cannot be captured by any VCS. With our approach, we will show that up to 38.9% of the files in open source systems have an incomplete history.

Problem Statement. *The history of source code files as recorded in VCSs is often incomplete and provides inaccurate information for history-based static code analyses.*

Reconstructing a complete source code history requires an *origin analysis* for each file. Reverse engineering the history of a file is not trivial: If a file is moved or renamed, it may also be modified at the same time. Hence, a simple comparison of whether the file's content remains the same is not sufficient. Current approaches that perform any kind of origin analysis, either on file level [7, 22] or function level [9, 10] are release-

*This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597111

based and are not run incrementally. All published case studies include only few releases of a software system as snap-shots and compare them to each other. Release-based origin analysis can not reverse-engineer the complete history in feasible time, and thus provides only a coarse picture. Our approach, by contrast, works incrementally and can analyze every commit, even in histories with ten thousands of commits. Hence, our approach provides a fast and fine-grained origin analysis.

For every commit in the system's history, our approach determines for an added file whether it was moved or copied from a file in the previous revision. We extract repository information to detect explicit moves and use heuristics based on cloning, naming, and location information to detect implicit moves. We evaluate the approach in terms of correctness, completeness, performance, and relevance with a case study with seven open source systems and a survey among developers. Although the approach is generally independent of the underlying version control system, we evaluated it only on SVN-based systems.

Contribution. *In this paper, we present an incremental, commit-based approach using repository information as well as clone, naming, and location information to reconstruct the complete history of a file.*

2. RELATED WORK

We group related work to origin analyses by file-based, function-based, and line-based origin analyses and code provenance. [16] gives an overview of other program element matching techniques in the current state of the art.

2.1 File-based Origin Analysis

The work by Lavoie et al., [22], is most similar to ours as the authors also focus on inferring the history of repository file modifications. They reverse engineer moves of files between released versions of a system using nearest-neighbor clone detection. They evaluate their approach on three open source systems Tomcat, JHotDraw, and Adempiere. The evaluation compares the moves detected by their approach with commit information extracted from the repository. Our work differs in three major aspects from the work by Lavoie: First, we also detect file copies, besides moves. Lavoie et al. do not consider copied files, hence, our approach provides more information to recover a complete history. Second, Lavoie et al. reconstruct file movements only between released versions of a system in feasible time. With our approach, we analyze every single commit in the history and find moves at commit level still with feasible runtime. Hence, we provide more details for developers: When a developer accesses the history of a file, *e. g.*, to view changes, review status, or commit authors, he needs the complete history on commit level and not just between released versions. Third, Lavoie et al. compare their results only against the repository oracle, although their approach reveals a significant amount of *implicit moves*. As the authors state, they were not able to thoroughly evaluate these implicit moves except for a plausibility check of a small sample performed by the authors themselves. From this point of view, our work is an extension of Lavoie's work: In addition to an evaluation based on the repository information, we conduct a survey amongst developers to evaluate implicit moves. Hence, we provide reliable information about the overall precision of our approach.

The work of [2] automatically identifies class-level refactorings based on vector space co- sine similarity on class identifiers. The approach aimed to identify cases of class replacement, split, merge, as well as factoring in and out of features. Based on a case study with only one system, the approach does not always identify a unique evolution path per class. Instead, the authors manually inspected the results. Our work, in contrast, uses clone detection as suggested by the authors in their future work section, to obtain unique results which we evaluated in a large case study based on repository and developer information.

Git [1] provides an internal heuristic mechanism to track renames and copies based on file similarity. To our best knowledge, there exists no scientific evaluation on Git's heuristic to which we could compare our approach. In contrast, many users report on web forums such as stackoverflow.com that Git fails to detect renames[1] or becomes very slow when the option to also detect copies is set.

Kpodjedo [19] addresses a similar problem with a different approach. Using snapshots of the Mozilla project, the authors reverse engineer a class diagram for each revision and apply the ECGM algorithm (Error Correcting Graph Matching) to track classes through the evolution of the software. They do not aim to find the origin for each file though, but to discover a stable core of classes that have existed from the very first snapshot with no or little distortion.

The work of [7] detects refactorings such as splitting a class and moving parts of it to the super- or subclass, merging functionality from a class and its super- or subclass, moving functionality from a class to another class, or splitting methods. The refactorings are detected by changes in metrics of the affected code – this relates to our origin analysis as they also track movements of code. However, the authors admit that their approach is vulnerable to renaming: "Measuring changes on a piece of code requires that one is capable of identifying the same piece of code in a different version." As the authors use only "names to anchor pieces of code", they cannot track metric changes in the same source code file if the file name changed, leading to a large number of false positives in their case study. Hence, our approach constitutes a solution to one of the drawbacks of the work by [7] by providing accurate renaming information.

On a very loose end, the work of [14] also relates to our work: it proposes a new merging algorithm for source code documents and includes a rename detector, which can detect renames of a file - which is also part of our origin analysis. However, the authors use their rename detector in quite a different context: They present a new merging algorithm that merges two different versions of the same file and can detect when one version is basically the same as the other except for a syntactical rename. Due to the context of merging two files, the input to their algorithm is quite different to the input of our algorithm.

2.2 Function-based Origin Analysis

Several papers have been published to track the origin of functions instead of files (classes). One could argue that aggregating the result of function tracking can solve the problem of file tracking. However, we disagree for the several reasons. First, there has been no scientific evaluation of how

[1] `http://stackoverflow.com/questions/14527257/git-doesnt-recognize-renamed-and-modified-package-file`, last access 2013-12-03

many functions a file may differ over and still be considered the same file. Our approach is also based on thresholding, but we present a thorough evaluation that confirms the choice of thresholds. Second, even if the thresholds for aggregating function tracking were evaluated, our origin analysis would produce more flexible results: Our origin analysis is based on clone detection in flexible blocks of source code such that we can detect that a file is the same even if code was extracted into new methods, added to a method, or switched between methods. Third, relying on function tracking would make the analysis dependent on languages that contain functions. Our approach, by contrast, is language-independent and can be also applied to function-less code artefacts such as HTML, CSS, or XML.

Godfrey et al. coined the term *origin analysis* in [9,10,25]. They detect merging and splitting of functions based on a call relation analysis as well as several attributes of the function entities themselves. Automatically detecting merging and splitting on file level remained an area of active research for the authors in [10] as this is currently done only manually in their tool. As merging files is part of our move detection, our work is an extension of their work.

In [18], the authors propose a function mapping across revisions which is robust to renames of functions. Their algorithm is based on function similarity based on naming information, incoming and outgoing calls, signature, lines of code, complexity, and cloning information from CCFinder. The authors conclude that dominant similarity factors include the function name, the text diff and the outgoing call set. In contrast, complexity metrics and cloning information (CCFinder) were insignificant. Our work also relies on naming information, however, for our work also the cloning information provided a significant information gain.

Rysselberghe [26] presents an approach to detect method moves during evolution using clone detection. However, in contrast to our work, as the authors use an exact line matching technique for clone detection, their approach cannot handle method moves with identifier renaming.

2.3 Line-based Origin Analysis

Quite a few papers track the origin of a single code line [3,5] with a similar goal–to provide accurate history information for metric calculations on the software's evolution. However, identifying changed source code lines cannot provide information for the more general problem of file origin analysis for two reasons. First, it has not been evaluated yet how many source code lines in a file can change such that the file remains the "same" file. Second, tracking single lines of code excludes a lot of context information. Many code lines are similar by chance, hence, aggregating these results to file level creates unreliable results.

2.4 Code Provenance

Code provenance and bertillonage [6,11] constitute another form of origin analysis: It also determines where a code artifact stems from, but either for copyright reasons (such as illegally contained open-source code in commercial software) or for detection of out-dated libraries. Code provenance differs from our origin analysis as we detect the origin of a file only within the current software containing the file whereas code provenance considers a multiple project scope for origin detection.

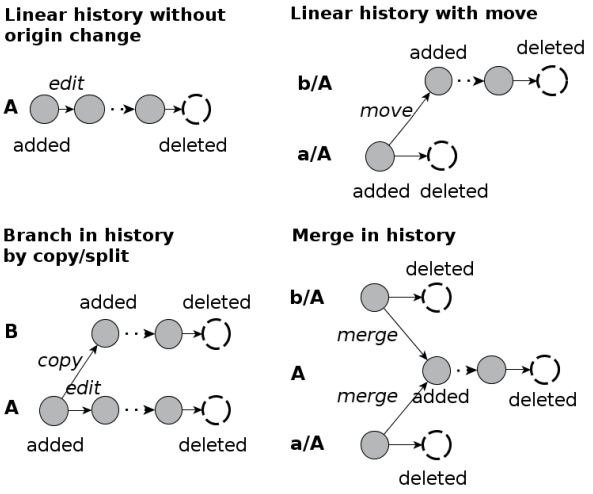


Figure 1: Metamodel of the history of a file.

3. TERMINOLOGY

We use the following terms to model the history of a source code file: In the simplest case, the history of a file is *linear* (Figure 1); a file is truly added – in the sense that the file is really new and had no predecessor in the system – with a specific name (*e. g.*, A.java or A.c) at a specific location (*e. g.*, a path) in revision n. Then the file may be edited in some revisions, and finally deleted (or kept until the head revision of the repository). The other cases which are more complex are the target of our detection (Figure 1):

- *Move*: A file is moved if it changes its location (its path) in the system. We also consider a rename of the file name as a move of the file. Repositories represent a move typically as a file that existed in revision n at some path, got deleted at this location in revision $n+1$, and was added at a different location and/or with a different name in revision $n+1$.
- *Copy*: A file is copied if the file existed in revision n and $n+1$ at some location (*e. g.*, a path), and was added at a second location in revision $n+1$ without being deleted at the original location. A copy may include a name change for the new file. In that case, a copy might also be a split of a file into two files.
- *Merge*: A file got merged if multiple files were unified into one single file. The merged file got added in revision $n+1$ and multiple files from revision n are deleted in revision $n+1$. As VCSs such as SVN cannot capture merges at all, repository information about them is missing.

A moved file still has a linear, unique history (no branch or split, and a unique predecessor and successor for each revision except those of the initial add and the final delete). A copied file can have two sucessors but both the original and the copied file have a unique predecessor. In case of a merge, the predecessors of the merged file are not unique.

The detection of moves and copies is conceptually the same in our approach, as they differ only insofar as the original file is deleted during a move, whereas it is retained during a copy. Hence, we will not differentiate between the two

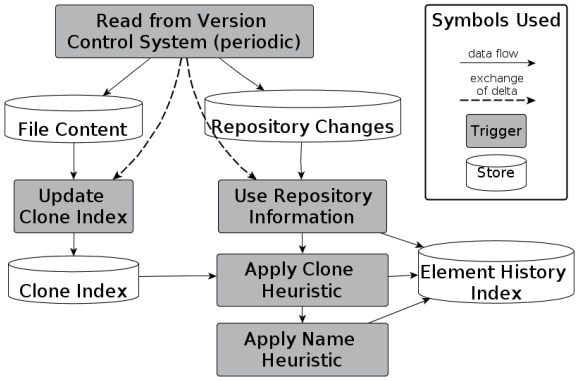


Figure 2: Design of a the approach.

in the remainder of this paper and use the term *move* as generalization for both copy and move.

Although we use the terms *branch* and *merge* in this paper, they do not have the same meaning as *branching* and *merging* in the SVN language: For SVN, a *branch* is used to create multiple development lines for experimental purposes or different customer versions of the same product. A *merge* is used afterwards to unify changes on a branch and the main trunk. In our case, however, we limit our analysis to the trunk of the repository only. Still, merges as described above can occur, where both the two original files are part of the trunk and were consolidated into one file.

4. APPROACH

In this section, we present the design and implementation details of our approach to recover the history of a source code file. For each commit, the approach determines whether an added file had a predecessor in the previous revision.

4.1 Underlying Framework

The approach is implemented within the code-analysis framework Teamscale [4, 12] which performs code analyses incrementally and distributable for large-scale software systems. It allows to configure analyses that are triggered by each commit in the history of a system – the analyses then process the added, changed, and deleted files per commit and update the results incrementally. The framework provides repository connectors to many common VCSs such as SVN, Git, or TFS and an incremental clone detector [13].

4.2 Overview

The approach detects explicit and implicit moves, copies, and merges of a source code file using different sources of information - the repository information (when move information is explicitly recorded) and information from heuristics (when explicit information is missing). We suggest two different heuristics, a *clone* heuristic and a *name-based* heuristic. The clone heuristic uses clone detection as a measure of content similarity and aims to detect renames. We do not use simpler metrics such as longest common sequence algorithm to find content similarity because we also want to detect files as moves even if their code was restructured. The name-based heuristic uses primarily information about the name and path, but also includes the content similarity based on clone detection to avoid false positives.The name-based heuristic can only detect moves without rename.

The heuristics are designed to complement each other. The heuristics and the repository information can be applied sequentially in any order. All three sources of information are described in more detail in Sections 4.3–4.5. For now, we choose an exemplary sequence of using the repository information first – as this is the most reliable one – and then, the clone heuristic, followed by the name heuristic.

In this setup, the approach processes the set of added files for each revision n in the history, as follows:

1. Extract information about explicit moves, *i. e.*, moves that were recorded by the VCS. Mark explicitly moved files as detected and remove them from the set.

2. For the remaining set, apply the clone detection to find files at revision $n-1$ that have a very similar content. Mark the detected files and remove them from the set.

3. For the remaining set, apply the name-based heuristic. Mark the detected files and remove them from the set.

We do not know in advance whether it is best to use only one heuristic, or to combine them both (either *name* before *clone* or *clone* before *name*). Figure 2 visualizes the exemplary sequential approach as described above. We will evaluate the best ordering in Section 5.

4.3 Explicit Move Information

Partly, information about a file's origin are already explicitly recorded in the VCS. VCSs such as SVN, for example, provide commands for developers such as *svn move*, *svn copy*, or *svn rename*. Using these commands for a move of a file f from path p to path p', SVN stores the information that p is the origin for f when added at location p'. Sophisticated development environments such as Eclipse offer rename refactorings or move commands that automatically record the origin information in the repository. The recorded information can then be extracted by our framework and stored as a detected move or copy.

4.4 Clone Heuristic

The clone heuristic processes all added files $f \in F$ of revision n in its input set as follows: Using incremental clone detection, it determines the file from revision $n-1$ which is the most similar to f. We use a *clone index* to retrieve possible origin candidates $f' \in F'$ and a *clone similarity* metric to determine if f and any f' are similar enough to be detected as a move. This metric is a version of the standard clone coverage.

Incremental Clone Index. We use parts of the token-based, incremental clone detection as described in [13]. However, any other incremental clone detector could be used as well. The clone detector splits each file into *chunks* which are sequences of l normalized tokens. The *chunk length* l is a parameter of the clone detection as well as the *normalization*. The chunks of all files are stored together with their MD5 hash value as a key in a *clone index* which is updated incrementally for each commit. With the MD5 hash value, we identify chunks in file f that were cloned from other files f' which are, hence, considered to be the set F' of possible origin candidates. With the information about cloned chunks,

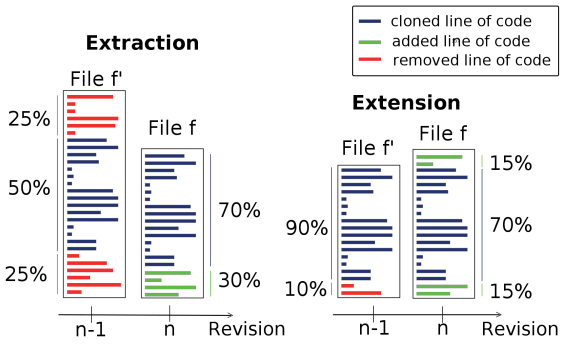


Figure 3: Examples of a detected move/copy of file f' to f with thresholds $\theta_1 = 0.7$ and $\theta_2 = 0.5$.

we define the clone similarity metric. (More details about the clone index and how to use it to retrieve clones are provided in [13].)

Calculating Clone Similarity. The *clone similarity* metric measures the similarity between two files: f is similar to f', if the number of chunks in file f that also occur in f' (based on the MD5 hash) relative to the total number of chunks in f is higher than a certain threshold θ_1. Intuitively, this corresponds to: f is similar to f' if at least a certain amount of code in f was cloned from f'. We pose a second constraint that also a certain amount θ_2 of code in f' still needs to be in f. Figure 3 shows two examples f' and f that are similar based on $\theta_1 = 0.7$ and $\theta_2 = 0.5$.

If we had used only threshold θ_1, then we would have, *e. g.*, detected a very large file f' as origin for a very small file f, if most of the content in f stems from f'. However, there are potentially many large files f' that could be the predecessor for the small piece of code in f, hence, the origin analysis is error-prone to false positives. Consequently, we use two thresholds to ensure that also a certain amount of code from f' is still in f.

Intuitively, one might set $\theta_1 = \theta_2$. This would allow to detect only moves/copies including extensions of the origin file – when the new file as additional code compared to its origin (Figure 3 right example). However, we also would like to detect extractions, *i. e.*, a part of f' is extracted into a new file f (Figure 3, left example). We observed those extractions frequently during preliminary experiments in practice and assume that they represent refactorings of existing code, when a new class is extracted or code is moved to a utility class. To also detect extractions, we set θ_2 smaller than θ_1. Setting θ_2 smaller than θ_1 still detects file extensions.

Move Detection. For the move detection, we calculate the similarity of each added file f and all files $f' \in F'$ at revision $n-1$ that have common chunks with f. The incremental clone index allows to retrieve these files efficiently. We mark the file f' with the highest clone similarity to f as origin.

Parameters. We selected the following parameters:

Chunk length: We set the chunk length to 25 after preliminary experiments showing that setting the chunk length is a trade-off between performance and recall of the clone heuristic: The smaller the chunk length, *i. e.*, the less tokens a chunk consists of, the higher the chance that there are hashing collisions in the clone index because shorter normalized token sequences appear more often in big systems. This makes the calculation of the clone similarity more expensive as it queries the index how many chunks from file f also appear in other files f'. On the other side, it holds that the larger the chunk size, the smaller the recall: A file consisting of few tokens, *e. g.*, an interface with only two method declarations, does not form a chunk of the required length and will not appear in the clone index. As no clone similarity can be calculated, it cannot be detected as move.

Normalization: We chose a conservative token-based normalization, which does not normalize identifier, type keywords, or string literals, but normalizes comments, delimiters, boolean-,character-, number literals and visibility modifier. We chose a conservative normalization as we are interested in content similarity rather than type 3 clones.

Thresholds After preliminary experiments on the case study objects (Section 5), we set $\theta_1 = 0.7$ and $\theta_2 = 0.6$. The evaluation will show that those thresholds yield very good results independent from the underlying system.

4.5 Name-based Heuristic

As a second heuristic, we use a name-based approach: To detect for each added file $f \in F$, whether it was moved, the name-based heuristic performs the following two actions:

1. It extracts all files $f' \in F'$ from the system that have the same name and sorts them according to their path similarity with f. (We use the term *name* for the file name, *e. g.*, `A.java` and *path* for the file location including its name, *e. g.*, `src/a/b/A.java`.)

2. It chooses $f' \in F'$ with the most similar path and the most similar content using cloning information.

Sorting the paths. The heuristic extracts all files $f' \in F'$ from the previous revision that have the same name and are therefore considered as origin candidates for moves. The origin candidates are sorted based on path similarity to f. For example when f has the path `/src/a/b/A.java` and the two candidates are f' =`/src/a/c/A.java` and f'' =`/src/x/y/A.java`, then f' will be considered more similar. Similarity between two paths is calculated with the help of a diff algorithm: With f' and f'' being compared to f, all three paths are split into their folders, using `/` as separator. The diff algorithm returns in how many folders f' and f'' differ from f. Path f' is more similar to f than f'' if it differs in less folders from f. If f' and f'' differ in the same number from f, the sorting is determined randomly.

Comparing the content. The heuristics iterates over all origin candidates, sorted according to descending similarity. Starting with f' with the most similar path to f, it compares the content of f' and f: It calculates the *clone similarity* between f' and f, which is the same metric as used in Section 4.4. To determine whether f' and f are similar enough to be detected as move/copy we use thresholds $\theta_1 = 0.5$ and $\theta_2 = 0.5$. We determined both thresholds with preliminary experiments. In comparison to the *clone heuristic*, we use lower thresholds as we have the additional information that the name is the same and the paths are similar.

For this heuristic, we use a separate, in-memory clone index in which we only insert file f and the current candidate f' of

the iteration. Hence, we can choose a very small *chunk size* without loosing performance drastically. After preliminary experiments, we set the chunk size to 5. As discussed in Section 4.4, the clone heuristic cannot detect moves and copies of very small files, *e. g.*, interfaces or enums, as the chunk size needs to be large to ensure performance. The name heuristic addresses this problem by using the name information to reduce the possible origin candidates and applies the more precise clone detection with small chunk size only on very few candidates. Hence, this heuristic can detect those small files that are missed by the clone heuristic.

5. EVALUATION

The evaluation analyzes four aspects of the approach which will be addressed by one research question each: Completeness, correctness, performance, and relevance in practice. Section 5.1 shows the research questions which are answered with one case study each (5.4-5.7).

5.1 Research Questions

RQ1 (Completeness): What's the recall of the heuristic approach? We evaluate if the heuristics retrieve all moves in the history of a software system. If the heuristics succeed to reconstruct the complete history of a source code file, they can provide accurate information for all static code analyses mining software histories.

RQ2 (Correctness): What is the precision of the approach? We determine how many false positives the approach produces, *i. e.*, how many files were determined as move although being newly added to the system. To provide a correct foundation for evolution mining static code analyses, the number of false positives should be very low.

RQ3 (Relevance): How many implicit moves does the approach detect? We evaluate how much history information our approach reconstructs. This indicates how many errors are made when applying static code analyses of software repositories without our approach.

RQ4 (Performance): What is the computation time of the algorithm? We measure how fast our approach analyzes large repositories with thousands of commits.

Ordering of the heuristics. We also evaluate whether it is best to use only the clone or only the name heuristic, or both in either ordering. The ordering influences correctness and completeness of the approach. As the heuristics are applied sequentially, the second heuristic processes only those files that were not detected as moves by the first heuristic. Hence, the first heuristic should have the highest precision because the second heuristic cannot correct an error of the first heuristic. The recall of the first heuristic is not the major decision criteria for the ordering as a low recall of the first heuristic can be compensated by the second heuristic.

5.2 Setup

For evaluation, we change the design of our algorithm (Figure 2) and remove the sequential execution of extracting explicit move information and applying the two heuristics. Instead, we execute all three in parallel. Figure 4 visualizes the evaluation setup. This gives us the opportunity to evaluate the results of both heuristics separately and use the information extracted from the repository as gold standard.

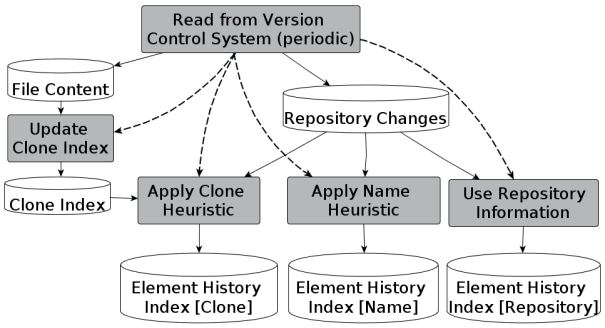


Figure 4: Setup for Evaluation.

5.3 Study Objects

Research questions 1-4 require different study objects. To evaluate recall and precision, we need systems with a large number of explicit moves, providing a solid gold standard. To show the relevance of our approach, we need systems with a large number of implicit moves. To select suitable case study objects, we run our setup on the systems shown in Table 1. All study objects use SVN as repository. Amonst others, the table shows the number of commits which is usually smaller than the difference between end and start revision as, in big repositories with multiple projects, not all commits affect the specific project under evaluation. Running our analysis first reveals how many explicit and implicit moves each repository has. The number of implicit moves is the result of our heuristics, which might be error-prone, but still sufficient to select the study objects. Table 2 shows that ConQAT, ArgoUML, Subclipse, and Autofocus have the largest numbers of explicit moves. ArgoUML and Autofocus also have many implicit moves.

ConQAT has a large number of explicit and a small number of implicit moves, as all developers develop in Eclipse and all major moves within the repository were done with the *svn move* command. ArgoUML has a high number of explicit moves. However, the significant number of implicit moves results from a CVS to SVN migration. Most implicit moves are dated prior to revision 11199, the beginning of the migration.[2] Although Autofocus 3 is primarily developed in Eclipse, there is still a significant number of implicit moves. Our manual inspection reveals that some major moves of packages (such as restructuring the `mira/ui/editor` package or moves from the `generator` to the `common` package) were not recorded. For Subclipse, only a small number of implicit but a large number of explicit moves were found, matching our initial expectation that Subclipse developers pay attention to maintaining their repository history as they develop a tool with the purpose of repository integration into Eclipse. Vuze (former Azureus) is also a system that was migrated from CVS to SVN.[3] Besides many implicit moves due to the migration, there are not many explicit moves in the new SVN. For all other systems, we do not have additional information about their history.

[2] An ArgoUML contributor states that this commit was done by the cvs2svn job and he himself did a manual cleanup in rev. 11242.

[3] http://wiki.vuze.com/w/Azureus_CVS

Table 1: Case Study Objects

Name	Language	Domain	History [Years]	Size [LOC]	Revision Range	# Commits
ArgoUML	Java	UML modeling tool	25	370k	2-19910	11721
Autofocus 3	Java	Software Development Tool	3	787k	18-7142	4487
ConQAT	Java	Source Code Analysis Tool	3	402k	31999-45456	9309
jabRef	Java	Bibliography reference manager	8	130k	10-3681	1521
jHotDraw7	Java	Java graphics GUI framework	7	137k	270-783	435
Subclipse	Java	Eclipse Plugin for Subversion	5	147k	4-5735	2317
Vuze (azureus)	Java	P2P file sharing client	10	888k	43-28702	20762

Table 3: Recall

System	Clone	Name	Clone-Name	Name-Clone
ConQAT	0.84	0.96	0.92	0.98
ArgoUML	0.89	0.97	0.97	0.98
Subclipse	0.91	0.98	0.99	0.99
Autofocus	0.73	0.82	0.85	0.86

Table 2: Explicit and Implicit Repository Moves

Name	Explicit	Implicit
ConQAT	8635	191
ArgoUML	3571	1358
Subclipse	573	38
Autofocus 3	3526	1135
JabRef	6	34
JHotDraw 7	298	175
Vuze	73	860

5.4 Completeness (RQ1)

5.4.1 Design

To evaluate the completeness of the heuristics, we use the recall metric. We calculate the recall based on the information extracted from the repository which gives us only an approximation of the true recall: Among all explicit moves, we calculate how many are found by our heuristics. However, the heuristics' true recall over a long history cannot be calculated as there is no source of information about how many files were moved. Even asking developers would not provide a complete picture if the history comprises several years. The experimental setup (Section 5.2) allows us to run the heuristics in parallel to extract the repository information. We chose those open source systems from Table 1 that contain the largest numbers of explicit moves, *i. e.*, ConQAT, ArgoUML, Subclipse and Autofocus. With their explicit move information, those systems create a solid gold standard for evaluation.

5.4.2 Results

Table 3 shows the recall for all possible orderings of the two heuristics (as explained in Subsection 5.1). The sequential execution of the name heuristic before the clone heuristic (annotated in the column header with *Name-Clone*) yields the most promising results, with a recall of above 98% for ConQAT, ArgoUML, and Subclipse. Only for the study object Autofocus, the recall is slightly lower with 86%. This is due to one limitation of the current approach: its failure to detect reverts in the repository. The approach can detect moves or copies only if the origin is present in the previous revision. If some part of the system was deleted and later readded (reverted), our approach cannot detect this. As Autofocus contains some reverts, the recall is slightly lower. The name-heuristic itself already achieves a recall of above 82% for all systems, indicating that many moves are done without renaming.

5.5 Correctness (RQ2)

5.5.1 Design

The correctness of the approach is evaluated with the precision metric. The precision of the approach cannot be evaluated solely based on explicit move information as the repository does not provide any information whether implicit moves detected by a heuristic are true or false positives. Hence, we evaluate the precision two-fold: First, we calculate a lower-bound for the precision based on the explicit move information assuming that all implicit moves are false positives. This assumption is certainly not true, but it allows us to automatically calculate a lower bound and make a first claim. The calculation is done on the same study objects as in the completeness case study because it requires a large number of explicit moves. Second, in a survey, we gather developer ratings from ConQAT and Autofocus developers for a large sample of implicit moves and use them to approximate the true precision of the approach.

Challenges in Evaluating the Precision. Although the precision is mathematically well defined as the number of true positives divided by the sum of true and false positives, defining the precision of our approach is not trivial: In case of a merge as shown in Figure 1, the origin predecessor of an added file is not unique. However, our heuristics are designed such that they will output only one predecessor. Hence, in case of a merge, it is not decidable which is the *correct* predecessor. In the absence of explicit move information or a disagreement between the two heuristics, there is no automatic way to decide which result is a true or false positive. For calculating the lower bound of the precision, we assumed the disagreement to represent two false positives.

Developer survey. We chose the two study objects ConQAT and Autofocus because their developers were available for a a survey. Table 4 shows the participation numbers as well as the average years of programming experience and the average years of programming for the specific project. As the participants had many years of programming experience and were core developers of the respective system under evaluation, we consider them to be suitable for evaluation.

Table 4: Experience of Survey Subjects [years]

System	Developers	∅ Program. Experience	∅ Project Experience
ConQAT	6	13.8	5.9
Autofocus	3	17	4.3

Table 5: Lower Bound for Precision

System	Clone	Name	Clone-Name	Name-Clone
ConQAT	0.91	0.98	0.92	0.97
ArgoUML	0.73	0.76	0.71	0.72
Subclipse	0.92	0.98	0.99	0.99
Autofocus	0.75	0.74	0.72	0.72

Table 6: Precision based on Developers' Rating

System	Clone	Name	Clone-Name	Name-Clone
ConQAT	0.975	0.925	0.95	0.95
Autofocus	1.0	0.90	0.90	0.90

Table 7: Statistical Approximation of Precision

System	Clone	Name	Clone-Name	Name-Clone
ConQAT	0.997	0.998	0.996	0.998
Autofocus	1.0	0.97	0.97	0.97

We showed the developers a statistic sample of implicit moves and moves for which a heuristic led to a different result than recorded in the repository. We randomly sampled the moves such that the developers evaluated n moves detected by the clone heuristic, n by the name heuristic and n by each of the sequential executions. However, as those four sample sets intersect, the developers rated less than $4n$ samples. For ConQAT, we set $n = 40$ and for Autofocus, we set $n = 20$. To evaluate a sample, developers were shown the diff of the file and its predecessor and were allowed to use any information from their IDE. The developers rated each move as *correct* or *incorrect*, indicating also their level of confidence from 0 (*I am guessing*) to 2 (*I am sure*). Each move was evaluated by at least three developers. In case of disagreement, we chose majority vote as decision criteria. As we expected most moves to be correct, we added m files with a randomly chosen predecessor in the survey to validate the developer's attention while filling out the survey. For ConQAT we chose $m = 20$, for Autofocus $m = 10$.

5.5.2 Results

Lower bound. Table 5 shows a lower bound for the precision. The accuracy of the lower bound depends on the number of implicit moves as they were all assumed to be false positives. Hence, the more implicit moves a system has, the lower the lower bound.

The systems ConQAT and Subclipse show a very high lower bound for the precision with above 90% for all scenarios. This means that among all explicit moves, the heuristics performed correctly in at least nine out of ten cases. The execution of name before clone heuristic, which has led to the highest recall, also leads to the highest lower bound of the precision (97% for ConQAT, 99% for Subclipse). For ArgoUML and Autofocus, the lower bound drops as both systems have a large number of implicit moves which were counted as false positives (see Table 2). However, even if all implicit moves were false positives, the precision would still be 71% or 72% respectively.

Developers' Rating. Table 6 shows the precision for the sample moves in the survey based on the developers' rating. For both survey objects, the clone heuristic performs better than the name heuristic, which achieves a precision of 92,5% for ConQAT and 90% for Autofocus 3. Developers occasionally disagreed with the name heuristic for the following reasons: In both systems, there are architectural constraints that different parts of the system need to have one specific file that has the same name and the almost same context. In ConQAT, these are the `Activator.java` and `BundleContext.java` which appear exactly once for each ConQAT bundle. The developers commented that those samples were clones but not necessarily a copy as the developers could not decide which predecessor is the right one. For Autofocus, developers commented on similar cases: Although the newly added class was very similar to the suggested predecessor, they did not consider it as a copy because it was added in a very different part of the system and should not share the history of the suggested predecessor.

For ConQAT, the clone heuristic achieves a precision of 97.5%, which is clearly above the precalculated lower bound of 91% In some cases, the clone heuristic outputs a different result than the repository due to the following reason: In ConQAT's history, the developers performed two major moves by copying the system to a new location in multiple steps. Each step was recorded in the repository and did not change a file's content. The clone heuristic chose a file as predecessor, which was in fact a pre-predecessor (an ancestor) according to the repository. In other words, the clone heuristic skipped one step of the copying process. However, developers still considered these results as correct. For Autofocus, the developers never disagreed with the clone heuristic, hence, it achieves a precision of 100%.

All samples with a random origin were evaluated as *incorrect*, showing that the participants evaluated with care and did not only answer *correct* for all cases. In terms of inter-rater-agreement, the Autofocus developers gave the same answers on all samples, the ConQAT developers agreed except for two cases when the majority vote was applied. Considering the confidence on judging whether a file was moved or copied from the suggested predecessor, the Autofocus developers had an average confidence value (per developer) between 1.77 and 2.0. The ConQAT developers indicated on average a confidence between 1.86 and 2.0.

Statistical Approximation. We combine the results from the survey on implicit moves and the repository information on explicit moves to calculate a statistical, overall approximation of the heuristics' precision: For all moves that were detected by a heuristic and that were explicitly recorded in the VCS, we use a precision of 100% as we assume the explicit move information to be correct. For all implicit moves, we use the precision resulting from the developer survey. We weight both precisions by the relative frequency of explicit and implicit moves. Table 7 shows the final results. On both survey objects, the clone and the name-based heuristic achieve a very high precision of above 97% in all cases with only minor differences between a single execution of a heuristic or both heuristics in either order. As the best recall was

Table 8: Relevance

Name	Implicit head moves	Head files	Information gain
ConQAT	114	3394	3.3%
ArgoUML	741	1904	38.9%
Subclipse	31	872	3.5%
Autofocus 3	1067	4585	23.2%
JabRef	30	628	4.7%
JHotDraw 7	145	692	20.9%
Vuze	666	3511	18.8%

Table 9: Execution times of the algorithm

Name	Commits	Total [h]	∅ per commit [s]
ConQAT	9309	2.1	0.83
ArgoUML	11721	9.6	2.9
Subclipse	2317	2.8	4.3
Autofocus 3	4487	6.5	5.2
JabRef	1521	2.4	5.6
JHotDraw 7	435	0.9	7.6
Vuze	20762	29.5	5.1

achieved by the execution of the name heuristic before the clone heuristic and differences in precision are only minor, we conclude that this is the best execution sequence.

5.6 Relevance (RQ3)

5.6.1 Design

To show the relevance of our approach in practice, we evaluate for all systems in Table 1 how many implicit moves exist. We calculate the number of implicit moves for a file in the head revision divided by the total number of head files to show the information gain of our approach. This fraction can be interpreted in two ways–either as average number of moves per file in the head revision that would not have been detected without our approach or as percentage of all files in the head that had on average one undetected move in their history. This quantifies the error rate of current state of the art repository analyses would have without our approach.

5.6.2 Results

Table 8 shows the approach's information gain. The number of implicit moves found in head files is smaller than the overall number of implicit moves: files deleted during the history do not appear in the head but their moves still count for the overall number of implicit moves. The table shows that for some systems significant origin information would be lost without our approach: For ArgoUML, *e. g.*, the information gain is 39% which mainly results from the loss of history information during the migration from CVS to SVN. The same applies to Vuze, for which our approach yields to an information gain of 19%. Without our approach, the origin information of the CVS history would have been lost in both cases. Also the information gain on other systems, such as jHotDraw (21%) or Autofocus (23%), reveals that our approach reconstructs a large amount of missing information. However, our approach's relevance is not limited to CVS-to-SVN conversions but also applies to any other VCS change. Naturally, the relevance of our approach varies with different systems. For well-maintained histories such as ConQAT or Subclipse, the information gain of our approach is less significant. As we restricted the information gain to files in the head, we ignored implicit moves for deleted files.

5.7 Performance (RQ4)

5.7.1 Design

We measure the algorithm's performance with the total execution time and the average execution time per commit. The experiments were run on a virtual server with 4GB RAM and 4 CPU cores (Intel Xeon). The approach is parallelized only between projects but not within a project. We evaluate the performance of the experimental setup (running heuristics in parallel, Figure 4) which denotes an upper bound of the intended execution of the algorithm (running heuristics sequentially).

5.7.2 Results

Table 9 shows the total execution time and the average per commit. Many systems were analyzed in about two hours. For the longest-lived system (Vuze, 20762 commits in 10 years), the approach took a little more than a day to finish. The average time per commit ranged from below one second up to eight seconds per commit. We consider the execution times of the algorithm feasible for application in practice.

Compared to the reported performance of Lavoie's algorithm, [22], our analysis runs significantly faster: To analyze two versions of a system, Lavoie reports between 24 and 600 seconds depending on the size of the system (between 240kLoc and 1.2 MLoc). With our systems being in the same size range, we can compare two versions on average in at most 8 seconds (jHotDraw). Our approach benefits from the incremental clone detection. However, we did not conduct an explicit comparison to a non-incremental approach as this has already been done in [13].

6. DISCUSSION AND FUTURE WORK

After presenting the results, we discuss applications and limitations of our approach, deriving future work.

Applications. Our approach is designed to incrementally reconstruct the history of a source code file to provide complete history data. This is relevant for all static code analyses that mine the software evolution, such as type change analysis, signature change analysis, instability analysis [16] or clone genealogy extractors, clone stability analyses, or defect prediction based on change frequency. Furthermore, for monitoring software quality continuously in long-lived projects, our approach enables to show a precise and complete history for all quality metrics such as structural metrics, clone coverage, comment ratio, or test coverage.

Limitations. Our approach cannot detect reverts when a file was readded from a prior revision. Future work might solve this problem by keeping all data in the clone index. However, it remains debatable whether a complete file's history should contain the revert or stop at the reverting add as the developer had deleted the file on purpose.

We designed our approach for the main trunk of the repositories, excluding branches and tags. Including branches and tags into our approach requires a cross linking between the main trunk and other branches. Otherwise, the name-based heuristic produces wrong results when a file is added to

a second branch because it will detect the file in the first branch to have a more similar path than the file in the trunk. Future work is required to establish the cross linking and to integrate this information with the heuristics.

Ongoing work comprises the comparison of our approach to the tracking mechanism of Git. For future work, we plan to conduct a quantitative and quality comparison.

7. THREATS TO VALIDITY

The recall is based only on the explicit move information and, hence, provides only an approximation of the real recall. However, to our best knowledge, there is no other way of evaluating the recall as even developers would not be able to tell you all moves, copies, and renames if the history of their software comprises several years. To make the recall approximation as accurate as possible we chose systems with a large number of explicit moves in their history.

To get a statistical sample to approximate the overall precision of the approach, we conducted a developer survey for ConQAT and Autofocus. As ConQAT is our own system, one might argue that the developers do not provide accurate information to improve the results of this paper. However, the first author of this paper was not a developer of ConQAT and did not participate in the case study.

We evaluated the use of our tool only on SVN and Java. Generally, our approach is independent from the underlying VCS and programming language. It can be also used with other VCS such as Git or TFS.

8. CONCLUSION

We presented an incremental, commit-based approach to reconstruct the complete history of a source code file, detecting origin changes such as renames, moves, and copies of a file during evolution. With a precision of 97-99%, the approach provides accurate information for any evolution-based static code analysis such as code churn or code stability. A recall of 98% for almost all systems indicates that the heuristics can reliably reconstruct any missing information. A case study on many open source systems showed that our approach succeeds to reveal a significant amount of otherwise missing information. On systems such as ArgoUML for example, 38.9% files had an incomplete history as recorded in the repository. Without our approach, current static code analyses that rely on file mapping during evolution would have made an error on more than every third file. In contrast to many state of the art tools, our approach is able to analyze every single commit even in large histories of ten thousand revisions in feasible runtime.

9. REFERENCES

[1] Git. `http://www.git-scm.com/`. [Online; accessed 2013-12-03].

[2] G. Antoniol, M. D. Penta, and E. Merlo. An Automatic Approach to Identify Class Evolution Discontinuities. In *IWPSE '04*, 2004.

[3] M. Asaduzzaman, C. Roy, K. Schneider, and M. D. Penta. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *ICSM'13*, 2013.

[4] V. Bauer, L. Heinemann, B. Hummel, E. Juergens, and M. Conradt. A Framework for Incremental Quality Analysis of Large Software Systems. In *ICSM'12*, 2012.

[5] G. Canfora, L. Cerulo, and M. D. Penta. Identifying Changed Source Code Lines from Version Repositories. In *MSR'07*, 2007.

[6] Davies, J. and German, D. M. and Godfrey, M. W. and Hindle, A. Software bertillonage: finding the provenance of an entity. In *MSR'11*, 2011.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA'00*, 2000.

[8] N. Göde and J. Harder. Clone stability. In *CSMR'11*, 2011.

[9] M. Godfrey and Q. Tu. Tracking structural evolution using origin analysis. In *IWPSE'02*, 2002.

[10] M. Godfrey and L. Z. Using origin analysis to detect merging and splitting of source code entities. *Software Engineering, IEEE Transactions on*, 31(2), 2005.

[11] Godfrey, M. W. and German, D. M. and Davies, J. and Hindle, A. Determining the provenance of software artifacts. In *IWSC'11*, 2011.

[12] L. Heinemann, B. Hummel, and D. Steidl. Teamscale: Software Quality Control in Real-Time. In *ICSE '14*, 2014.

[13] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *ICSM'10*, 2010.

[14] J. Hunt and W. Tichy. Extensible language-aware merging. In *ICSM'02*, 2002.

[15] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *ISSRE'96*, 1996.

[16] M. Kim and D. Notkin. Program Element Matching for Multi-version Program Analyses. In *MSR'06*, 2006.

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An Empirical Study of Code Clone Genealogies. In *FSE'05*, 2005.

[18] S. Kim, K. Pan, and E. J. Whitehead, Jr. When Functions Change Their Names: Automatic Detection of Origin Relationships. In *WCRE'05*, 2005.

[19] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Recovering the Evolution Stable Part Using an ECGM Algorithm: Is There a Tunnel in Mozilla? In *CSMR'09*, 2009.

[20] J. Krinke. Is Cloned Code More Stable than Non-cloned Code? In *SCAM'08*, 2008.

[21] J. Krinke. Is cloned code older than non-cloned code? In *IWSC'11*, 2011.

[22] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou. Inferring Repository File Structure Modifications Using Nearest-Neighbor Clone Detection. In *WCRE'12*, 2012.

[23] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE'08*, 2008.

[24] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE'05*.

[25] Q. Tu and M. W. Godfrey. An Integrated Approach for Studying Architectural Evolution. In *IWPC'02*, 2002.

[26] F. Van Rysselberghe and S. Demeyer. Reconstruction of Successful Software Evolution Using Clone Detection. In *IWPSE'03*, 2003.

7.2 Analysis of Software Changes

The paper "How Do Java Methods Grow?" (Paper **B**, [84]) contributes to the *automatic recommendation* step of the approach. In particular, we derived the change-based prioritization from it.

Goal The paper aims to obtain a better understanding of how software systems and their methods grow over time. A better understanding how methods are changed helps to schedule refactorings that are closely aligned with ongoing development.

Approach To obtain a better understanding of software evolution, the approach mines the history as recorded in the version control system. For each method, it records its length after any modification. To obtain accurate results, it is important to track methods even in presence of refactorings such as method renames, parameter changes, file renames, or file moves. Therefore, the approach contains a fine grained *origin-analysis* for methods which was adopted from the file-based origin analysis in Paper **A**, [89].

Evaluation The paper comprises a case study of nine open source and one industry system. Each system has at least three years of history and reveals a strong growth in system size. In total, we analyzed 114,000 methods in a history of 72 years with about 61,000 commits.

Results The paper shows that the case study systems grow by adding new methods rather than through growth in existing methods. Many methods—about half of them—even remain unchanged after their initial commit. They reveal a strong growth only if modified frequently. These results revealed little variance across all projects. As a summary, the paper gives the recommendation for developers to focus refactoring effort on frequently modified code hot spots as well as on newly added code.

Thesis Contribution We use the results to derive the change-based prioritization for internal findings from it: As half of the methods remain unchanged in the analyzed history, refactoring them would have no benefit in terms of enhanced changeability (but only in enhanced readability). Hence, our change-based prioritization does not prioritize these methods, but focuses on methods that were modified or newly added in the last iteration (see Section 6.2.3). Removing findings in modified code reduces overhead costs for code understanding and retesting because performing a change request already comprises both. Additionally, modified methods have a slightly higher probability to be modified again (see study in Section 5.5) and, hence, developers are expected to benefit from the findings removal in terms of enhanced changeability.

How Do Java Methods Grow?

Daniela Steidl, Florian Deissenboeck
CQSE GmbH
Garching b. München, Germany
{steidl, deissenboeck}@cqse.eu

***Abstract*—Overly long methods hamper the maintainability of software—they are hard to understand and to change, but also difficult to test, reuse, and profile. While technically there are many opportunities to refactor long methods, little is known about their origin and their evolution. It is unclear how much effort should be spent to refactor them and when this effort is spent best. To obtain a maintenance strategy, we need a better understanding of how software systems and their methods evolve. This paper presents an empirical case study on method growth in Java with nine open source and one industry system. We show that most methods do not increase their length significantly; in fact, about half of them remain unchanged after the initial commit. Instead, software systems grow by adding new methods rather than by modifying existing methods.**

I. Introduction

Overly long methods hamper the maintainability of software—they are hard to understand and to change, but also to test, reuse, and profile. Eick et al. have already determined module size to be a *risk factor* for software quality and "cause for concern" [1]. To make the code modular and easy to understand, methods should be kept small [2]. Small methods also enable more precise profiling information and are less error-prone as Nagappan et al. have shown that method length correlates with post-release defects [3]. Additionally, the method length is a key source code property to assess maintainability in practice [4], [5].

Technically, we know *how* to refactor long methods—for example with the *extract method* refactoring which is well supported in many development environments. However, we do not know *when* to actually perform the refactorings as we barely have knowledge about: How are long methods created? How do they evolve?

If methods are born long but subsequently refactored frequently, their hampering impact on maintainability will diminish during evolution. In contrast, if overly long methods are subject to a broken window effect[1], more long methods might be created and they might continue to grow [7]. Both scenarios imply different maintenance strategies: In the first case, no additional refactoring effort has to made as developers clean up on the fly. In the second case, refactoring effort should be made to prevent the broken window effect, *e. g.*, by refactoring a long method immediately after the first commit.

To better know when to refactor overly long methods, we need a deeper understanding of how methods evolve. In a case study on the history of ten Java systems, we study how single methods and the overall systems grow. We track the length of methods during evolution and classify them as *short*, *long*, and *very long* methods based on thresholds. The conducted case study targets the following five research questions regarding the evolution of a single method (RQ1, RQ2, RQ3) and the entire system (RQ4, RQ5):

RQ1 *Are long methods already born long?* We investigate whether methods are already long or very long at their first commit to the version control system or whether they gradually grow into long/ very long methods.

RQ2 *How often is a method modified in its history?* We examine which percentage of methods is frequently changed and which percentage remains unchanged since initial commit.

RQ3 *How likely does a method grow and become a long/very long method?* We check if methods, that are modified frequently, are likelier to grow over time, or, in contrast, likely to be shortened eventually.

RQ4 *How does a system grow—by adding new methods or by modifying existing methods?* We build upon RQ2 to see how many changes affect existing methods and how many changes result in new methods.

RQ5 *How does the distribution of code in short, long, and very long methods evolve?* We investigate if more and more code accumulates in long and very long methods over time.

The contribution of this paper is two-fold: First, we give insights into how systems grow and, second, we provide a theory for when to refactor. As the case study reveals that about half the methods remain unchanged and software systems grow by adding new methods rather through growth in existing methods, we suggest to focus refactoring effort on newly added methods and methods that are frequently changed.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant Q-Effekt, 01IS15003A". The responsibility for this article lies with the authors.

[1]The broken window theory is a criminological theory that can also be applied to refactoring: a dirty code base encourages developers to try to get away with 'quick and dirty' code changes, while they might be less inclined to do so in a clean code base [6], [7].

II. Related Work

A. Method Evolution

Robles et al. analyze the software growth on function level combined with human factors [8]. However, the authors do not use any origin analysis to track methods during renames or moves. Instead, they exclude renamed/moved functions. The result of our RQ2 is a replication of their finding that most functions never change and when they do, the number of modifications is correlated to the size—but our result is based on a thorough origin analysis. Whereas Robles et al. further study the social impact of developers in the project, we gain a deeper understanding about how systems and their methods grow to better know when methods should be refactored.

More generally, in [9], Girba and Ducasse provide a meta model for software evolution analysis. Based on their classification, we provide a *history-centered* approach to understand how methods evolve and how systems grow. However, the authors state that most history-centered approaches lack fine-grained semantical information. We address this limitation by enriching our case study with concrete examples from the source code. With the examples, we add semantic explanations for our observed method evolution patterns.

B. System Size and File Evolution

Godfrey and Tu analyze the evolution of the Linux kernel in terms of system and subsystem size and determine a super linear growth [10]. This work is one of the first steps in analyzing software evolution which we expand with an analysis at method level. In [11], the authors discover a set of source files which are changed unusually often (active files) and that these files constitute only between 2-8% of the total system size, but contribute 20-40% of system file changes. The authors similarly mine code history and code changes. However, we gain a deeper understanding about the evolution of long methods rather than the impact of active files.

C. Origin Analysis

In [12], we showed that up to 38% of files in open source systems have at least one move or rename in its history which is not recorded in the repository. Hence, tracking code entities (files or methods) is important for the analysis of code evolution. We refer to this tracking as origin analysis.

In [13], [14], the authors present the tool Beagle which performs an *origin analysis* to track methods during evolution. This relates to our work, as we also conduct a detailed origin analysis. However, instead of using their origin analysis, we reused our file-based approach from [12] which has been evaluated thoroughly and was easily transferable to methods.

To analyze method evolution, the authors of [15] propose a sophisticated and thoroughly evaluated approach to detect renamings. Our origin analysis can detect renamings as well, however, at a different level: Whereas for us, detecting a renaming as an origin change suffices, the work of [15] goes further and can also classifies the renaming and with which purpose it was performed. As we do not require these details, our work relies on a more simplistic approach.

In [16], the authors propose a change distilling algorithm to identify changes, including method origin changes. Their algorithm is based on abstract syntax trees whereas our work is based on naming and cloning information. As we do not need specific information about changes within a method body, we can rely on more simplistic heuristics to detect origin changes.

Van Rysselberghe and Demeyer [17] present an approach to detect method moves during evolution using clone detection. However, in contrast to our work, as the authors use an exact line matching technique for clone detection, their approach cannot handle method moves with identifier renaming.

D. Code Smell Evolution

Three groups of researchers examined the evolution of code smells and all of them reveal that code smells are often introduced when their enclosing method is initially added and code smells are barely removed [18]–[20]. Our paper explains these findings partially: As most methods remain unchanged, it is not surprising that their code smells are not removed. For the specific code smell of long methods, our paper confirms that the smell is added with the first commit (many long methods are already born long) and barely removed (many long methods are not shortened).

III. Terms and Definition

Commit and Revision. A commit is the developer's action to upload his changes to the version control system. A commit results in a new revision within the version control system.

Initial and Head Revision. We refer to the initial revision of a case study object as the first analyzed revision of the history and to the head revision as the last analyzed revision respectively. Hence, the initial revision must not necessarily be the initial revision of the software development, and the head revision is likely not the actual head revision of the system.

Number of Statements. To measure the length of a method, we use the number of its statements. We normalize the formatting of the source code by putting every statement onto a single line. Then, we count the lines excluding empty lines and comments. The normalization eliminates the impact of different coding styles among different systems such as putting multiple statements onto a single line to make a method appear shorter.

Green, yellow, and red methods. Based on two thresholds (see Subsection IV-C), we group methods into short, long, and very long entities, and color code these groups as green (short), yellow (long), and red (very long) methods both in language as in visualization. A green (yellow/red) method is a method that is green (yellow/red) in the head revision.

Touches and Changes. We refer to the touches per method as the creation (the initial commit) plus the edits (modification) of a method, to the changes as the edits excluding the creation of the method—hence: $\#\text{touches} = \#\text{changes} + 1$.

TABLE I: Case Study Objects

Name	History [Years]	System Size [SLoC]			Revision Range	Commits	Methods		
		Initial	Head	Growth			Analyzed	Yellow	Red
ArgoUML	15	9k	177k		2-19,910	11,721	12,449	5%	1%
af3	3	11k	205k		18-7,142	4,351	13,447	5%	1%
ConQAT	3	85k	208k		31,999-45,456	9,308	14,885	2%	0%
jabRef	8	3.8k	156k		10-3,681	1,546	8,198	10%	4%
jEdit	7	98k	118k		6,524-23,106	2,481	6,429	9%	2%
jHotDraw7	7	29k	82k		270-783	435	5,983	8%	1%
jMol	7	13k	52k		2-4,789	2,587	3,095	7%	2%
Subclipse	5	1k	96k		4-5,735	2,317	6,112	9%	2%
Vuze	10	7.5k	550k		43-28,702	20,676	29,403	8%	3%
Anony.	7	118k	259k		60-37,337	5,995	14,651	5%	1%
Sum	**72**	**375.3k**	**1,912k**			**61,417**	**114,652**		

IV. Approach

We analyze a method evolution by measuring its length after any modification in history. We keep track of the lengths in an *length vector*. Based on the length vector, we investigate if the length keeps growing as the system size grows or, in contrast, if methods are frequently refactored.

A. Framework

We implemented our approach within the incremental analysis framework Teamscale [21]. Teamscale analyzes every single commit in the history of a software system and, hence, provides a fine-grained method evolution analysis. To obtain the complete history of a method, we use the fully qualified method name as key to keep track of a method during history: the fully qualified method name consists of the uniform path to the file, the method name, and all parameter types of the method. In cases the fully qualified method name changes due to a refactoring, we conduct an *origin analysis*.

B. Origin Analysis

To obtain an accurate length measurement over time, tracking the method during refactorings and moves is important—which we refer to as *origin analysis*. If the origin analysis is inaccurate, the history of an entity will be lost after a method or file refactoring and distort our results. Our method-based origin analysis is adopted from the file-based origin analysis in [12] which analyzes each commit incrementally. In principle, this origin analysis detects origin changes by extracting possible origin candidates from the previous revision and using a clone-based content comparison to detect the most similar candidate based on thresholds. We evaluated it with manual random inspection by the authors on the case study data.

We use a total number of 4 heuristics which are executed for each commit on the added methods—an added method is a method with a fully qualified method name that did not exist in the previous revision. For each added method, the heuristics aim to determine if the method is *truly* new to the system or if the method existed in the previous revision with a different fully qualified method name. A method that is *copied* is not considered to be new. Instead, it will inherit the history of its origin. The following heuristics are executed sequentially:

Parameter-change-heuristic *detects origin changes due to a changed parameter type.* It compares an added method to all methods of the previous revision in the same file and with the same name. The comparison is done with a type-II-clone detection on the content of the method body as in [12]. If the content is similar enough based on thresholds we detect this method as origin for the added method.[2]

Method-name-change-heuristic *detects origin changes due to renaming.* It compares an added method to all methods of the previous revision that were in the same file and had the same parameter types with the same content comparison.

Parameter-and-method-name-change-heuristic *detects origin changes due to renaming and changed parameter type* It compares an added method to all methods of the previous revision that were in the same file. As there are more potential origin candidates than for the previous two heuristics, the content comparison of this heuristic is stricter, *i. e.*, it requires more similarity to avoid false positives.[3]

File-name-change-heuristic *detects origin changes due to method moves between files.* It compares an added method to all methods of the previous revision that have the same method name and the same parameter types (but are located in different files).[4]

To summarize, our origin tracking is able to detect the common refactorings (as in [22]) method rename, parameter change, method move, file rename, or file move. It is also able to detect a method rename combined with a parameter change. Only a method rename and/or parameter change combined with a file rename/move is not detected. However, based on our experience from [12], we consider this case to be unlikely

[2]For replication purposes, the thresholds used were $\theta_1 = \theta_2 = 0.5$. For the meaning of the thresholds and how to choose them, please refer to [12]

[3]Similarity thresholds $\theta_1 = 0.7$ and $\theta_2 = 0.6$

[4]We also use the higher similarity thresholds to $\theta_1 = 0.7$ and $\theta_2 = 0.6$.

as we showed that file renames rarely occur: many file moves stem from repository restructurings without content change.

C. Method Length and Threshold-based Classification

To measure method length, we use its number of statements (see Section III). We include only changes to the *method body* in its length vector: As we analyze the growth evolution of methods during modification, we do not count a move of a method or its entire file as a content modification. Excluding white spaces, empty and comment lines when counting statements, we do not count comment changes either.

Based on their length, we classify methods into three categories: Based on previous work [23] and general coding best practices [2], we consider methods with less than 30 statements to be *short*, between 30 and 75 statements to be *long*, and with more than 75 statements to be *very long*—under the assumption that a method should fit entirely onto a screen to be easy to understand. We further discuss this threshold-based approach in Section VIII.

V. Case Study Set Up

The case study targets the research questions in Section I.

Study Objects. We use ten Java systems which cover different domains, and use Subversion as version control, see Table I. Nine of them are open source, the anonymous system is industrial from an insurance company. We chose study objects that had either a minimum of five years of history or revealed a strongly growing and—in the end—large system size (*e. g.*, ConQAT and af3). The systems provide long histories ranging from 3 to 15 years and all show a growing system size trend. The number of analyzed commits in the table is usually smaller than the difference between head and initial revision because, in big repositories with multiple projects, not all commits affect the specific project under evaluation. For all systems, we analyzed the main trunk of the subversion, excluding branches.

Excluded Methods. For each study object, we excluded generated code because, very often, this code is only generated once and usually neither read nor changed manually. Hence, we assume generated entities evolve differently than manually maintained ones (or do not evolve at all if never re-generated).

We also excluded test code because we assume test code might not be kept up-to-date with the source code and, hence, might not be modified in the same way. In [24], the authors showed that test and production code evolved synchronously only in one case study object. The other two case study objects revealed phased testing (stepwise growth of testing artifacts) or testing backlogs (very little testing in early stages of development). For this paper, we did not want to risk test code distorting the results, but future work is required to examine the generalizat if our results on production code can or cannot be transfered to test code.

We excluded methods that were deleted during history and only work on methods that still exist in the head revision. In some study systems, for example, code was only temporarily added: In jabRef, *e. g.*, , antlr library code was added and removed two months later as developers decided to rather include it as a .jar file. We believe these deleted methods should not be analyzed. We also want to exclude debugging code, as this code is only used temporarily and hence, likely to be maintained differently than production code. Excluding deleted methods potentially introduces a bias because, for example, also experimental code that can be seen as part of regular development, is excluded. However, we believe that including deleted methods introduces a larger bias than excluding them.

We excluded simple getter and setter methods as we expect that these methods are not modified during history. Table I shows for each system the total number of remaining, analyzed methods as well as the percentage of yellow and red methods. Yet, the table's growth plots show the size of the code base *before* excluding deleted, getter, and setter methods. Hence, for jabref, the plot still shows the temporary small pike when the antlr library was added and deleted shortly after.

VI. Research Question Evaluation

For each research question, we describe the evaluation technique, provide quantitative results, supplement them with qualitative examples when necessary, and discuss each result.

A. Are long methods already born long? (RQ1)

We investigate whether the yellow and red methods are already yellow or red at the first commit of the method.

1) Evaluation: For each project, we count the number of methods that were already yellow or red upon their first commit relative to the total number of yellow or red methods in the head revision. Over all projects, we also calculate the average $\varnothing$ and the standard deviation σ. As the projects differ greatly in the overall number of yellow/red methods, we calculate the weighted average and the weighted standard deviation and use the number of methods per project as weight.

2) Quantitative Results: Table II shows the results: between 52% and 82% of yellow methods were already born yellow—on average about 63%. Among all red methods, between 38% and 88% with an average of 51% were born already red.

3) Discussion: More than half of the red methods are already born red and two third of the yellow methods are already born yellow. This average reveals a small standard deviation among the ten case study objects. Hence, these methods do not grow to become long but are already long from the beginning. We also investigated that green methods are already born green: in all case study objects, between 98–99% of the green methods are already born green. As our origin analysis tracks method refactorings and also code deletions within a method, we conclude that most short methods are initially committed as a short method rather than emerging from a longer method.

4) Conclusion: Most yellow/red methods are already born yellow/red rather than gradually growing into it.

B. How often is a method modified in its history? (RQ2)

We examine how many methods are frequently changed and how many remain unchanged since their initial commit.

TABLE II: Percentage of yellow/red methods that were born already yellow/red with average ∅ and standard deviation σ

ArgoUML	af3	ConQAT	jabRef	jEdit	jHotDraw	jMol	Subclipse	Vuze	Anony.	∅	σ
62% 46%	72% 57%	72% 88%	64% 58%	82% 73%	72% 68%	60% 38%	67% 48%	52% 45%	63% 49%	63% 51%	10% 9%

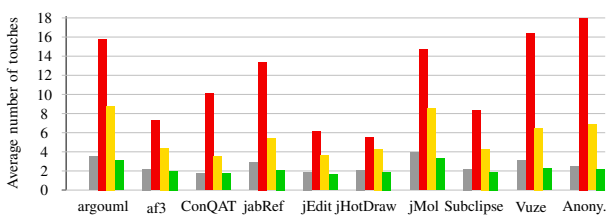


Fig. 1: Average number of touches overall (gray), to short (green), long (yellow), and very long (red) methods

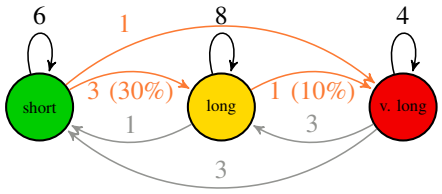


Fig. 2: Method growth (orange) and method shrinking transitions (gray) in a Markov chain

1) Evaluation: First, we evaluate the average number of touches to a method over its life time. We analyze this average for all methods and for all green, yellow, and red methods individually. Second, we compute the percentage of methods that remain unchanged.

2) Quantitative Results: Figures 1 shows the average number of touches. The results are very similar for all case study systems. On average, any method is touched 2–4 times, a green method between 2 and 3 times. This average increases on yellow and even more on red methods: Red methods are changed more often than yellow methods than green methods.

Table III shows how many methods are changed or remain unchanged after their initial commit. For each system, the table denotes the percentage of changed and unchanged methods, which sum up to 100%. The horizontal bar chart visualizes the relative distribution between changed (gray) and unchanged (black) entities. Only between 31% and 63% of the methods are touched again after initial commit. On average, 54% of all methods remain unchanged. This is the weighted average over all percentages of unchanged methods, with the number of analyzed methods from Table I as weights.

3) Examples: In ArgoUML, frequently touched red methods comprise the `main` method (167 touches) and `run` (19 touches) method in the classes `Main.java` and `Designer.java`, for example. Other methods that are frequently touched are the parsing methods for UML notations such as `parseAttribute` in `AttributeNotationUml.java` or `parseTrigger` in `TransitionNotationUml.java`. Parsing seems to be core functionality that changes frequently.

4) Discussion: When estimating where changes occur, one can make different assumptions: First, one could assume that changes are uniformly distributed over all lines of source code. Second, one could assume that they are uniformly distributed over the methods. Third, changes are maybe not uniformly distributed at all. Our results support the first assumption—under which longer methods have a higher likelihood to get changed because they contain more lines of code. We also evaluated if yellow and red methods have a longer history in our case study than green methods. This would be a very simple reason why they are changed more often. We analyzed the history length of each method, counting the days from its initial commit to the head revision. However, the average analyzed history of a yellow or red method is about the same as for a green one. Even when analyzing the average number of changes per day, the result remains the same: red methods are modified more often than yellow ones and yellow methods are more often modified than green methods.

As about half of the methods remain unchanged, this raises two questions: First, have certain components become unused such that they could be deleted? To mitigate this threat of analyzing out-dated code components, we additionally analyzed where changes occur on file-level. In contrast to the methods, we found that 91% of all files[5] are touched at least twice, 83% at least three times, still 67% at least five times. While the file analysis does not eliminate the threat completely, we yet believe it reduces its risk. Second, this also raises the question where the changes to a system actually occur. If many existing methods remain unchanged, changes either focus on only half the methods or must result in new methods. RQ4 examines this in more detail.

5) Conclusion: On average, a method gets touched between two and fours times. About half the methods remain unchanged.

C. How likely do modified methods grow and become yellow or red? (RQ3)

The results of RQ1 and RQ2 showed that many methods are already born yellow and red and many methods are not even changed. This research question examines if methods that are frequently modified grow over time or, in contrast, if their likelihood to be refactored increases when modified often.

1) Evaluation: We examine how likely a green method becomes yellow and a yellow method becomes red—analyzing the transitions between green, yellow, and red methods—and, vice versa, how likely it is refactored.

[5]This is the average over all systems with min. 80% and max. 100%

TABLE III: Percentage of changed (gray) / unchanged (black) methods

ArgoUML	af3	ConQAT	jabRef	jEdit	jHotDraw	jMol	Subclipse	Vuze	Anony.
62%/38%	46%/54%	31%/69%	49%/51%	35%/65%	46%/54%	63%/37%	39%/61%	48%/52%	48%/52%

Transition likelihoods in Markov model. In particular, we model the method evolution as a Markov diagram with states *green*, *yellow*, and *red*, see Figure 2. To investigate the *method growth transitions*, we examine the green → yellow, yellow → red, green → red transitions. We further examine the red → yellow, yellow → green, red → green transitions and refer to them as *method shrinking transitions*. However, strictly speaking, these transitions can not only result from refactorings, but also from code deletions within a method. Hence, the resulting *shrinking* probabilities denote only an upper bound for the real refactoring probability.

We model a transition based only on the method's state in the initial and head revision, *e. g.*, a method that was born green, grows yellow, and is refactored to green again, will lead to a transition from green to green. To estimate the likelihood of a transition, we calculate the relative frequency of the observed transitions in the histories of the case study objects: if a method was, *e. g.*, born green and ended up yellow in the head revision, then we count this method evolution as one transition from state green to yellow. Figure 2 gives a fictitious example with 30 methods in total, 10 methods each being born green, yellow, and red respectively. Among all yellow methods, 8 remained yellow till the head, one became green, one became red. Among all green methods, 6 remained green, three evolved into yellow, one into red. In this example, the relative frequency for a method to switch from green to yellow would be 30% ($=\frac{3}{10}$), to switch from yellow to red would be 10% ($=\frac{1}{10}$).

Modification-dependent Markov model. We analyze the transition likelihoods in the Markov model under four different constraints depending on the number n of touches per method.

First, we calculate the Markov model for all methods ($n \geq 1$). Then, we compute the model on the subset of methods being touched at least twice ($n \geq 2$), at least five ($n \geq 5$) and at least ten times ($n \geq 10$). We investigate whether methods that are modified often ($n \geq 10$) have a higher probability to grow than less modified ones. In contrast, one could also assume that a higher number of modifications increases the likelihood of a refactoring, *e. g.*, when developers get annoyed by the same long method so that they decide to refactor it if they have to touch it frequently. We choose these thresholds for n because five is approximately the average number of touches over all yellow methods in all study objects and ten is the smallest upper bound for the average touches to a yellow method for all systems, see Figure 1. A larger threshold than 10 would reduce the size of the method subset too much for some case study objects—making the results less representative.

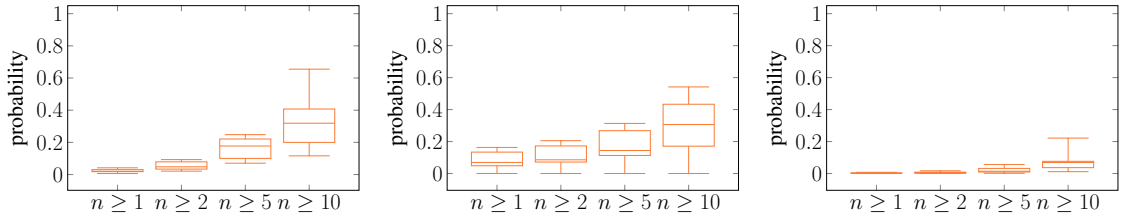
Aggregation. For each observed transition combined with each constraint on n, we aggregate the results over all case study objects by calculating the median, the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentile as well as the minimum and maximum.

2) Quantitative Results: Figure 3 shows the results for method growth transitions with box-and-whisker-plots. Taking all methods into account without modification constraint ($n \geq 1$), all three growth transition probabilities are rather small. The highest probability is for the yellow → red transition with a mean of 7%. Hence, we can not observe a general increase in method length which is because many methods are not changed. However, the mean of every growth transition probability increases from $n \geq 1$ to $n \geq 10$: the more a method is modified, the more likely it grows. For the yellow → red transition, for example, the mean increases from 7% to 32%. This means that every third yellow method that is touched at least ten times will grow into a red method. For the short methods, 21% of the methods being touched at least five times grow into long methods. If short methods are touched at least ten times, they grow into a long method with 27% probability. The transition green → red occurs less often.

Figure 4 shows the results for the method shrinking transitions (stemming from refactorings or code deletions). For all three transitions and independent from the number of modifications, the mean is between 10% and 25%. For the red → yellow transition, this probability is between 10% and 15%, for yellow → green between 15% and 22%.

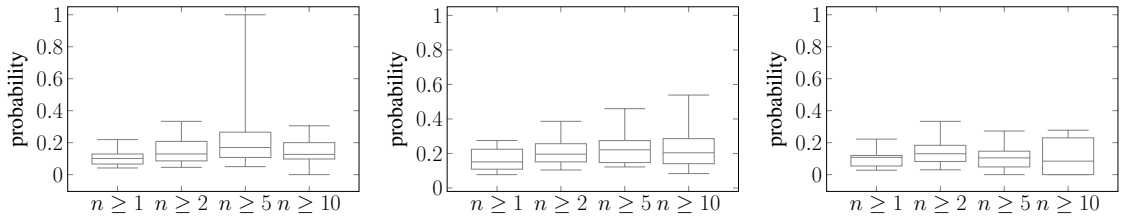
3) Examples: The method `setTarget` in `TabStyle.java` (ArgoUML) serves as a method, that grows from yellow to red while being frequently touched: It only contained 34 statements in revision 146 (year 1999). After 19 touches and four moves, it contained 82 statements in revision 16902 (year 2009). The method gradually accumulated more code for various reasons: The type of its parameter `t` got generalized from `Fig` to `Object`, requiring several additional `instanceof Fig` checks. Additionally, the method now supports an additional `targetPanel` on top of the existing `stylePanel` and it also now repaints after validating.

4) Discussion: This research question shows that there is a strong growth for frequently modified methods: Every third yellow method that is touched at least ten times grows into a red method and every fifth green method that is touched at least five times grows into a yellow method. However, the overall probability ($n \geq 1$) that a method grows and passes the next length threshold is very small for all three growth transitions (with a the largest mean of 7% for yellow → red). As we know that all case study systems are growing in system size (see Table I), this raises the question where the new code is added. In VI-B, we concluded that changes either affect only half of the code or result mainly in new methods. As many methods are unchanged and we do not observe a general



(a) $green \rightarrow yellow$ (b) $yellow \rightarrow red$ (c) $green \rightarrow red$

Fig. 3: Probabilities for *growth* transitions in the Markov model for methods



(a) $red \rightarrow yellow$ (b) $yellow \rightarrow green$ (c) $red \rightarrow green$

Fig. 4: Probabilities for *refactorings* transitions in the Markov model for methods

growth in method length, we suspect that most changes lead to new methods. The next research question will address this.

In terms of refactorings, we cannot observe a strong correlation between the number of modifications and the mean shrinking probability. We conclude that the upper bound for the probability of method to be refactored is independent from the number of modifications it receives. In general, the probability that a method shrinks is rather low—only between every tenth and every fifth method cross the thresholds.

5) Conclusion: The more an entity is modified, the likelier it grows. In contrast to previous research questions that found no size increase for methods in general, we reveal a strong method growth for methods that are modified. However, based on our markov modeling, the number of modifications does not have an impact on the method shrinking probability. Our observed likelihood that a method is refactored is at most 25%.

D. How does a system grow—by adding new methods or by modifying existing methods? (RQ4)

As RQ2 revealed that a large percentage of methods remain unchanged after the initial commit, we aim to determine how a system grows—by primarily adding new methods or by modifying existing ones?

1) Evaluation: We use the following metric to calculate the quotient of code growth that occurs in new methods:

$$\text{growth\_quotient} = \frac{\text{code\_growth}_{\text{new}}}{\text{code\_growth}_{\text{existing}} + \text{code\_growth}_{new}}$$

With code growth, we refer to the number of added minus the number of deleted statements. Instead of using code churn (added *plus* deleted statements), we use the number of added *minus* deleted statements because we are interested in where the software grows. In case of positive code growth in new and in existing methods (more added statements than deleted statements), the growth quotient takes a value between 0 and 1 and, hence, denotes the percentage of code growth that occurs in new methods. If the growth quotient is negative, more code is deleted in existing methods than added in new methods (hence, the system got reduced in size). If the growth quotient is larger than 1, code was deleted in existing methods, but more statements were added in new methods.

To differentiate between code growth in new and in existing methods, we need to define the concept of a new method: we need to define a *time interval* in which a method is considered to be *new*. All methods that were added *after* the interval start date are considered *new methods* for this interval. For the calculation of the subsequent interval, these methods will be counted as existing methods. Whenever our origin analysis detects a method extraction, we consider the extracted method still as *existing* code as this code was not newly added to the system—it rather denotes a refactoring than a development of new code contributing to the system's grow.

We decided to use the interval lengths of one week, 6 weeks and three months. We also could have used different time intervals to define the concept of a new method and analyzed the history *e. g.*, commit-based. Hence, code growth could only occur in a new method in the commit with which the method was added to the system. An implementation of a new feature that spreads out over several commits would then result in new code growth only in the first commit. All other commits would produce code growth in existing methods. This contradicts the intention of the research question to investigate how the system grows over the time span of many years because—with this intention—the entire implementation of a new feature should result in code churn in new methods.

TABLE IV: Average growth quotient ∅ and its deviation σ on 3-months, 6-weeks, and weekly basis

		ArgoUML	af3	ConQAT	jabRef	jEdit	jHotDraw	jMol	Subclipse	Vuze	Anony.
3m	∅	85%	96%	96%	70%	72%	62%	74%	71%	76%	93%
	σ	56%	4%	3%	28%	24%	33%	37%	24%	12%	10%
6w	∅	83%	92%	95%	69%	67%	53%	75%	66%	95%	92%
	σ	100%	21%	4%	32%	38%	39%	37%	30%	209%	13%
1w	∅	91%	84%	83%	43%	45%	52%	95%	50%	59%	85%
	σ	750%	88%	142%	171%	62%	52%	231%	102%	41%	103%

2) Quantitative Results: Table IV shows the growth quotient as the average ∅ over all intervals and its standard variance σ. On a three month basis the code growth is between 62% and 96% in new methods and on the 6-weeks basis between 53% and 95%. Hence, for both interval spans, at least half of the code growth occurs in new methods rather than in existing methods. For the weekly basis, the growth quotient drops to 43% up to 95%.

3) Examples: The history of jEdit provides illustrative examples: Commit 17604 adds "the ability to choose a prefered DataFlavor when pasting text to textArea" and increases the system size by 98 SLoC—`JEditDataFlavor.java` is added (7 SLoC), `RichTextTransferable.java` is modified slightly within an existing method, and the rest of the system growth stems from the modification of `Registers.java` which grows by adding two new `paste` methods that both have a new, additional `DataFlavor` parameter. Commit 8845 adds a new layout manager by adding few lines to `VariableGridLayout` within existing methods and adding two new files, increasing the overall system size by 872 SLoC.

4) Discussion: Unsurprisingly, for the weekly time interval, the growth quotient drops for many systems as it depends on the commit and development behavior (systems in which the new feature implementations spread over weeks rather than being aggregated in one commit reveal a lower growth quotient for a shorter interval). However, based on the 6-weeks and three-months basis, we observe that—in the long run and independent from the commit behavior—software grows through new methods. These results match the outcome of the previous research questions: Most systems grow as new methods are added rather than through growth in existing methods. In particular, many methods remain unchanged after the initial commit (RQ2). Our results further match the result of [25] which showed that earlier releases of the system are no longer evolved and that most change requests to the software in their case study were perfective change requests.

5) Conclusion: Systems grow in size by adding new methods rather than by modifying existing ones.

E. How does the distribution of code in short, long, and very long methods evolve over time? (RQ5)

RQ1-RQ3 investigated the growth of a single method and RQ4 investigated the system growth. This research question examines how the overall distribution of the code over green, yellow, and red methods (*e. g.*, 50% code in green methods, 30% in yellow methods, 20% in red methods) evolves. This distribution corresponds to the probability distribution that a developer who changes a single line of code has to change a green, yellow, or red method. If the probability to touch a line in a yellow or red method increases over time, we consider this an increasing risk for maintainability—and an indicator for a required maintenance strategy to control the amount code in yellow or red methods.

1) Evaluation: After each commit, we calculate the distribution of the code over green, yellow, and red methods: we sum up the statements in all green, yellow, and red methods separately and count it relative to the overall number of statements in methods. Thus, we obtain a code distribution metric and—over the entire history—the distribution trend.

2) Quantitative Results: Figures 5 shows the results for the evolution of the code distribution in green, yellow, and red methods. On the x-axis, we denote the years of analyzed history, the y-axis denotes the code distribution. The red plot indicates the relative amount of code in red methods, the yellow plot indicates the relative amount of code in both red and yellow methods. The green lines indicate the amount of code in green, yellow, and red methods, and hence, always sums up to 100%. An increasing red or yellow plot-area indicates as a *negative trend* or an increasing risk for maintainability.

Figure 5 shows that there is no general negative trend. Only four systems (af3, jHotdraw, Subclipse, and Vuze) reveal a slight increase in the percentage of code in yellow and red methods. For ConQAT, jabref, and jEdit, the trend remains almost constant, for ArgoUML, jMol, and the industry system it even improves (partly).

3) Discussion: For systems without negative trend, we assume that the length distribution of the new methods is about the same as for the existing methods. As we have shown that there is no major length increase in existing methods, but systems grow by adding new methods instead, it is plausible that the overall code distribution remains the same.

We briefly investigated whether code reviews could impact the evolution of the overall code distribution. Based on the commit messages, ConQAT and af3 perform code reviews regularly. ConQAT shows a positive evolution trend, af3, in contrast, one of the stronger negative trends. Hence, we cannot derive any hypothesis here. For the other systems, the commit messages do not indicate a review process but future work and developer interrogation is necessary to verify this.

4) Conclusion: With respect to the evolution of the code distribution in green, yellow, and red methods, we were not able to show a general negative trend, *i. e.*, no increasing risk for maintenance. Instead, it depends on the case study object.

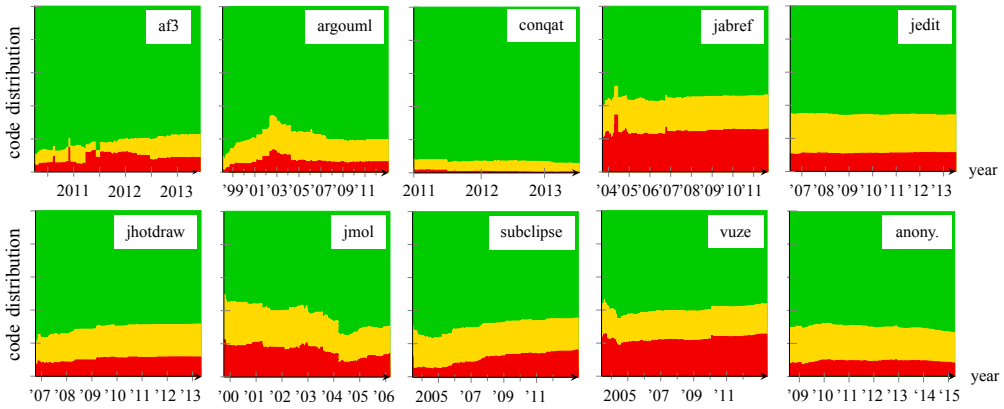


Fig. 5: Evolution of the code distribution over green, yellow, and red methods

VII. Summary and Discussion

The findings from our empirical study on software growth can be summarized as follows: Most yellow and red methods are already born yellow or red rather than gradually growing into yellow/red: Two out of three yellow methods were already yellow at their initial commit and more than half of the red method were born red. Furthermore, about half of the methods are not touched after their initial commit. On average, a method is only touched two to four times during history. This average holds for all study objects with low variance. Hence, for most methods the length classification within the thresholds is determined from the initial commit on. However, if a method is modified frequently, the length does grow: among all methods touched at least ten times, every third methods grows from a yellow to a red method. Nevertheless, over the entire system, the probability of a method to grow is low even though the system itself is growing in size. This can be explained by growth through newly added methods rather than by growth in existing methods. Finally, the distribution of the code in green, yellow, and red methods evolves differently for the case study systems. This distribution corresponds to the probability distribution that a developer has to understand and change a yellow or red method when performing a change on a random source code line. In our case study objects, we could not observe that the probability to change a yellow or red method generally increases. Generally, our case study did not show a significant difference between the industry and the open source system. However, we only analyzed one industry system. Hence, this claim cannot be generalized.

Implications. The implications of our work are based on two main results: First, most systems grow by adding new methods rather than through growth in existing methods. Second, existing methods grow significantly only if touched frequently. Hence, we derive the following recommendations—while differentiating between the benefits of short methods for readability, testability, reusability, and profile-ability on the one side, and changeability on the other side.

First, to obtain short methods that are easy to read, test, reuse and profile, developers should focus on an acceptable method length before the initial commit as we have shown that methods are unlikely to be shortened after. Even if these methods are not changed again, they will most likely still be read (when performing other changes), tested, and profiled (unless they become unmaintained code that could be deleted, of course). Given that the overall percentage of code in yellow and red methods reaches up to 50% in some case study objects (see Figure 5), the method size remains a risk factor that hampers maintainability even in the absence of further changes.

Despite knowing that many long methods are already born long, we yet do not know *why* they are born long. Do developers require pre-commit tool support to refactor already in their workspace, prior to commit? Or what other support do they need? Future work is required to obtain a better understanding.

Second, when refactoring methods to obtain a changeable code base, developers should focus on the frequently modified hot spots as we have shown that methods grow significantly when being touched frequently. When developers are modifying a frequently changed method, they have to understand and retest it anyway. Hence, refactoring it at the same time saves overhead. Furthermore, the method's change frequency over the past might be a good predictor for its change frequency in the future. However, future work is required to investigate this.

To summarize, developers should focus on frequently modified code hot spots as well as on newly added code when performing refactorings. This paper supports our hypothesis in [26], in which we showed that improving maintainability by prioritizing the maintenance defects in new code and in lately modified code is feasible in practice.

VIII. Threats to Validity and Future Work

Internal validity. Our work is based on a threshold-categorization of methods into green, yellow, and red methods. This obfuscates a growth of a method within its category, *e. g.*, a method length increase from 30 to 74 statements, affecting the results of RQ1, RQ3, RQ5. We introduced two thresholds to measure significant growth (such as a yellow-red transition)

in contrast to minor length increases of few statements. Setting the thresholds remains a trade-off between obfuscating some growth versus including minor length increases. However, we used these thresholds as they have gained common acceptance in industry among hundreds of customers' systems of the software quality consulting company CQSE.

Construct validity. The validity of our results depends on the accuracy of our origin analysis. Based on a thoroughly evaluated origin analysis for files [12], the adapted origin analysis for methods was evaluated manually by the authors with a large random sample. By manual inspection, we were able to validate only the precision, *i. e.*, that the detected method refactorings were correct. We were not able to validate the recall on the case study objects. We did verify it on our artificial test data set. However, for our case study, a lower recall would mean that we do not record some method extractions but treat the extracted methods as new methods. This does not affect the results of RQ1 and RQ5. It would affect RQ2–RQ4 but previous work has shown that overall absolute number of method refactorings is not very high [22] so we assume that our results are valid. Further, we measured the method length in number of statements. One could argue that comments should be included. However, as the overall comment ratio and also the amount of commented out code varies greatly (see [27]), we believe including comments would add unnecessary noise to our data and make the results less comparable.

External validity. We conducted our empirical case study only on ten Java systems. To generalize our results, the case study needs to be extended to other OO systems as well as to test code. We have first indications that C/C++ and CSharp results are similar but they require future work to be published.

IX. Conclusion

Many software systems contain overly long methods which are hard to understand, change, test, and profile. While we technically know, *how* to refactor them, we do not know *when* we should refactor them as we had little knowledge about how long methods are created and how they evolve. In this paper, we provide empirical knowledge about how methods and the overall system grow: We show that there is no significant growth for existing methods; most overly long methods are already born so. Moreover, many methods remain unchanged during evolution: about half of them are not touched again after initial commit. Instead of growing within existing methods, most systems grow by adding new methods. However, if a method is changed frequently, it is likely to grow. Hence, when performing refactorings, developers should focus on frequently modified code hot spots as well as on newly added code.

X. Acknowledgements

We deeply thank Andy Zaidman for his excellent reviews.

References

[1] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transaction on Software Engineering*, 2001.

[2] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008.

[3] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," in *Int'l Conference on Software Engineering*, 2006.

[4] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized Code Quality Benchmarking for Improving Software Maintainability," *Software Quality Control*, vol. 20, no. 2, 2012.

[5] I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability," in *Int'l Conference on the Quality of Information and Communications Technology*, 2007.

[6] M. Gladwell, *The Tipping Point: How Little Things Can Make a Big Difference*. Back Bay Books, 2002.

[7] E. Ammerlaan, W. Veninga, and A. Zaidman, "Old Habits Die Hard: Why Refactoring for Understandability Does Not Give Immediate Benefits," in *Int'l Conference on Software Analysis, Evolution and Reengineering*, 2015.

[8] G. Robles, I. Herraiz, D. German, and D. Izquierdo-Cortazar, "Modification and developer metrics at the function level: Metrics for the study of the evolution of a software project," in *Int'l Workshop on Emerging Trends in Software Metrics*, 2012.

[9] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 3, 2006.

[10] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," in *Int'l Conference on Software Maintenance*, 2000.

[11] L. Schulte, H. Sajnani, and J. Czerwonka, "Active Files As a Measure of Software Maintainability," in *Int'l Conference on Software Engineering*, 2014.

[12] D. Steidl, B. Hummel, and E. Juergens, "Incremental Origin Analysis of Source Code Files," in *Working Conference on Mining Software Repositories*, 2014.

[13] M. Godfrey and Q. Tu, "Growth, Evolution, and Structural Change in Open Source Software," in *Int'l Workshop on Principles of Software Evolution*, 2001.

[14] ——, "Tracking Structural Evolution Using Origin Analysis," in *Int'l Workshop on Principles of Software Evolution*, 2002.

[15] V. Arnaoudova, L. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc, "REPENT: Analyzing the Nature of Identifier Renamings," *IEEE Trans. on Software Engineering*, vol. 40, no. 5, 2014.

[16] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, 2007.

[17] F. Van Rysselberghe and S. Demeyer, "Reconstruction of Successful Software Evolution Using Clone Detection," in *Int'l Workshop on Principles of Software Evolution*, 2003.

[18] R. Peters and A. Zaidman, "Evaluating the Lifespan of Code Smells Using Software Repository Mining," in *European Conference on Software Maintenance and Reengineering*, 2012.

[19] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," in *Int'l Conference on Software Engineering*, 2015.

[20] A. Chatzigeorgiou and A. Manakos, "Investigating the Evolution of Code Smells in Object-oriented Systems," *Innovations System Software Engineering*, vol. 10, no. 1, 2014.

[21] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software Quality Control in Real-Time," in *Int'l Conf. on Software Engineering*, 2014.

[22] Dig, Danny and Comertoglu, Can and Marinov, Darko and Johnson, Ralph, "Automated Detection of Refactorings in Evolving Components," in *European Conference on Object-Oriented Programming*, 2006.

[23] D. Steidl and S. Eder, "Prioritizing Maintainability Defects by Refactoring Recommendations," in *Int'l Conf. on Program Comprehension*, 2014.

[24] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, 2011.

[25] P. Mohagheghi and R. Conradi, "An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes," in *Int'l Symposium on Empirical Software Engineering*, 2004.

[26] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler, "Continuous Software Quality Control in Practice," in *Int'l Conference on Software Maintenance and Evolution*, 2014.

[27] D. Steidl, B. Hummel, and E. Juergens, "Quality Analysis of Source Code Comments," in *Int'l Conference on Program Comprehension*, 2013.

7.3 Prioritizing External Findings Based on Benefit Estimation

The paper "Feature-based Detection of Bugs in Clones" (Paper **C**, [87]) contributes to the *automatic recommendation* step of the approach. In particular, it contributes to the classification of *external findings*.

Goal Amongst many static analysis findings, clones are very relevant as duplicated code increases the maintenance effort and are error-prone: Clones bear the risk of incomplete bugfixes when the bug is fixed in one code fragment but at least one of its copies is not changed and remains faulty. As we find incompletely fixed clones in almost every system, we aim to provide an approach to automatically classify inconsistent (gapped) clones as faulty or non-faulty: we evaluate in how far certain features of clones can be used to automatically identify incomplete bugfixes to help developers and quality engineers.

Approach We evaluated which factors induce bugs in cloned code and in how far we can speed up the detection process of incompletely fixed bugs by using an automatic clone classification. We often observe that clones with incompletely fixed bugs have specific features—for example, they have only few differences (gaps). For clones with multiple gaps, the differences are much more likely to be intentional. Other examples comprise an additional `!=null` check or a comment preceding the gap and indicating a bug fix. To detect which clone features are relevant for predicting bugs, we employ a machine learner which labels each inconsistent clone as *faulty* or *non-faulty*.

Evaluation We evaluated the machine learner with cross-validation on the existing clone data set from [49]. The data set comprises three industrial CS systems and one open-source system in Java. For the inconsistent clones of these systems, the data set contained expert ratings from the developers; for each clone, the developers judged whether the inconsistency was intentional or unintentional and, in the latter, faulty or non-faulty.

Results The machine learning classifier achieved a precision of 22% for clones with faulty inconsistencies which is better than manual inspection in random order. The results provide some insights about successes and limitations of a feature-based approach: The approach is well-suited to correlate syntactic properties of cloned fragments with the existence of incomplete bugfixes. However, we observed that the presence of an incomplete bugfix strongly depends on implicit context information that cannot be expressed in features. Hence, the precision was not significantly higher than manual inspection chances.

Thesis Contribution In the context of this thesis, the classifier aims to detect faulty inconsistent clones as *external* findings for the automatic recommendation as described in Section 6.2.2. The inconsistent clones serve as an example for which the internal/external classification cannot be done based on the findings category, but has to be done for each finding individually. However, as the precision is not very high, the paper reveals limitations of the automated detection of external findings. Instead, it shows that the manual inspection step is absolutely necessary in this case.

Feature-Based Detection of Bugs in Clones

Daniela Steidl, Nils Göde
CQSE GmbH, Germany
{steidl, goede}@cqse.eu

***Abstract*—Clones bear the risk of incomplete bugfixes when the bug is fixed in one code fragment but at least one of its copies is not changed and remains faulty. Although we find incompletely fixed clones in almost every system, it is usually time consuming to manually locate these clones inside the results of an ordinary clone detection tool. In this paper, we describe in how far certain features of clones can be used to automatically identify incomplete bugfixes. The results are relevant for developers to locate incomplete bugfixes—that is, defects still existing in the system—and for us as clone researchers to quickly find examples that motivate the use of clone management.**

***Index Terms*—Software quality, code clones, bug detection**

I. Introduction

The continuous research and improvement of clone detection algorithms has led to a number of industrial strength approaches, for example [1]–[3], that can detect not only identical code fragments but also similar code fragments with minor differences. These clones are commonly referred to as *inconsistent* clones or *gapped* clones. Figure 1, taken from [2], shows such an example. Multiple studies, for example [2], [4], [5], have shown that these differences are often unintentional and may sometimes point to bugs, which have been fixed only partially. That is, the bug was fixed in some of the fragments, but at least one of the copies was not changed and the bug remained effectively in the system.

As quality experts, we are frequently confronted with large industrial systems from domains such as automotive systems, energy controls, or insurance business. Our clone analysis, an integral part of our quality assessment, shows that incomplete bugfixes exist in almost every system. However, it is unclear which factors induce bugs in cloned code. Bugs in clones are relevant for both developers and researchers. Clones that resemble incomplete bugfixes point to bugs that still exist is software systems and may cause sever damage. Hence, detecting them allows to reduce the number of defects in software systems. Besides improving correctness, incomplete bugfixes are also well-suited examples for clone researchers to motivate the use of clone management. Finding clone-related bugs helps us as researchers and quality experts to argue why clone management should be an integral part of quality control.

Our experience tells us that incompletely fixed clones exist in almost every system. Locating them, however, is a time-consuming process. A number of approaches have been suggested to sort clones according to the probability of containing a bug. However, none of these approaches was evaluated with respect to how effective the sorting really is. Therefore, we are currently left with manual inspection of the clones in random order. Given the number of clones that is usually returned by an ordinary clone detection tool, the inspection takes long since incomplete bugfixes are usually found in only a small percentage of all detected clones [1], [2].

Problem Statement. *Software contains a large number of clones. Currently, there exists no algorithm to efficiently detect gapped clones that contain an incompletely fixed bug as it is unclear which factors induce bugs in cloned code.*

In this paper, we evaluate which factors induce bugs in cloned code and in how far we can speed up the detection process of incompletely fixed bugs by using an automatic clone classification. We often observe that clones with incompletely fixed bugs have specific features.[1] For example, these clones often have only few differences (gaps). For clones with multiple gaps, the differences are much more likely to be intentional. To detect which clone features are relevant for predicting bugs, we employ machine learning algorithms. Please note that we do not think such an algorithm can provide a perfect classification, but any automated classification with higher precision than manual inspection in random order would be beneficial.

Contribution. *We examine which clone features are relevant to predict incompletely fixed bugs.*

Outline. This paper is structured as follows. Section II presents previous work which is related to ours. Section III describes the data used for machine learning. We explain the relevant features of clones in Section IV. Section V, Section VI, and Section VII outline the machine learning algorithm we use along with the evaluation technique. Section VIII presents our results which are discussed in Section IX. Section X contains threats to validity, our conclusions are given in Section XI.

II. Related Work

In this section, we present previous work, which is related to ours. We limit ourselves to publications that investigate the relation between clones and bugs as part of a case study. For a general overview of clone research, please refer to existing surveys—e. g., [6] and [7].

Li et al. [4] detected copy-paste related bugs based on the consistency of identifiers in clone pairs. In particular, they calculated the *UnchangedRatio* that describes to which degree

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

[1] In this paper we use the term *feature*, taken from the machine-learning terminology, to describe characteristics of code clones.

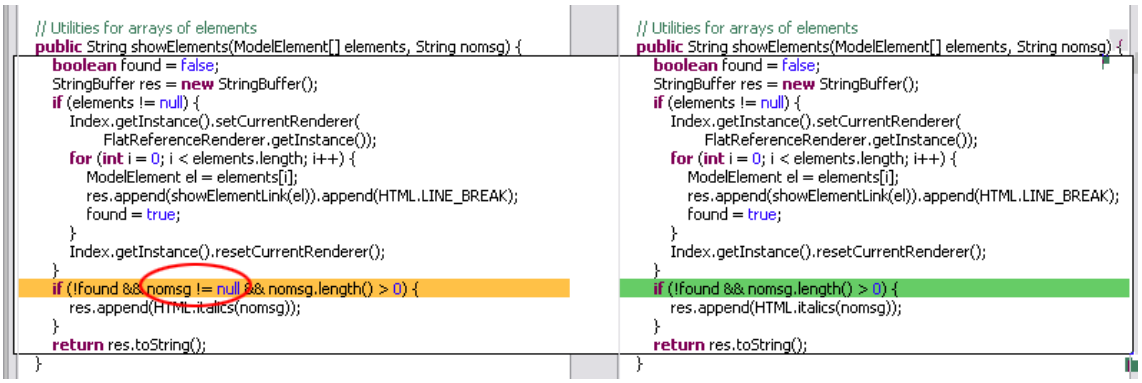


Fig. 1. Example of an inconsistent clone

identifiers are consistent. Their hypothesis is the lower the *UnchangedRatio* (the more identifiers have been changed), the more likely the clone pair contains a bug—except for when the *UnchangedRatio* is 0. Although Li et al. use the *UnchangedRatio* to prune the results and sort clone pairs according to their likelihood of having a bug, there is no empirical investigation of how useful the *UnchangedRatio* really is to identify clones with bugs.

An alternative approach to detect bugs based on inconsistencies has been presented by Jiang and colleagues [8]. Instead of inconsistencies between the cloned fragments themselves, inconsistencies in the contexts of the cloned fragments have been used to locate bugs. Jiang et al. identified different types of context inconsistencies which potentially indicate whether the inconsistency is a bug or not. But again, the usefulness of the types of inconsistencies has not been empirically evaluated.

Higo et al. [5] presented another approach to detect bugs based on inconsistencies. Clones which might contain a bug were extracted by subtracting the results of a more restrictive detector from those of a more tolerant clone detector to extract certain types of clones which potentially contain a bug. Whether these particular clones are more likely to contain a bug than others has not been investigated.

An elaborate study on bugs in inconsistent clones has been conducted by Juergens et al. [2]. For systems from different domains, experts rated whether a given inconsistent clone contained a bug or not. The results show that a notable number of inconsistencies in clones are actually unintentional and contain a bug. However, apart from the inconsistency, no other features of the clones have been analyzed according to whether they might help to find bugs.

In summary, all of the previous studies identified bugs in clones, using a particular pre-sorting based on certain features to identify clones likely to contain a bug. However, no previous work investigated how good the pre-sorting actually is. The usefulness of certain clone features to categorize the clone as defective or not is unkown. In this paper, we extend previous research by establishing a list of features motivated by earlier work and our practical experience and use machine learning to analyze the usefulness of these features.

TABLE I
SUBJECT SYSTEMS

Name	Organization	Language	Age [Years]	Size [kLOC]
A	MunichRe	C#	6	317
B	MunichRe	C#	4	454
C	MunichRe	C#	2	495
Sysiphus	TUM	Java	8	281

III. Case Study Data

In the following, we describe the training data used to predict incompletely fixed bugs in gapped clones. To obtain reasonable results, it is important to have a sufficiently large set of clone classes where experts for the corresponding code determined if they contain an incompletely fixed bug. Since this process is very time-intensive for us and the experts, we reuse data of a previous study [2], which had a different goal, but nevertheless collected data suitable for this approach. These data include the code base of the subject systems, the detected clones, and the experts' rating.

Systems. The training data comprises five systems studied in [2]. Due to our syntax-based classification, we excluded the COBOL system because of the significant syntactic differences between COBOL and the C-like languages C# and Java. The remaining subjects are three systems developed by different organizations for the *Munich Re Group*—one of the largest re-insurance companies in the world—and a collaboration environment for distributed software development projects, *Sysiphus*, developed at the Technische Universität München (TUM). Details about the systems are given in Table I.

Clone Detection. We also reused the clones from the former study, which were detected using our tool ConQAT.[2] Generated code was excluded prior to the detection. The clone detection algorithm is in principle token-based. However, the individual tokens of each statement were merged into a single artificial token representing that statement. Hence, detection is based on the sequence of statements rather than the sequence

[2]www.conqat.org

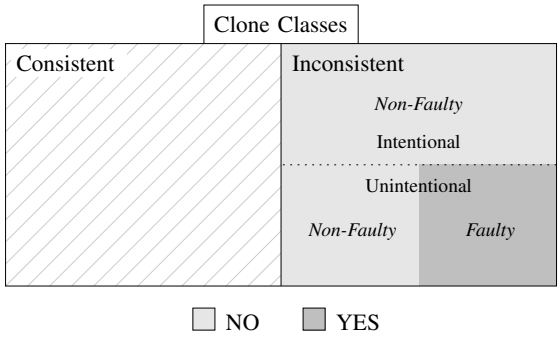


Fig. 2. Rating of clone classes as faulty and non-faulty

of original tokens. Identifiers and literals have been normalized except for regions of repetitive code. Clones were not allowed to cross method boundaries. The minimal clone length was set to 10 statements. For a more detailed description of the algorithm and its parameters, please refer to [2].

Rating. We also reused the developers' rating of the clones from the previous study. The detected clone candidates were presented to the developers who rated them in three steps:

1) The developers removed false positives of the clone detection, i. e., code fragments without semantic relationship although the tool detected them as clones. From the true positives, we removed clones without inconsistencies (gaps).
2) For inconsistent clones, the developers rated the inconsistencies as unintentional or intentional.
3) If a clone class was unintentionally inconsistent, developers classified it as faulty or non-faulty, or don't know in case their decision was unsure. Intentional inconsistent clones are non-faulty by definition.

Figure 2 visualizes the sets of different clones. In cases where developers could not determine intentionality or faultiness, e. g., because none of the original developers was available for rating, the inconsistencies were treated as intentional and non-faulty. We used the developers' answers from the third step to assign each clone class one of the following labels:

- **YES.** Differences between gapped clones were unintentional and faulty: the clone class contains an incompletely fixed bug (dark gray in Figure 2).
- **NO.** The differences between the cloned fragments are on purpose or were unintentional but do not contain an incompletely fixed bug (light gray in Figure 2).

Data Size. In total, the developers rated 582 gapped clones, among which 104 clones were labeled with *YES*, i. e., contained an incompletely fixed bug. 456 clones were labeled with *NO*. We excluded 22 clones from the data set which were labeled with *don't know*. The resulting data set is an *imbalanced data set* as one class (with label *NO*) is highly overrepresented compared to the other class (with label *YES*). We describe in Section VI how we handle the imbalanced data set in our approach.

IV. Features

Our hypothesis is that we can identify incompletely fixed bugs based on certain features of gapped clones. Therefore, we evaluate in how far this hypothesis holds for a given set of features. The features we use originate from recurring patterns that we observed during many years of manually inspecting gapped clones. This experience includes but is not limited to the clones in our training data set. Please note that the set of all possible features is infinite and our selection provides only a first starting point. However, our analysis can be repeated for other features with only little effort. The remainder of this section describes the selected features that turned out to be useful for machine learning and documents our assumptions why we believe a feature is relevant for predicting faulty inconsistencies.

We use features to describe both the whole content of cloned fragments (*global* features) as well as only the gaps between the clones (*local* features).

A. Global Context Features

- **length (Integer)** – *Length of copied code fragment ignoring gaps.*
  Bugs are more likely to occur when there are differences within long similar code sequences compared to only short fragments.
- **nesting_depth (Integer)** – *Maximal nesting depth of a code fragment.*
  Faulty inconsistencies between code fragments are more likely to occur in algorithmic code with higher nesting depth than in data code.
- **comment (Boolean)** – *Any gap is preceded by a comment.*
  Often developers comment on a gap when fixing a bug and denote an issue-id or a short bug-fix discription. Hence, we assume commented gaps to indicate an incomplete bugfix.
- **preceded_by_assignments (Boolean)** – *Any gap is preceded by two assignment statements.*
  It is easy to forget a single assignment while writing a set of assignments, e. g., setting various attributes of an object. Hence, a gap preceded by a set of assignments can indicate a faulty inconsistency. We use the simple heuristic to detect assignments by searching for lines containing = but not `if`, `else`, `for`, and `while`.

B. Local Gap Features

- **attribute (Boolean)** – *Any gap contains an attribute declaration.*
  An attribute declaration seems to be a major change, indicating that the change was on purpose.
- **new (Boolean)** – *Indicator for any gap containing the keyword* `new`.
  The keyword `new` signals the creation of an object. We assume the creation of a new object to represent a major change on purpose.

- **equal_null (Boolean)** – *Any gap contains a condition including* `== null`.
  While writing `if`, `for`, or `while` statements, it is easy to forget a null check. Hence, we believe a gap containing a null check indicates an incomplete bugfix.
- **not_equal_null (Boolean)** – *Any gap contains a condition including* `!= null`.
  Analogous as the previous feature, we believe that an easily forgotten `!= null` check indicates a faulty inconsistency.
- **continue (Boolean)** – *Any gap contains the keyword* `continue`.
  During the implementation of loops, developers easily forget necessary continues. We assume this indicates a faulty inconsistency.
- **break (Boolean)** – *Any gap contains the keyword* `break`.
  During the implementation of loops, developers easily forget necessary breaks. We assume this indicates a faulty inconsistency.
- **num_token_type (Integer)** – *Number of different token types in the gaps.*
  We believe inconsistencies are more likely to occur in algorithmic code than in data code with fewer token types than algorithmic code.
- **method_call (Boolean)** – *Any gap matches the method call pattern*
  We use `[a-zA-Z]+\.[a-zA-Z]+\(.*\)` as a regular expression for a method call and assume that a method call in a gap indicates a bugfix.

C. Feature Implementation Details

Both global and local features of gapped clones contain boolean and integer features. In general, machine learning algorithms are applicable not only for feature representations with a unique type (e. g., an integer vector) but also for feature representations with mixed types (e. g., a feature vector with booleans and integers). However, including integer features with a broad range of possible values (e. g., the *length* feature with a value range of [*minimal clone length*..∞]) in a feature vector with mostly boolean features causes problems during machine learning. The classifier predominantly uses this feature to achieve a high precision/recall: The classifier makes a decision by splitting the large feature value range into very small subintervals such that most subintervals contain exactly one training data point. Consequently, the classifier separated the data set and does not need to consider other features. Solely based on this large-range integer feature, the classifier performs reasonably well. However, it strongly overfits the training data and cannot be generalized.

To avoid this phenomenon, we manually limit the number of different values for integer features that have a broad value range, i. e., the *length* and *num_token_type* feature. (The *nesting_depth* feature, in contrast, reveals a small value range in our data set with most values being in the interval $[0..4]$ and a barely occurring maximum value of 6. Hence, the chance of overfitting based on this feature is low.)

For the *length* features we split the value range as follows: [*minimal clone length*..20] is mapped to value 0, $[21..\infty]$ to 1. For the *num_token_type* feature, values in $[0..5]$ are transformed to 0, $[6..10]$ to 1 and $[11..\infty]$ to 2. The feature value transformations were obtained based on preliminary experiments and manual inspection of the data set.

V. Decision Trees

We assume that the features outlined in the previous section are relevant to classify a gapped clone as incomplete bugfix or not. Using a machine learning algorithm, we analyze how relevant each feature truly is. In this paper, we use *decision trees* as classifier and the standard machine learning library WEKA[3] for implementation.

A decision tree is a tree with decision nodes as interior nodes and class labels as leaves. To classify an instance, the decision tree is a set of `if-then-else` statements: Based on the value of a single feature, a decision is made at each interior node to further traverse the left or the right branch of the tree until a leaf is reached. The class label represented by the leaf is assigned to the instance as its classification.

After preliminary experiments comparing the performance of various different classifiers (support vector machines, naives bayes, AdaBoost, etc.) we chose decision trees due to their results being the most promising and easy to understand: the result of a decision tree can be easily visualized and understood by humans as opposed to, e. g., the high-dimensional hyperplanes of support vector machines. In particular, we used the J48 implementation of a decision tree [9].

VI. Cost-Sensitive Training

As described in Section III, our underlying training data set is imbalanced. This is problematic as machine learning algorithms operate under two assumptions:

1) The goal is to maximize the prediction accuracy.
2) The test data has the same underlying distribution as the training data.

In our data set, 18.6% of the data is labeled with *YES* (minority class), 81.4% is labeled with *NO* (majority class). Consequently, if the classifier predicted label *NO* for all data, it would already achieve an accuracy of 81.4% by default without learning from the data.

Machine learning research has proposed a variety of methods to overcome issues with imbalanced data sets, [10], i. e., artificially sampling the data set to balance class distributions or adapting the learning algorithms: To improve the data set, upsampling approaches (interpolating more data samples of the minority class) or downsampling approaches (removing samples of the majority class) exist. In our case, the naive approach of randomly downsampling (using the WEKA SpreadSubsample) did not succeed as the results strongly varied with the randomly chosen negative data points and hence overfitted the data. More sophisticated approaches to

[3] http://www.cs.waikato.ac.nz/ml/weka/

TABLE II
COST MATRIX FOR FALSE POSITIVES AND FALSE NEGATIVES

Classified as →	*YES*	*NO*
YES	0	5 (FN)
NO	1 (FP)	0

upsample the data (e. g., with the SMOTE algorithm [11]) did not significantly improve the results.

Instead of balancing the distribution of the data by sampling, we used cost-sensitive training to adapt the learning algorithm to the imbalanced data set: Cost-sensitive learning addresses imbalanced data by using cost matrices describing the costs for misclassifying any particular data example [10]. We used the WEKA CostSensitiveClassifier in combination with a decision tree to reweight the data points according to the cost matrix in Table II which was obtained by preliminary experiments. The matrix indicates that false negative mistakes are penalized five times more than false positive mistakes which makes the classifier more sensitive to the minority class of the training data. For more detailed information about cost-sensitive training, refer to [10] and the WEKA documentation.

VII. EVALUATION METHOD

We evaluate the cost-sensitive decision with k-fold cross-validation [12], using precision and recall. Precision and recall are calculated for each label l separately and then (weighted) averaged over all labels. As we use the experts' rating (*YES* or *NO*) the classification problem is binary.

Precision and recall. Pecision represents the probability that the classifier makes a correct decision when predicting that the instance is labeled with l. Recall denotes the probability that an instance with label l is detected by the classifier.

For this approach we are particularly interested in the precision for the positive label (*YES*): The algorithm's goal is to detect clones that contain incompletely fixed bugs. To achieve this, we are currently left with manual detection. For a quality engineer, finding incomplete bugs in an unfamiliar code base corresponds to inspecting code clones in a random order. Hence, the probability for the quality engineer to find an incomplete bugfix is equivalent to the relative frequency of incomplete bugfixes among all gapped clones. Hence, if the algorithm has a higher precision for the positive label than visiting clones in random order, it speeds up the process of retrieving incomplete bug fixes. For the scenario of a beginning continuous control process, precision is more important than recall to convince developers of the necessity of clone management. In our training data set, the relative frequency of incomplete bugfixes is 18.6% ($= 104/560$). Hence we aim for a precision above 19%.

Cross validation. The overall evaluation of the algorithm is based on k-fold cross validation, a standard evaluation technique in machine learning to determine the quality of a classifier [12]: The training data set is split into k subsets.

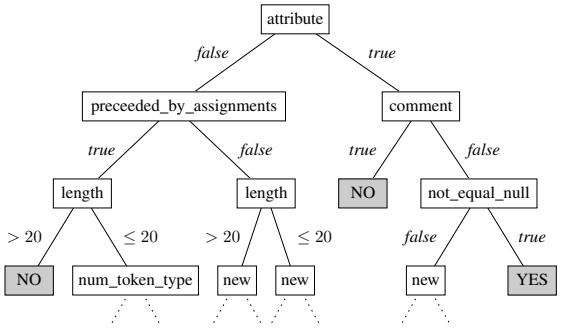


Fig. 3. Resulting classification model for bug detection

The classifier is built k times with $k-1$ out of k subsets as training data. For each run, the remaining subset is used as test data and the classifier's precision and recall are calculated on this fold. To get an overall evaluation, precision and recall are averaged over all k runs. Common values for k are 5 or 10. In this paper, we only present results of 5-fold cross validation as they did not differ significantly from 10-fold cross validation.

VIII. RESULTS

This section describes the results of our approach obtained by applying a decision tree algorithm on the real-world data set presented in Section III with the features described in Section IV. We first interpret the classification model learned and show precision, recall, and classification errors based on the results from cross-validation.

A. Classification Model

Figure 3 shows the top levels of the resulting decision tree. The complete tree with a maximum depth of ten is not included in this paper due to length restrictions. The visualization of the classification model provides insights about the most relevant features for bug identification. The closer a feature is located to the root of the decision tree, the more information gain it provides for clone classification. The feature used for the root is the local feature *attribute* indicating that at least one gap contains an attribute declaration. At the next depth levels, the tree uses the features *preceded_by_assignments* and *comment* (depth 1), *length* and *not_equal_null* (depth 2), and *new* and *num_token_types* (depth 3).

In the following, we present a simplified, top-level interpretation of the decision nodes: The existence of an attribute declaration in any gap indicates a non-faulty gap if the gap was commented. If no gap contained an attribute declaration, but a gap was preceded by variable assignments, the decision tree mostly decides for a faulty gap. If neither the *attribute* nor the *preceded_by_assignments* features are true, the decision tree splits the data based on the *length* and the *new* feature. In most of the cases, if a new object was created in any gap, the tree classifies the majority of data points as faulty.

B. Classification Evaluation

Table III shows the classification's confusion matrix. Rows represent the real label of the data, columns indicate the

TABLE III
CONFUSION MATRIX

Classified as →	*YES*	*NO*
YES	62	42
NO	218	238

TABLE IV
RESULTS OF J48 FOR CLONE CLASSIFICATION

Class	TP Rate	FP Rate	Precision	Recall
YES	0.596	0.478	0.221	0.596
NO	0.522	0.404	0.850	0.522
Weighted Average	0.536	0.418	0.733	0.536

label assigned by the classifier. Hence, entries on the matrix diagonal are correct classifications, other entries indicate classification errors. In total, the classifier labels 300 instances correctly (54% of all data), 62 clones with incomplete bugfixes and 238 clones without bug. The classifier did not detect 42 incomplete bugfixes and misclassified 218 clones without bug.

Table IV shows the classification's precision, recall, true-positive rate, and false-positive rate which are calculated for each label individually and summarized in a weighted average over all instances (weights for each label correspond to the relative frequency of the label within the training data). Precision is calculated as the number of true positives over true positives plus false positives, recall as number of true positives over true positives plus false negatives.

For clones with incomplete bugfixes, the algorithm achieves a precision of 22.1% ($= \frac{62}{62+218}$), better than the target precision of 18.6% (see Section VII). The recall of 59.6% ($= \frac{62}{62+42}$) indicates that the classifier detects more than every other incomplete bugfix. For clones without bug, precision is 85%, recall 52%. However, these metrics were not the primary target of our approach. In total, the classifier results in a weighted average precision of 73% and recall of 53%.

We also used a 10-fold cross validation and ran both the 5-fold and 10-fold cross validation multiple times with different random seeds but did not detect a significant change in the output.

IX. DISCUSSION

The results revealed that the machine learning approach achieves a higher precision for incompletely fixed bugs than manual detection in an unfamiliar code base where clones are inspected in random order and hence, the probability to find an incomplete bug fix corresponds to the relative frequency of incomplete bug fixes in the data set. However, our experiments also show limitations of feature-based approaches which are discussed in this section. First, we enumerate additional features that were not beneficial for bug prediction. Second, we show why not all information relevant for bug detection can be captured in features. Third, we discuss the challenges in obtaining a data base for classification.

A. Excluded Features

Initially, we experimented with more features than presented in Section IV. However, including some features in the data representation caused a lower performance of the classifier—a phenomenon frequently seen during machine learning. Nevertheless, it is valuable information which features do not provide information for detecting incompletely fixed bugs in clones. The following list describes our initial assumptions more in detail and explains the consequences when adding a single feature to the feature set.

- **gap (Integer)** – *Total number of gaps in a clone class.* The more the code fragments differ, the less likely the clone class contains a bug. If there are only few differences, the probability of a bug is higher. Similar assumptions have been stated in previous work [4], [8]. Contrary to our assumption, our experiments showed that using the *gap* instead of the *length* feature or both together lowers precision and recall for faulty inconsistencies.
- **if / else (Boolean)** – *Any gap contains the keyword* `if` *or* `else`*.* We assumed that a forgotten `if` or `else` statement indicates an incompletely fixed bug. Both features reduce the performance of the decision tree and do not seem to have significant information gain. The *else* feature does not even occur in the pruned version of the decision tree, the *if* feature only at high depths close to the leaves.
- **boolean_check (Boolean)** – *Any gap contains a == together with a boolean literal* Similar to the *not_equal_null* and *equal_null* features, we assumed that a forgotten condition in a `if`, `while`, or `for` statement can indicate incomplete bugfixes. Adding or removing this feature from the data representation does not influence the result significantly. It neither provides information gain nor causes the decision tree to make wrong decisions.
- **boolean_assign (Boolean)** – *Any gap contains an = assignment of a boolean literal* We assumed that forgetting to assign a boolean variable with control structures such as `loop`, `while`, or `if` often causes bugs when the boolean variable is used as a control condition. Adding this features lowers precision and recall.

B. Limitations of Features

Experimenting with different sets of features provides some insights about successes and limitations of a feature-based approach. The feature-based approach is well-suited to correlate syntactic properties of cloned fragments with the existence of incomplete bugfixes such as a missing `!= null` condition.

However, we observed that the presence of an incomplete bugfix strongly depends on implicit context information that cannot be expressed in features. Often, it is hard for any human, who is not an active developer of the code base, to judge whether an inconsistency is unintentional or error-prone.

For example, the presence of a comment preceding a gap may indicate two things: Either a bug was fixed and

commented but forgotten in the other cloned fragment—indicating an incomplete bugfix—or the developer cloned code and commented on intentionally added new functionality—indicating an intentional inconsistency. A human might be able to differentiate both cases by referring to the natural language information of the comment. A feature-based representation, however, fails to capture this difference. In case the comment contains signal words such as "bug", "fix", or "added", natural language processing can be useful for classification, but this does not generalize to an arbitrary commenting style.

The presence of the keywords `if` or `else` is another such example that can either indicate an incomplete bugfix (forgotten `if` condition in one code fragment) or the intentional adaptation of a clone fragment to a different scenario. In such cases, it is even for a human hard to make a decision about the correctness without deep knowledge of the code.

C. Challenge of Data Collection

The success of any machine learning approach depends on the training data used. In particular, we experienced that the imbalanced class distribution of our data hinders further success. With random downsampling, we achieved a precision for the positive label up to 68% in some cases. However, this performance varied so strongly with the random sample that it cannot be generalized. Nevertheless, the partial success of the approach makes us believe that using automated clone classification has potential for future work if a better (larger and balanced) data set of inconsistent clones is available.

X. Threats to Validity

This section summarizes various factors that threaten the general validity of our results. First of all, our choice of features was based on syntactic properties of C-like languages like Java, C++, and C#. Therefore, we cannot generalize our results to all programming languages.

Second, we mainly focused on the application of our approach in the domain of quality control to motivate the use of clone management. Hence, we argued that the precision of the positive label is the most important evaluation aspect. For other use cases, which include the retrieval of remaining bugs in the system, recall is also crucial. Future work should study features that improve the recall while maintaining precision.

Third, using experts' ratings as evaluation contains several threats already described in [2]. However, within this context, we rely on the experts' rating as no other evaluation method is available.

Furthermore, the configuration of the clone detection tool strongly influences detection results shown to the experts. Refer to the justification in [2], which is based on a pre-study and long-term experience with clone detection.

XI. Conclusion and Future Work

In this paper, we examined how much a feature representation of gapped clones can help to retrieve incomplete bugfixes in cloned code. We experimented with a broad range of features and used machine learning to determine their relevance in bug prediction. The machine learning classifier achieved a precision of 22% for clones with faulty inconsistencies which is better than manual inspection in random order.

For developers, our approach can be used as a recommendation tool for finding incomplete bugfixes to remove errors still remaining in the system. For the use-case of motivating clone management, our approach helps to quickly find faulty clones in our customers' code. We often observed that one or two examples of unintentional faulty inconsistent clones from the customers' own code can significantly support our argument to use clone management. In that sense, our approach is a first step to minimize the effort of retrieving convincing examples.

Our approach is a good starting point for future work as it has shown that feature-based bug detection and automated clone classification has a certain potential. The two most promising directions for future work are the creation of a larger data set for learning and the evaluation of other features. These may, for example, even include non-syntactic features like change frequency or authorship.

References

[1] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 311–320.

[2] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 485–495.

[3] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th International Conference on Program Comprehension*. IEEE Computer Society, 2008, pp. 172–181.

[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[5] Y. Higo, K. Sawa, and S. Kusumoto, "Problematic code clones identification using multiple detection results," in *Proceedings of the 16th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2009, pp. 365–372.

[6] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[7] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queens University at Kingston, Ontario, Canada, Technical Report, 2007.

[8] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2007, pp. 55–64.

[9] R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[10] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. on Knowl. and Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009.

[11] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[12] R. Kohavi, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, ser. IJCAI '95. Morgan Kaufmann, 1995, pp. 1137–1143.

7.4 Prioritizing Internal Findings Based on Cost of Removal

The paper "Prioritizing Maintainability Defects based on Refactoring Recommendations" (Paper **D**, [86]) contributes to the *automatic recommendation* of the approach. It demonstrates how to detect *internal* findings which reveal low removal costs.

Goal The paper aims to detect findings, in particular code clones and long methods, which are easy to refactor and, thus, cost very little time for the developers to remove. Additionally, we examined which external context information influences the decision to remove a single finding in industrial software development.

Approach The approach recommends code clones and long methods that are easy to refactor. We proceed in two steps: First, we detect the findings using a clone and a long method detector. Second, for both categories, we use different heuristics to detect findings that can be refactored easily. The selection is mostly based on a heuristic data flow analysis on source code. All heuristics are designed to match one of the two refactoring patterns *pull-up method* or *extract method*.

Evaluation We conducted a survey with developers from two industrial teams to find out if they would accept the suggested findings in their own code for removal. We evaluated which findings developers considered easy to remove and whether they would actually remove them. Further, we examined the limits of automatic finding recommendation by quantifying how much external context information is relevant to make the removal decision.

Results The survey showed that the heuristics recommend findings that developers consider to be easy to remove with high precision. Developers stated that 80% of the recommended findings were both easy and useful to remove. When developers would not have removed a finding, we analyzed what context information they used for their decision. For the global analysis of code clones, developers considered a lot more external context information than for the local analysis of long methods: Developers did not want to remove code clones mostly due to external reasons unrelated to the code itself (e. g. waiting for a general design decision or external factors causing the redundancy such as duplicated specifications). Developers rejected to shorten long methods mostly due to the nature of the code when the methods were not hard to understand, e. g. if they contained UI-code, logging code, or only filled data objects.

Thesis Contribution For this thesis, the paper serves as an example how to detect internal findings that can be removed with low removal costs. These findings are prioritized in the thesis's approach for systems with highly-dynamic maintenance as described in Section 6.2.3. The paper's results further provide context information for the manual inspection as described in Section 6.3.2. It provided guidelines to reject non-relevant findings that developers consider not worth to be removed. Beyond the scope of this paper, the idea to detect very-easy-to-remove findings can also be extended to other findings, e. g. long classes or deep nesting.

Prioritizing Maintainability Defects Based on Refactoring Recommendations*

Daniela Steidl
CQSE GmbH
Germany
steidl@cqse.eu

Sebastian Eder
Technische Universität
München
Germany
eders@in.tum.de

ABSTRACT

As a measure of software quality, current static code analyses reveal thousands of quality defects on systems in brown-field development in practice. Currently, there exists no way to prioritize among a large number of quality defects and developers lack a structured approach to address the load of refactoring. Consequently, although static analyses are often used, they do not lead to actual quality improvement. Our approach recommends to remove quality defects, exemplary code clones and long methods, which are easy to refactor and, thus, provides developers a first starting point for quality improvement. With an empirical industrial Java case study, we evaluate the usefulness of the recommendation based on developers' feedback. We further quantify which external factors influence the process of quality defect removal in industry software development.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*

General Terms

Management, Measurement

Keywords

Software Quality, Dataflow Analysis, Finding Prioritization

1. INTRODUCTION

Software systems evolve over time and without effective counter measurements, their quality gradually decays, making the system hard to understand and maintain [8, 17]. For easy program comprehension and effective maintenance, research and practitioners have proposed many different static code analyses. Analyzing the software quality, these analyses reveal quality defects in the code, which we refer to as *findings*. Static code analyses comprise structural metrics (file size, method length, maximal nesting depth), redundancy measurements (code cloning), test assessment (test coverage, build stability) or bug pattern detection. When quality analysis tools (such as ConQAT, Sonar, or Teamscale) are introduced to a software system that has been maintained over years, they typically reveal thousands of findings. We have frequently observed this in practice as software quality consultants of the CQSE, as we audited and monitored the quality of hundreds of systems in domains ranging from insurance to automotive or avionics. Also, this paper provides a benchmark showing that only two analyses (code duplication and method structure) already reveal up to a thousand findings each, in industry and open source systems.

Problem Statement. *Existing quality analyses reveal thousands of findings in a grown software system and developers lack a structured prioritization mechanism to start the quality improvement process.*

Confronted with a huge number of findings, developers lack a starting point for a long-term software improvement process. Often, they do not fix any findings because they do not know which findings are worth to spend the refactoring effort on. To actually start removing quality defects, developers need a *finding prioritization* mechanism to differentiate between the findings. Whether developers prioritize the removal of one finding over another depends on the expected return on invest. However, precisely estimating the ratio of expected costs for the removal and the expected gain in maintenance and program understanding is a difficult task. Often, the decision to remove a finding does not only depend on the finding itself but also on additional context information: for example, if the finding is located in critical code or, in contrast, in code that will be deleted.

We suggest to start removing defects which are easy to refactor, resulting in low expected costs for the removal – assuming that a quality defect which is easy to remove indicates an appropriate refactoring matching the existing code structure. In a case study on two industrial Java systems, we evaluate if developers agree to remove findings prioritized by our approach. We determine which additional context information developers consider in this decision.

Our analysis focuses on the two findings *code clones* and *long methods*, which are both maintainability defects that

*This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...$15.00
http://dx.doi.org/10.1145/2597008.2597805

do not necessarily contain bugs. Long methods threaten maintainability as they are hard to understand, reusable only at very coarse granularity, and result in imprecise profiling information. Duplicated code increases the effort and risk of changes as changes must be propagated to all instances and might be done inconsistently [14]. With the long method detection being a *local* analysis [1] and the clone detection a *global* one[1], we use two conceptually different analyses and evaluate if the scope of an analysis is reflected in the type of context information required for the removal decision.

Contribution. *We provide a prioritization mechanism for code clones and long methods based on expected low costs for removal and quantify which external context information influences the removal of quality defects.*

2. RELATED WORK

As we use refactoring recommendations to prioritize quality defects, our related work is grouped into the state of the art of refactorings and prioritizing quality defects.

2.1 Refactorings

In the following, we discuss refactorings [9] to remove code clones and long methods.

Refactoring Clones. The most common refactorings for clone removal in object-oriented programming languages are the pull up and extract method refactoring [7, 10–12]. All three approaches suggest, detect, or automatically perform refactorings. In contrast, our work provides an evaluation with developers of two industrial systems: we evaluate when refactorings of quality defects are actually useful and what context information developers need to decide in the removal decision. Existing approaches lack such an evaluation.

In [7], the authors provide a model for a semi-automatic tool support for clone removal. At a very early stage, they outline in a position paper when clones can be refactored in object-oriented languages. We use the same refactoring idea, but take it to the next level by providing an evaluation.

The approach in [10–12] automatically detects possible refactorings of code clones by the above two refactoring patterns. This approach uses similar data flow heuristics and static analyses to detect clones which can be refactored. However, the evaluation only demonstrates that the tool finds clones that can be refactored and shows the reduction rate on the Java systems Ant and ANTLR [10, 12]. In the case study of [11], the authors classify the refactored clones in different groups depending on how difficult the refactoring is but did not evaluate based on developer interviews. In contrast, we include a developer survey investigating which clones developers would or would not refactor.

Refactoring Long Methods. In [20], the authors focus on tool support to refactor long methods. They quantify in an experiment the shortcomings of the current extract method refactoring in Eclipse. Based on the observed drawbacks, they propose new tools and derive recommendations for future tools. In general, the paper focuses on the tool-support for refactoring. Our approach, in contrast, focuses on which findings developers remove first.

[1]A local analysis requires only information local to a single class, file, or method. A global analysis result of a file, however, can be potentially affected by changes in other files.

The approach in [19] automatically detects code fragments from Self programs that can be extracted from methods or can be pulled up in the inheritance hierarchy. The complete inheritance structure is changed to gain smaller, more consistent, and better reusable code. However, the authors only claim that most of their refactorings improve the inheritance structure. Concrete proof based on developers' opinions is missing. Our work, in contrast, examines when developers are actually willing to restructure a method.

The approach in [23] bases on test-first refactoring: test code is reviewed to detect quality defects which are then refactored (*e. g.*, with the extract method refactoring) in source and test code. Implicitly, the approach prioritizes findings that are found through examining the test code. In contrast, we examine how developers prioritize long methods in the source code without considering the test code.

2.2 Prioritizing Quality Defects

External Review Tool Results. [2, 3] propose an approach to prioritize findings produced by automatic code review tools. There, the execution likelihood of portions of code is calculated using a static profiler and then used to prioritize the findings. The authors focus on the applicability of their approach to detect likelihood of execution and not on the impact of their prioritization. In contrast, we use a different prioritization mechanism (low costs for removal) and evaluate whether developers from industry would accept our recommendation in practice.

An approach to prioritize warnings produced by external review tools (FindBugs, JLint, and PMD) is described in [15]. The authors use the warning category and a learning algorithm based on the warning's lifetime to prioritize with the assumption that the earlier a warning is addressed the more important it is. In contrast, we concentrate on the ease of removal of clones and long methods instead of review tools and we also focus on a specific instance of a quality defect rather than prioritizing its category. Another approach of prioritizing warnings is based on the likeliness of a warning to contain a bug [4] with a case study evaluating how well the bug prediction mechanism performs. In contrast, we focus on maintainability and not on functional correctness while providing information about which external factors influence the prioritization of findings.

Code Clones. The work in [24] uses a model for prioritizing clones including maintenance overhead, lack of software quality, and the suitable refactoring methods. Compared to us, this approach has a more complex mechanisms to prioritize clones. However, the authors do not indicate whether the resulting prioritization matches the developers' opinion. We, in contrast, evaluate with developers from two industry systems. Another approach to prioritize code clones is described in [5]: clones are summarized in work packages by transferring the problem of making work packages into a constrained knapsack problem, considering the time it takes to remove the clone. The authors show that their estimations of the clone removal time based are close to reality. As before, the prioritization of the approach is not validated. In [21], we focused on detecting bugs in inconsistent gapped clones, assuming that these should receive the highest prioritization. In this paper, we only focus on identical clones.

Table 1: Heuristics for Clones and Long Methods

	Heuristic	Refactoring	Sorting
Clones	Common-Base-Class	Pull-Up Method	*method-length* ↓
	Common-Interface	Pull-Up Method	*method-length* ↓
	Extract-Block	Extract Method	*block-length* ↓
Long Meth.	Inline-Comment	Extract Method	*num-comments* ↓
	Extract-Block	Extract Method	*block-length* ↓
	Extract-Commented-Block	Extract Method	*block-length* ↓
	Method-Split	Extract Method	*num-parameters* ↑

3. APPROACH

Our approach recommends findings that are easy to refactor to give developers a starting point for quality defect removal, focusing on two category of findings, *code clones* and *long methods*. We proceed in two steps: First, we detect the defects using a clone and a long method detector. Second, for both categories, we use different heuristics to determine where refactorings can be applied which is mostly decided based on a heuristic dataflow analysis on source code. All heuristics are designed to match one of the two refactoring patterns *pull-up method* or *extract method*. Table 1 gives a high-level overview of the heuristics and their corresponding refactorings, which will be explained later in this section.

Each heuristic provides a criterion to sort its recommended findings: the best recommendations are the top findings in the sorting – Table 1 shows the sorting for each heuristic. A threshold for each criterion can be used to cut off the bottom of the sorted list to limit the number of recommended findings. However, this threshold depends on the amount of findings a developer is willing to consider as his starting point as well as on the current quality status of the system: if the system is of low maintainability with many findings, the threshold in practice will be higher than for systems with few findings. For this paper, we have set the thresholds based on preliminary experiments. Future work is required to determine them with a thorough empirical evaluation.

3.1 Findings Detection

This approach uses a code clone and a long method detector, implemented within the quality analysis tool ConQAT [6]. Their configurations are described the following.

Clone Detection. We use the existing code clone detection of ConQAT [13] and configured it to detect *nearly* type-I clones[2]: We neither normalize identifiers, fully qualified names, type keywords, nor string literals. We do normalize integer, boolean, and character literals and visibility modifiers, comments, and delimiters. With this normalization, we detect almost identical clones – identical except of the above mentioned literals, modifiers, etc.[3]

Long Method Detection. We detect long methods by counting lines of code. We define a method to be *long* if it has more than 40 lines of code (LoC). The threshold results from the experience of the CQSE GmbH and recommendations from [18]: it is based on the guideline that a method should be short enough to fit entirely onto a screen.

[2]For definition of type-I, II, and III clones, refer to [16].

[3]We restrict our analysis to identical clones as we want to prioritize clones which are easy to refactor – assuming that type-I-clones are easier to refactor than type-II or type-III: Pulling up a type-II clone might, *e. g.*, require the use of generics if the clone contains objects of different types.

3.2 Dataflow Analysis on Source Code

To detect findings that can be refactored, we use a heuristic dataflow analysis on source code. We use a heuristic that performs without byte code, as we do not need source code that compiles. The heuristics provides a def-use analysis of all variables in the source code as the definition and the uses of variables determine whether code can be refactored.

The heuristic extracts all variables from the source code, differentiating between local and global ones using a shallow parser[4]. For each local variable, it searches for definition and uses, identifying reads and writes. A definition is detected when a type (*e. g.*, `int`, `Object`) is followed by an identifier (*e. g.*, `a` or `object`). A write is any occurrence of a variable – after its definition – on either the left side of an assignment (*e. g.*, `a = 5`), within an assignment chain (*e. g.*, `a=b=c;`), or in a modification (*e. g.*, `a++`). Any other occurrence which is neither its definition nor a write is a read.

The heuristic was only implemented for the Java programming language and, hence, the evaluation in Section 4 includes only Java systems, too. A similar heuristic can be designed for other object-oriented programming languages.

3.3 Refactoring Patterns

We consider the two common refactorings *pull up method* and *extract method*. To refactor clones, the *pull up method* refactoring can be applied if the clone covers a complete method and if the clone instances do not have a different base class: the cloned method can be moved to the parent class. If the parent class already provides an implementation of this method or other subclasses should not inherit the method, a new base class can be introduced at intermediate hierarchy level that inherits from the former parent class. The *extract method* refactoring can also be applied to clones by moving parts of a cloned method into a new method. To detect extractable parts of a method – called *blocks* – we use the dataflow heuristic. The new method can be in a super class (if one exists) or a utility class.

To refactor long methods, we suggest the *extract method* refactoring. A long method can be shortened by extracting one or several blocks into a new method. A long method can also always be split into two methods with the first method returning the result of the second method. However, this is only an appropriate refactoring if the second method can operate with a feasible number of parameters. We also consider splitting a method as an extract method refactoring.

3.4 Heuristics for Code Clones

The following describes three heuristics to recommend code clones, which are summarized in Table 1.

3.4.1 Common-Base-Class Heuristic

This heuristic suggests to refactor clones by pulling up a cloned method to a common base class. As the superclass already exists, we assume that this can be done easily.

[4]A shallow parser only parses to the level of individual statements, but not all details of expressions. This makes the parser easier to develop and more robust.

Heuristic. This heuristics recommends a clone if all instances have the same base class and if they cover at least one method, detecting this by using a shallow parser.

Sorting and Threshold. To sort the results of this heuristic, we use the length of the methods to be pulled up, measured in number of statements, and sort in descending order. We assume that the more duplicated code can be eliminated, the more useful the recommendation of this finding. The heuristic cuts off the sorting if a clone does not contain a minimal number of l statements in total in the methods that are completely covered by the clone – l is called the *min-length-method* threshold. After preliminary experiments, we set $l = 5$. For example, a clone is included in the sorting, if it covers one method with 5 statements, or five methods with one statement each.

3.4.2 Common-Interface Heuristic

This heuristic recommends the *pull-up* refactoring for clones that implement the same interface. We impose the constraint that no instance of the clone has an existing base class yet (other than Java `Object`) as it occurs frequently that classes with different base classes implement the same interface. In this case, however, pulling up a cloned method can be difficult or maybe not possible. Without an existing base class, we assume that the cloned methods can be easily pulled up into a new base class which should implement the interface instead.

Heuristic. It recommends a clone if the instances all have the same interface and no base class, and if the clone covers at least one complete method, detected by a shallow parser.

Sorting and Threshold. We use the length of the method to be pulled up as descending sorting criteria. The heuristic has the *min-length-method* threshold as the previous heuristic, also with $l = 5$.

3.4.3 Extract-Block Heuristic

This heuristic addresses the *extract method* refactoring. The extracted method ought to be placed in the super or utility class.

Heuristic. It recommends a clone if all its instances contain a block that can be extracted which we detect with our dataflow heuristic: A block is extractable if it writes only at most one local variable that is read after the block before being rewritten. After extracting the block into a new method, this local variable will be the return value of the new method (or `void` if such a variable does not exist). Any local variable of the method that is read within the block to be extracted must become a parameter of the new method.

Parameters. This heuristic operates with one parameter, called *max-parameters*, which limits the maximal number of method parameters for the extractable block. We set it to 3, *i. e.*, a block that would require more than 3 method parameters is not recommended. We pose this constraint to ensure readability of the refactored code: Extracting a method that requires a huge amount of parameters does not make the code more readable than before. The choice of threshold 3 is only based on preliminary experiments. Setting this parameter depends on the quality of the underlying system: For systems with high quality that only reveal few findings, one can still raise this threshold. For systems with low quality, however, we decided to limit this number to recommend only those blocks that are obvious to extract.

Sorting and Threshold. We chose the length of the extractable block as sorting criteria (descending). As for the two other clone heuristics, we assume that the more code can be extracted, the more useful the recommendation. To cut off the sorting, we use the *min-block-length* threshold which requires a minimum number of statements in the extractable block. After preliminary experiments, we set this to 10 as we did not perceive shorter clones to be worth to be extracted as a utility method.

3.5 Heuristics for Long Methods

We propose four heuristics for long methods with a sorting mechanism each to prioritize the results (see Table 1). All heuristics target at the *extract method* refactoring.

Before applying any heuristic, we use two filters to eliminate certain long methods: First, a repetitive code recognizer detects code which has the same stereotypical, repetitive structure, *i. e.*, a sequence of method calls to fill a data map, a sequence of setting attributes, or a sequence of method calls to register JSPs. Long methods containing repetitive code will not be recommended by any heuristic as we frequently observe in industry systems that these methods are easy to understand due to their repetitive nature despite being long. Second, all four heuristics can be parameterized to exclude static initializers from their recommendations. Depending on the context, many industry systems use static initializers to fill static data structures. It remains debatable whether moving the data to an external file, *e. g.*, a plain text file instead of keeping it within the source code improves readability. However, this debate is out of the scope of this paper and the answer might be specific to the underlying system.

3.5.1 Inline-Comment Heuristic

Heuristic. It recommends a long method, if the method contains at least a certain amount of inline comments. This heuristic is based on the observation in [22] that short inline comments indicate that the following lines of code should be extracted into a new method. [22] observes this for inline comments with at most two words. As we have the additional information that the method containing the inline comments is very long, we take all inline comments into account independent from the number of words contained. However, we excluded all inline comments with a `@TODO` tag or with commented out code [22].

Sorting and Threshold. We sort in descending number of inline comments, assuming that the more comments a method contains the more likely a developer wants to shorten it. To cut off the sorting, the *min-comments* threshold requires the recommended method to contain at least certain number of inline comments – which we set to 3.

3.5.2 Extract-Block Heuristic

Heuristic. It recommends a long method if it contains a block which could be extracted into a new method based on our heuristic dataflow analysis. This heuristic operates in the same way and with the same parameters as the *Extract-Block* heuristic for clones (Section 3.4.3).

Parameters. We set *max-parameters* to 3.

Sorting and Threshold. For a descending sorting, we use the length of the extractable block, measured in number of statements. We assume the more code can be extracted to shorten a method, the more useful the recommendation. We set the *min-block-length* threshold to 5.

3.5.3 Extract-Commented-Block Heuristic

Heuristic. This heuristic recommends a subset of the methods recommended by the *extract-block* heuristic by posing additional constraints: It only recommends methods if the extractable block is commented and is either a `for` or a `while` loop, or an `if` statement. This heuristic is based on [22]: short inline comments often indicate that the following lines of code should be extracted to a new method. We assume that the comment preceding a loop or an `if`-statement strongly indicates that the block performs a single action and, hence, should be extracted into its own method.

Parameters. The *max-parameters* parameter is set to 3.

Sorting and Threshold. As before, we use the length of the extractable block as descending sorting criterion. As we assume that the comment already indicates that the block performs a single action on its own, we lower the *min-block-length* threshold and set it to 4.

3.5.4 Method-Split Heuristic

Heuristic. This heuristic recommends a long method if it can be split into two methods with at most a certain number of parameters. It iterates over every top-level block of the method and calculates based on the dataflow how many parameters the new method would require if the original method was split after the block.

Sorting and Threshold. We sort in ascending order of number of parameter for the extracted method. We assume that the less method parameters required, the nicer the refactoring. The sorting operates with a range for the *max-parameter* threshold, indicating how many method parameters the new method requires. After preliminary experiments, we recommend a method for splitting only if the second method works with at least one and at most three parameters. We choose the minimum of one parameter to eliminate methods that only fill global variables such as data maps or that set up UI components. We choose a maximum of three to guarantee easy readability of the refactored code.

4. EVALUATION DESIGN

We evaluate the usefulness of our approach and its limitation. The approach aims at giving the developers a concrete starting point for finding removals. Hence, the top recommended findings by each heuristic should be accepted by developers for removal. Consequently, the approach should have a high precision. In contrast, we do not evaluate the recall of the approach because we are only interested in a useful but not complete list of recommendations. Further, we evaluate what additional context information is necessary to prioritize findings based on developers' opinions.

4.1 Research Questions

RQ1: How many code clones and long methods exist in brown-field software systems? We conduct a benchmark among industry and open-source system to show that clone detection and structural metrics reveal hundreds of findings. We show that the amount of findings is large enough that prioritizing quality defects becomes necessary.

RQ2: Would developers remove findings recommended by our approach? We conduct a developer survey to find out if developers accept the suggested findings for removal. We evaluate which findings developers consider easy to remove and whether they would actually remove them.

Table 2: Benchmark and Survey Objects

Name	Domain	Development	Size [LoC]
ArgoUML	UML modeling tool	Open Source	389952
ConQAT	Code Quality Analysis Toolkit	Open Source	207414
JEdit	Texteditor	Open Source	160010
Subclipse	SVN Integration in Eclipse	Open Source	127657
Tomcat	Java Servlet and JavaServer Pages technologies	Open Source	417319
JabRef	Reference Manager	Open Source	111927
A	Business Information	Industry	256904
B	Business Information	Industry	227739
C	Business Information	Industry	290596
D	Business Information	Industry	105270
E	Business Information	Industry	222914
F	Business Information	Industry	150348
G	Business Information	Industry	55095

RQ3: What additional context information do developers consider when deciding to remove a finding? In the survey, we also ask which context information developers consider when deciding to remove a finding. We evaluate the limits of automatic finding recommendation and quantify how much external context information is relevant. We examine if the type of the context information is different for a local analysis such as the long method detection compared to the global clone detection analysis.

4.2 Study Objects

Table 2 shows the study objects used for the benchmark and the survey, including six open source and seven industry systems, all written in Java. Due to non-disclosure agreements, we use anonymous names A-G for the industry systems. The systems' size range from about 55kLoC to almost 420kLoC. As the open source and industry system span a variety of different domains, we believe that they are a representative sample of software development.

For the benchmark (RQ1), we used all 6 open source and 7 industry systems. For the survey (RQ2, RQ3), we interviewed developers from system C and F. One system stems from Stadtwerke München (SWM), the other from LV 1871: SWM is a utility supplying the city of Munich with electricity, gas, and water. The LV 1871 is an insurance company for life policies. Both companies have their own IT-department – the LV with about 50, the SWM with about 100 developers. We were not able to expand the survey, because other developers were not available for interviews.

4.3 Benchmark Set Up

The benchmark compares all systems with respect to code cloning and method structuring to show that measuring only these two aspects already results in a huge amount of findings in practice. We calculate the clone coverage[5] and the number of (type-II) clone classes per system. We use clone coverage for benchmarking as this is a common metric to measure the amount of duplicated code. Further, we show the number of clone classes to illustrate the amount of (clone) findings a

[5]coverage is the fraction of statements of a system which are contained in at least one clone [13].

Table 3: Experience of Developers

System	∅ Program. Experience	∅ Project Experience	Evaluated Findings
C	19.5 years	4.6 years	65
F	10.8 years	4.5 years	72

developer is confronted with. In terms of method structuring, we show how many long methods exist – further findings among which a developer needs to prioritize. For a better understanding on their impact on maintainability, we also denote how much code relative to the system's size is located in a short (green, < 40 LoC), long (yellow, ≥ 40 LoC), or very long (red, ≥ 100 LoC) method. This distribution reveals the probability that a source code lines is within a long method and, hence, the probability that changing a code line invokes understanding a lone method.

4.4 Survey Set Up

We conducted personal interviews with 4 developers from SWM and LV 1871 each. Based on their general and project specific programming experience (Table 3), we consider them experienced enough for a suitable evaluation. In the interview, we evaluated findings in random order that were recommended by a heuristic and findings not recommended by any heuristic or cut off due to the sorting threshold – which we refer to as *anti-recommendations* or *anti*-group.

Sorting and Sampling. For each heuristic and the anti-groups, we sampled 8 findings unless the heuristic recommended fewer findings[6]. Table 3 shows the overall number of evaluated findings. Within the anti-groups, we sampled randomly. Within the findings recommended by one heuristic, we sorted as depicted in Table 1 and chose the 8 top findings of the sorting. Hence, we evaluate the top recommendations of each heuristic against a random sample from findings that were not recommended by any heuristic. This constitutes an evaluation of the complete approach rather than an evaluation of every single heuristic itself.

Interrater Agreement. We showed most of the samples to only one developer except of two findings per sample group which were evaluated by two developers each. With two opinions on the same finding, we coarsely approximate the interrater agreement. Conducting the survey was a trade-off between evaluating as many findings as possible and getting a statistical significant interrater-agreement due to time limitations of the developers. We decided to evaluate more samples at the expense of a less significant interrater agreement to get more information about when developers do or do not remove a finding. Using our heuristics to provide recommendations to a development team, it is sufficient if one developer decides to remove a recommended finding as long as the heuristics recommend only a feasible set of findings.

Survey Questions. For each finding, we asked the developer two questions.

SQ1: How easy is the following finding to refactor? We provided the answer possibilities *easy*, *medium*, and *hard* and instructed to answer based on the estimated refactoring time and the amount of required context information to refactor correctly: *easy* for 10 minutes, *medium* for about 30 minutes and some required context information, and *hard* for an hour or more and deep source code knowledge. To estimate the refactoring time, we told the developers not to consider additional overhead such as adapting test cases or documenting the task in the issue tracker.

SQ2: Assuming you have one hour the next day to refactor long methods (code clones), would you remove the following long method (clone)? If not, please explain, why. We instructed to answer *No*, if the refactoring was too complicated to be performed in an hour or if the developer did not want to remove the finding for any other reason. We took free text notes about the answer in case of a *No*.

Evaluation Criteria. For SQ1, we calculate the number of answers in the categories *easy*, *medium*, and *hard*. For a disagreement between two developers, we take the higher estimated refactoring time to not favor our results. For SQ2, we calculate the *precision* of each heuristic: how many findings would be removed by a developer divided by the overall number of evaluated findings. For anti-sets, we calculate an *anti-precision* – the number of rejected findings divided by the overall number of evaluated anti-recommendations. We aim for both a high precision and a high anti-precision. In case of a disagreement, we assume that one willing developer is sufficient to remove the finding and count the conflict as a *Yes*. This interpretation leads to better precision but also to a worse anti-precision. Contrarily, evaluating a conflict as a *No* would lead to a worse precision, but a better anti-precision. Both interpretations favor one metric while penalizing the other – we avoid a bias by not only showing the resulting precisions, but also the raw answers.

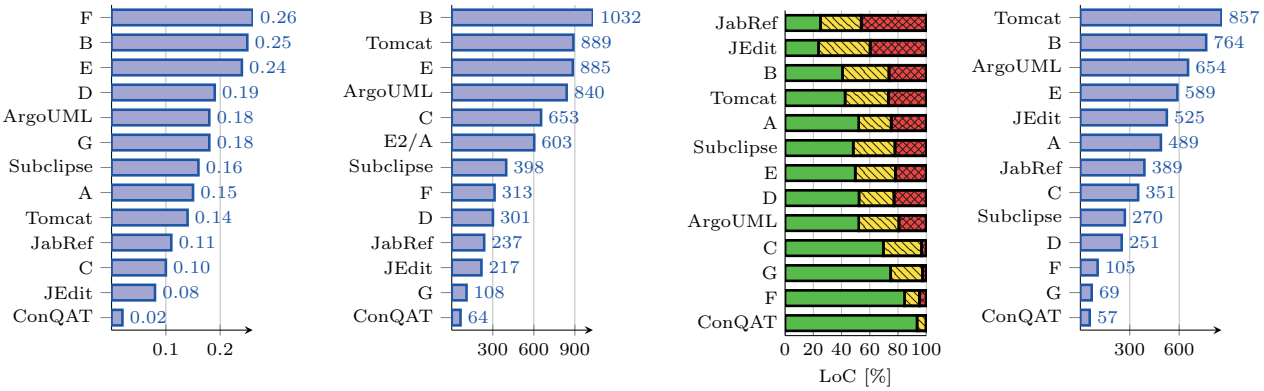
5. RESULTS

5.1 How many code clones and long methods exist in software systems? (RQ1)

Figure 1 shows the benchmark. The clone coverage ranges from 2% to 26%, including application and test code, but excluding generated code. The duplicated code leads to 64 findings (clone classes) for ConQAT and up to 1032 findings for System B. In terms of long methods, the percentage of lines of code that lies in short methods (*i. e.*, less that 40 LoC) varies significantly and ranges from 93% in ConQAT down to 24% for jEdit. Figure 1 shows the percentage of short methods with green, solid filled color, long methods with yellow, hatched color, and very long with red, double hatched color. There are a total number of 57 long methods in ConQAT up to 857 long methods in Tomcat. (The ordering of systems based on the total number of methods and the distribution of source code into short, long, and very long methods is not necessarily the same as, *e. g.*, 50% of the code could be located in only three long methods.)

Conclusion. The code quality measured with clone coverage and code distribution over short and long methods varies between the 13 systems, revealing hundreds of findings on all systems. As the number of findings is too large to be inspected manually, the benchmark shows that prioritizing quality defects is necessary even for only two metrics.

[6]In Sys. C, the Common-Interface heuristic only recommended 5 findings, the Extract-Comm.-Block heuristic 4.



(a) Clone Coverage (b) # Clone Classes (c) Code in Long Methods (d) # Long Methods

Figure 1: Benchmark

5.2 Would developers remove the recommended findings? (RQ2)

Table 4 indicates the developers' estimation how difficult it is to remove a finding (SQ1) based on the number of given answers *easy*, *medium*, and *hard*. For System F, the developers mostly estimated the difficulty to be *medium* or *hard* for the anti-sets. For the heuristics, they predominantly answered *easy*, confirming that our heuristic reveal findings that are easy to remove. For System C, the anti-sets do not reveal the highest costs for removing. Due to filtering repetitive code and static data map fillers, many of the methods in the anti-set are technically easy to refactor. However, developers would not have refactored these methods due to a lack of expected maintenance gain which is reflected in the high anti-precision shown later. Hence, the lower expected costs are no threat to our prioritizing approach. Among all long method heuristics in System C and F, the inline-comment heuristic recommends findings that are the hardest to remove (highest number of *medium* and *hard* answers) as it is the only heuristic that does not use the data-flow analysis.

For each heuristic and anti-set, Table 5 shows the unique *Yes* and *No* answers to SQ2 as well as the number conflicts. Those three numbers sum up to 8 – the number of samples per heuristic – unless less than 8 findings were recommended in total. Summing up all *Yes*, *No*, and conflicting answers (counting as *Yes*) on findings recommended by a heuristic, 84 of 105 (80%) recommended findings were accepted for removal. We conclude that overall, prioritizing by low refactoring costs matches greatly the developers' opinions. The table further indicates precision and anti-precision: The precision for all heuristics are with above 75% very high except of the extract-block heuristic on clones for System C (50%) and the extract-block heuristic on long methods for System F (63%). The anti-precisions are between 63% and 100%. This confirms that developers are willing to remove the findings recommended by a heuristic and mostly confirmed to not remove findings from our anti group. The anti-precisions of 63% on long methods result from many conflicts in the anti-sets: As we count a conflict as a *Yes*, the anti-precision drops o 63% although most answers were unique *No*s.

Table 4: Estimated Refactoring Costs (SQ1)

	C			F		
	Easy	Med.	Hard	Easy	Med.	Hard
Com.-Base-Class	5		3	7		1
Com.-Interface	3	2		6	1	1
Extract-Block	2	4	2	5	1	2
Anti	5	1	2	2	1	5
Inline-Comment	2	5	1	4	2	2
Extract-Block	5		3	8		
Extract-Commented-Block	2	2		6		2
Method-Split	6		2	5		3
Anti	5		3	4	1	3

Interrater Agreement. Developers gave more conflicting answers for long methods (7 conflicts) than for clones (2 conflicts). The removal decision seemed to be more obvious for a global than for a local finding. In System F, most conflicts were on long methods containing anonymous inner classes. Some developers wanted to perform the easy refactoring (which was not considered by our heuristics), others argued to prioritize other long methods as they considered the anonymous classes to be less of a threat for maintainability. Other reasons for conflicts included different opinions about how easy a long method is to understand and how critical the code is in which a long method or clone is located.

Conclusion. The survey showed that the heuristics recommend findings that developers consider to be easy to remove with a very high precision. Developers stated that 80% of the recommended findings are useful to remove. Also, they would not have prioritized most findings from the anti-set. Hence, our approach provides a very good recommendation.

Discussion. One could argue that with our prioritization developers do not carry out more complex and perhaps more critical defect resolutions. However, even if the obviously critical findings are treated separately, the remaining number of findings is still large enough to require prioritization: if the maintenance gain is expected to be equal, then prioritization based on low removal costs is a useful strategy.

Table 5: Decisions to Remove Findings (SQ2)

		C		F	
		Yes / Conflict / No	Precision	Yes / Conflict / No	Precision
Clones	Common-Base-Class	6 / 2	0.75	8	1.0
	Common-Interface	4 / 1	0.8	6 / 2	0.75
	Extract-Block	4 / 4	0.5	5 / 1 / 2	0.75
	Anti	1 / 1 / 6	0.75 (Anti)	8	1.0 (Anti)
Long Method	Inline-Comment	7 / 1	0.89	7 / 1	0.88
	Extract-Block	8	1.0	5 / 3	0.63
	Extract-Comm.-Block	2 / 1 / 1	0.75	6 / 2	0.75
	Method-Split	7 / 1	0.89	5 / 2 / 1	0.88
	Anti	2 / 1 / 5	0.63 (Anti)	3 / 5	0.63 (Anti)

5.3 What additional context information do developers consider? (RQ3)

As a second goal of this work, we quantify which additional context information developers consider. Table 6 shows the reasons why developers did not prioritize a recommended findings. For System F, developers rejected three clones because they are located in two different components which were cloned from each other. The developers rather wanted to wait for a general management decision whether to remove the complete redundancy between the two components. They rejected other clones because they were located in different eclipse projects or because they were located in a manually written instead of generated data object (DTO). Developers rejected most long methods because they did not consider the code to be critical to understand (code without business logic, UI code, logging, or filling data objects) and, hence, were not expected to threaten maintainability. Other reasons include methods that were maybe unused and methods that were not expected to be changed soon.

Most rejected clones from System C were due to redundant xml schemas. As the creation of the xml sheets is not in control of the developers, they cannot remove the redundancy. They further rejected clones that contained only minor configuration code and were, thus, not considered to be error-prone.

Table 6: Reasons for Rejections

System		Reason	Count
F	Clones	Major Cloned Component	3
		Different Eclipse Projects	1
		Manually Generated DTO	1
F	Long Methods	Only UI-Code (not critical, not to test with unit tests)	3
		No Business Logic	2
		Only Logging	1
		Only filling data objects	1
		Maybe unused code	1
		No expected changes	1
C	Clones	Redundancy due to xml schema	4
		Only configuration code	2
		Logging, no critical code	1
C	L.M.	No complexity	2
		No Business Logic (formatting strings, filling data object)	2

Developers of system C rejected long methods which had no complexity or critical business logic.

Conclusion. The survey shows that the considered context information depends on the type of finding or, more general, on the nature of the analysis: For the global analysis of clones, the developers considered a lot more external context information unrelated to the source code. In contrast, for the local analysis of long methods, they rejected the findings mostly due to the nature of the code itself: Developers did not remove clones primarily when they either waited for a general design decision or when the root cause for the redundancy was out of their scope. They rejected long methods mostly when they did not consider the method hard to understand (*e. g.*, UI-code, logging, filling data object).

6. CONCLUSION AND FUTURE WORK

We proposed a heuristic approach to recommend code clones and long methods that are easy to refactor. We evaluated in a survey with two industry systems if developers are willing to remove the recommended findings. The survey showed that 80% of all evaluated, recommended findings would have been removed by developers. When developers would not have removed a finding, we analyzed what context information they used for their decision. For the global analysis of code clones, developers considered a lot more external context information than for the local analysis of long methods: Developers did not want to remove code clones mostly due to external reasons unrelated to the code itself as they were waiting for a general design decision or because the redundancy was caused by external factors. Developers rejected to shorten long methods mostly due to the nature of the code when the methods were not hard to understand, *e. g.*, . if they contained UI-code, logging code, or only filled data objects. In the future, we want to conduct a larger case study to how if our results from the clone and long method analysis can be transferred to other global and local analyses, too, and, hence, can be generalized.

Acknowledgement

We deeply appreciate the cooperation with our two industrial partners SWM and LV 1871, that provided us their source code and time. We also thank Fabian Streitel for his implementation of Java data flow analysis heuristic.

7. REFERENCES

[1] V. Bauer, L. Heinemann, B. Hummel, E. Juergens, and M. Conradt. A framework for incremental quality analysis of large software systems. In *ICSM'12*, 2012.

[2] C. Boogerd and L. Moonen. Prioritizing software inspection results using static profiling. In *SCAM '06*, 2006.

[3] C. Boogerd and L. Moonen. Ranking software inspection results using execution likelihood. In *Proceedings Philips Software Conference*, 2006.

[4] C. Boogerd and L. Moonen. *Using software history to guide deployment of coding standards*, chapter 4, pages 39–52. Embedded Systems Institute, Eindhoven, the Netherlands, 2009.

[5] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *GECCO '06*, 2006.

[6] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Softw.*, 2008.

[7] S. Ducasse, M. Rieger, G. Golomingi, and B. B. Tool support for refactoring duplicated oo code. In *ECOOP'99*, 1999.

[8] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Softw.*, 2001.

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[10] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In *Kansai Science City*, pages 220–233. Springer, 2004.

[11] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, and K. Words. Aries: Refactoring support environment based on code clone analysis. In *SEA 2004*, 2004.

[12] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On software maintenance process improvement based on code clone analysis. In *PROFES'02*, 2002.

[13] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: Incremental, distributed, scalable. In *ICSM'10*, 2010.

[14] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE '09*, 2009.

[15] S. K. and M. Ernst. Prioritizing warning categories by analyzing software history. In *MSR '07*, 2007.

[16] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE '06*, 2006.

[17] M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96*, 1996.

[18] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2008.

[19] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA '96*, 1996.

[20] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *ICSE '08*, 2008.

[21] D. Steidl and N. Goede. Feature-based detection of bugs in clones. In *IWSC '13*, 2013.

[22] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *ICPC'13*, 2013.

[23] A. van Deursen and L. Moonen. The video store revisited – thoughts on refactoring and testing. In *XP'12*, 2002.

[24] R. Venkatasubramanyam, S. Gupta, and H. Singh. Prioritizing code clone detection results for clone management. In *IWSC '13*, 2013.

7.5 Prioritizing Findings in a Quality Control Process

The paper "Continuous Software Quality Control in Practice" (Paper **E**, [85]) embeds the *iterative finding prioritization* into a quality control process. The evaluation shows that the process creates a set-up in which the combined effort of management, developers, and quality engineers actually results in a long-term decreasing number of findings.

Goal The paper provides an enhanced quality control process to show how code quality can effectively be improved in practice.

Approach The quality control process aims to integrate developers, quality engineers, and managers. After defining a quality goal and quality criteria, the process is applied iteratively: In each iteration, the quality engineer manually inspects the analysis results and selects a subset of the revealed findings for removal. He prioritizes findings in new code and lately modified code. For each selected finding, he writes a task for the developers. In regular time intervals, he forwards a report to management to document the effect of the provided quality improvement resources. In the paper, the description of the approach is enhanced with details from a running case study of 41 systems of the Munich RE company in the .NET and SAP area.

Evaluation We evaluated the approach with a qualitative impact analysis of four sample systems at the reinsurance company Munich RE. With the tool Teamscale (Paper **F**), we were able to analyze the complete history of these four systems and evaluate the number of findings before and after quality control was introduced: First, we chose the number of clones, long files, long methods, and high nesting findings as one trend indicator as these quality findings require manual effort to be removed—in contrast to, e. g., formatting violations that can be resolved automatically. We refer to this trend as the overall findings trend. Second, we provide the clone coverage trend and compare both trends with the system size evolution, measured in SLOC.

Results The study reveals a positive trend only after the introduction of the quality control: the overall findings trend decreases for all systems after the process started as well as the clone coverage, even though the systems still grow in size. Those trends go beyond anecdotal evidence but are not sufficient to scientifically proof our method. However, Munich RE decided only recently to extend our quality control from the .NET area to all SAP development. As Munich RE develops mainly in these two areas, most application development is now supported by quality control. The decision to extend the scope of quality control confirms that Munich Re is convinced by the benefit of quality control.

Thesis Contribution This paper shows that the change-based prioritization of internal findings in new and lately modified code (as described in Section 6.2.3) can be effectively applied in practice. It further demonstrates how the joint effort of developers, quality engineers and managers in a quality control process can allocate enough resources to remove findings continuously (see Section 6.5) and improve quality in the long-term—despite ongoing cost and time pressure.

2014 IEEE International Conference on Software Maintenance and Evolution

Continuous Software Quality Control in Practice

Daniela Steidl*, Florian Deissenboeck*, Martin Poehlmann*, Robert Heinke†, Bärbel Uhink-Mergenthaler†
* CQSE GmbH, Garching b. München, Germany
† Munich RE, München, Germany

***Abstract*—Many companies struggle with unexpectedly high maintenance costs for their software development which are often caused by insufficient code quality. Although companies often use static analyses tools, they do not derive consequences from the metric results and, hence, the code quality does not actually improve. We provide an experience report of the quality consulting company CQSE, and show how code quality can be improved in practice: we revise our former expectations on quality control from [1] and propose an enhanced continuous quality control process which requires the combination of metrics, manual action, and a close cooperation between quality engineers, developers, and managers. We show the applicability of our approach with a case study on 41 systems of Munich RE and demonstrate its impact.**

I. Introduction

Software systems evolve over time and are often maintained for decades. Without effective counter measures, the quality of software systems gradually decays [2], [3] and maintenance costs increase. To avoid quality decay, *continuous quality control* is necessary during development and later maintenance [1]: for us, quality control comprises all activities to monitor the system's current quality status and to ensure that the quality meets the quality goal (defined by the principal who outsourced the software development or the development team itself).

Research has proposed various metrics to assess software quality, including structural metrics[1] or code duplication, and has led to a massive development of analysis tools [4]. Much of current research focuses on better metrics and better tools [1], and mature tools such as ConQAT [5], Teamscale [6], or Sonar[2] have been available for several years.

In [1], we briefly illustrated how tools should be combined with manual reviews to improve software quality continuously, see Figure 1: We perceived quality control as a simple, continuous feedback loop in which metric results and manual reviews are used to assess software quality. A quality engineer – a representative of the quality control group – provides feedback to the developers based on the differences between the current and the desired quality. However, we underestimated the amount of required manual action to create an impact. Within five years of experience as software quality consultants in different domains (insurance companies, automotive manufacturers, or engineering companies), we frequently experienced that tool support alone is not sufficient for successful quality control in practice. We have seen that most companies cannot create an impact on their code quality although they employ tools for quality measurements because the pressure to implement new features does not allow time for quality assurance: often, newly introduced tools get attention only for a short period of time, and are then forgotten. Based on our experience, quality control requires actions beyond tool support.

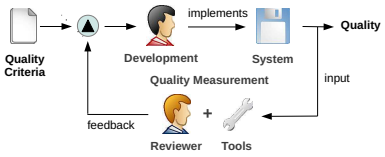


Fig. 1. The former understanding of a quality control process

In this paper, we revise our view on quality control from [1] and propose an enhanced quality control process. The enhanced process combines automatic static analyses with a significantly larger amount of manual action than previously assumed to be necessary: Metrics constitute the basis but quality engineers must manually interpret metric results within their context and turn them into actionable refactoring tasks for the developers. We demonstrate the success and practicability of our process with a running case study with Munich RE which contains 32 .NET and 9 SAP systems.

II. Terms And Definitions

- A *quality criterion* comprises a metric and a threshold to evaluate the metric. A criterion can be, *e. g.*, to have a clone coverage below 10% or to have at most 30% code in long methods (*e. g.*, methods with more than 40 LoC).
- *(Quality) Findings* result from a violation of a metric threshold (*e. g.*, a long method) or from the result of a static code analysis (*e. g.*, a code clone).
- *Quality goals* describe the abstract goal of the process and provide a strategy how to deal with new and existing findings during further development: The highest goal is to have no findings at all, *i. e.*, all findings must be removed immediately. Another goal is to avoid new findings, *i. e.*, existing findings are tolerated but new findings must not be introduced. (III-B will provide more information).

III. The Enhanced Quality Control Process

Our quality control process is designed to be *transparent* (all stakeholders involved agree on the goal and consequences

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant EvoCon, 01IS12034A. The responsibility for this article lies with the authors.

[1] *e. g.*, file size, method length, or nesting depth

[2] http://www.sonarqube.org/

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.95

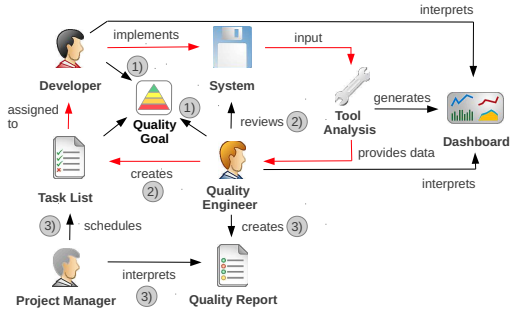


Fig. 2. The enhanced quality control process

of failures), *actionable* (measuring must cause actions) and *binding* (quality engineers, developers, and managers follow the rules they agreed on.). These following three main activities reflect these conceptual ideas and are depicted in Figure 2.

1) The quality engineer defines a quality goal and specific criteria for the underlying project in coordination with management and in agreement with the developers. Criteria are usually company-specific, the goal needs to be adapted for each system. A common definition of the quality goal and criteria makes the process transparent.
2) The quality engineer takes over responsibility to manually interpret the results of automatic analyses. He reviews the code in order to specify actionable and comprehensible refactoring tasks at implementation level for the developers which will help to reach the predefined quality goal. This makes the process actionable as metric results are turned into refactoring tasks.
3) In certain intervals, the quality engineer documents which quality criteria are fulfilled or violated in a quality report and communicates it to the management. This creates more quality awareness at management level. The project manager reviews and schedules the tasks of the quality engineer, making the process binding for management (to provide the resources) and for the developers (to implement the suggested refactorings).

In the following, we will discuss the process in detail. To demonstrate the practicability of our approach, we combine the description of the process with details from a running case study of more than four years at Munich RE. When we enhance the process description with specific information from Munich Re, we use a "@MunichRe" annotation as follows:

@Munich RE. The case study demonstrates how we apply the quality control process to 32 .NET and 9 SAP software systems mainly developed by professional software services providers on and off the premises of Munich RE. These applications comprise ∼11,000 kLoC (8,800 kLoC maintained manually, 2,200 kLoC generated) and are developed by roughly 150 to 200 developers distributed over multiple countries.

A. Quality Engineer

The quality engineer (QE) is in charge of interpreting the metric results, reviewing the system, writing the report and communicating it to developers and managers. The QE must be a skilled and experienced developer: he must have sufficient knowledge about the system and its architecture to specify useful refactoring tasks. This role can be carried out by either a member of the development team or by an external company. Based on our experience, most teams benefit from a team-external person providing neutral feedback and not being occupied by daily development. We believe that the process's success is independent from an internal or external QE, but dependent on the skills of the quality engineer.

@Munich RE. In all 41 projects of our case study, the CQSE fulfills the tasks of a quality engineer who operates on site. Due to regular interaction with the quality engineer, developers perceive him as part of the development process.

B. Quality Goals

The right quality goal for a project constitutes the backbone for a successful control process. Based on our experience, the following four goals cover the range of different project needs.

QG1 (*indifferent*) Any quality sufficient – No quality monitoring is conducted at all.

QG2 (*preserving*) No new quality findings – Existing quality findings are tolerated but new quality findings must not be created (or consolidated immediately).

QG3 (*improving*) No quality findings in modified code – Any code (method) being modified must be without findings, leading to continuous quality improvement during further development, similar as the boy scouts rule also used for extreme programming ("Always leave the campground cleaner than you found it" [7]).

QG4 (*perfect*) No quality findings at all – Quality findings must not exist in the entire project.

@Munich RE. Among the 41 projects, 2 projects are under QG1, 18 under QG2, 10 under QG3 and 11 under QG4.

C. Quality Criteria

The quality criteria depend on project type and technologies used. The quality engineer is responsible for a clear communication of the criteria to all involved parties, including third-party contractors for out-sourced software development.

@Munich RE. For .NET projects, we use 10 quality criteria based on company internal guidelines and best practices: code redundancy, structural criteria (nesting depth, method length, file size), compiler warnings, test case results, architecture violations, build stability, test coverage, and coding guidelines violations. For SAP systems, we use seven quality criteria with similar metrics but different thresholds: clone redundancy, structural criteria, architecture violations, critical warnings and guideline violations.

D. Dashboard

As part of quality control, a dashboard provides a customized view on the metric results for both developers and quality engineers and makes the quality evaluation transparent (see

Figure 2). The dashboard should support the developer to fulfill the quality goal of his projects and show him only findings relevant to the quality goal: For QG2 and QG3, we define a reference system state (baseline) and only show new findings (QG2) or new findings and findings in modified code (QG3) since then. Measuring quality relative to the baseline motivates the developers as the baseline is an accepted state of the system and only the relative system's quality is evaluated.

@Munich RE. We use ConQAT as the analysis tool and dashboard, which integrates external tools like FxCop, StyleCop, or the SAP code inspector.

E. Quality Report

In regular time intervals, the quality engineer creates a quality report for each project. The report gives an overview of the current quality status and outlines the quality trend since the last report: The report contains the interpretation of the current analysis results as well as manual code reviews. To evaluate the trend of the system's quality, the quality engineer compares the revised system to the baseline with respect to the quality goal. He discusses the results with the development team. The frequency of the quality report triggers the feedback loop and is in accordance to the release schedule. The quality engineer forwards the report to the project management to promote awareness for software quality. This guarantees that developers do not perceive quality control as an external short-term appearance, but as an internal management goal of the company.

@Munich RE. Quality reports are created for the majority of applications every three or four months. Exceptions are highly dynamic applications with five reports a year and applications with little development activities with only one report a year.

F. Task List

Based on the differences in the current findings and the quality goal, the quality engineer manually derives actionable refactoring tasks[3]. He forwards these tasks to the developers and the project manager who can decide to accept or reject a task. For accepted refactorings, the project manager defines a due date. For rejected refactorings, he discusses the reasons for rejection with the quality engineer. Before the next report is due, the quality engineer checks that developers completed the scheduled tasks appropriately. The task list constitutes the focal point in the combination of tools, quality engineers, and management. The success of the process depends on the ability of the quality engineer to create tasks that effectively increase the system's quality.

@Munich RE. The quality engineer manually inspects all findings relevant to the project's quality goal. He creates tasks for findings he considers to hamper maintainability. It remains up to him to black-list findings that are not critical for code understanding or maintenance.

[3]*e. g., Remove the redundancy between class A and B by creating a super class C and pull up methods x,y,z)*

TABLE I
SAMPLE OBJECTS FOR IMPACT ANALYSIS

Name	QG	Quality Control	Size	# Developers
A	3	4 years	200 kLoC	4
B	3	5 years	276 kLoC	5
C	3	3 years	341 kLoC	4
D	4	1.5 years	15 kLoC	2

IV. IMPACT @MUNICH RE

Proving the process' success scientifically is difficult for several reasons: First, we do not have the complete history of all systems available. Second, we cannot make any reliable prediction about the quality evolution of these systems in case our process would not have been introduced. Consequently, we provide a qualitative impact analysis on four sample systems with an evolution history of up to five years rather than a quantitative impact analysis of all 41 systems. For these samples, we are able to show the quality evolution before and after the introduction of quality control.

First, we chose the number clones, long files, long methods, and high nesting findings as one trend indicator as these quality findings require manual effort to be removed – in contrast to, *e. g.*, formatting violations that can be resolved automatically. We refer to this trend as the overall findings trend. Second, we provide the clone coverage trend and compare both trends with the system size evolution, measured in SLoC. All trends were calculated with Teamscale [6].[4]

We show these trends exemplary for the three systems with the longest available history (to be able to show a long-term impact) and with a sufficiently large development team size (to make the impact of the process independent from the behavior of a single developer). As these three systems are all QG3, we choose, in addition, one QG4 system (Table I). Figures 3–6 show the evolution of the system size in black, the findings trend in red (or gray in black-white-print), and the clone coverage in orange (or light gray in black-white-print). The quality controlled period is indicated with a vertical line for each report, *i. e.*, quality control starts with the first report date.

Figure 3 shows that our quality control has a great impact on System A: Prior to quality control, the system size grows as well as the number of findings. During quality control, the system keeps growing but the number of findings declines and the clone coverage reaches a global minimum of 5%. This shows that the quality of the system can be measurably improved even if the system keeps growing.

For System B, quality control begins in 2010. However, this system has already been in a scientific cooperation with the Technische Universitaet Muenchen since 2006, in which a dashboard for monitoring the clone coverage had been introduced. Consequently, the clone coverage decreases

[4]In contrast to ConQAT, Teamscale can incrementally analyze the history of a system including all metric trends within feasible time. Hence, although ConQAT is used as dashboard within Munich Re, we used Teamscale to calculate the metric trends.

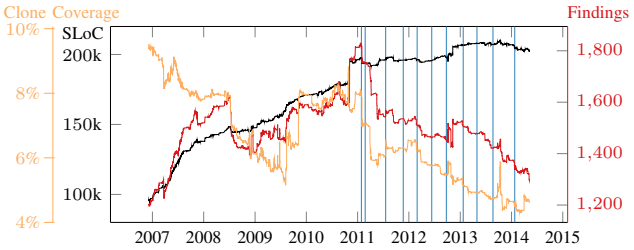


Fig. 3. System A

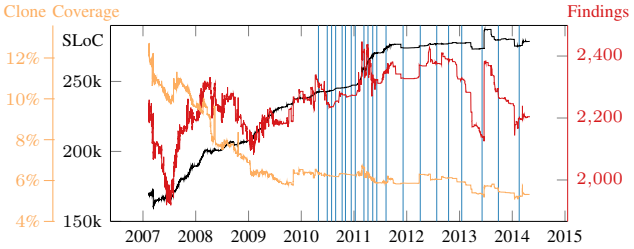


Fig. 4. System B

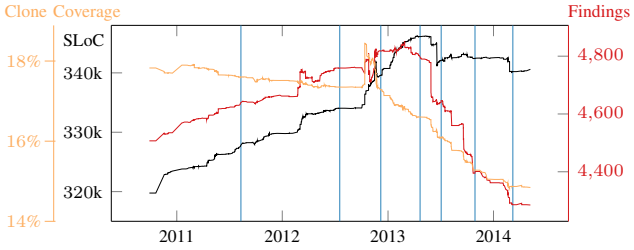


Fig. 5. System C

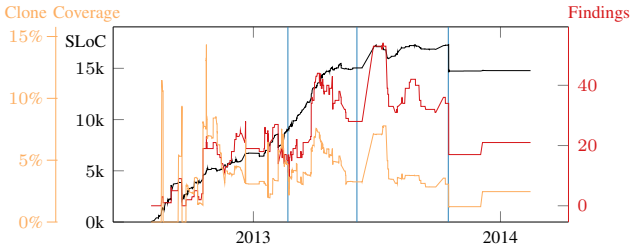


Fig. 6. System D

continuously in the available history (Figure 4). The number of findings, however, increases until mid 2012. In 2012, the project switched from QG2 to QG3. After this change, the number of findings decreases and the clone coverage settles around 6%, which is a success of the quality control. The major increase in the number of findings in 2013 is only due to an automated code refactoring introducing braces that led to threshold violations of few hundred methods. After this increase, the number of findings start decreasing again, showing the manual effort of the developers to remove findings.

For System C (Figure 5), the quality control process shows a significant impact after two years: Since the end of 2012, when the project also switched from QG2 to QG3, both the clone coverage and the overall number of findings decline. In the year before, the project transitioned between development teams and, hence, we only wrote two reports (July 2011 and July 2012).

System D (Figure 6) almost fulfills QG4 as after 1 year of development, it has only 21 findings in total and a clone coverage of 2.5%. Technically, under QG4, the system should have zero findings. However, in practice, exactly zero findings is not feasible as there are always some findings (*e. g.*, a long method to create UI objects or clones in test code) that are not a major threat to maintainability. Only a human can judge based on manual inspection of the findings whether a system still fulfills QG4, if it does not have exactly zero findings. In the case of System D, we consider 21 findings to be few and minor enough to fulfill QG4.

To summarize, our trends show that our process leads to actual measurable quality improvement. Those trends go beyond anecdotal evidence but are not sufficient to scientifically proof our method. However, Munich RE decided only recently to extend our quality control from the .NET area to all SAP development. As Munich RE develops mainly in the .NET and SAP area, most application development is now supported by quality control. The decision to extend the scope of quality control confirms that Munich Re is convinced by the benefit of quality control. Since the process has been established, maintainability issues like code cloning are now an integral part of discussions among developers and management.

V. Conclusion

Quality analyses must not be solely based on automated measurements, but need to be combined with a significant amount of human evaluation and interaction. Based on our experience, we proposed a new quality control process for which we provided a running case study of 41 industry projects. With a qualitative impact analysis at Munich RE we showed measurable, long-term quality improvements. Our process has led to measurable quality improvement and an increased maintenance awareness up to management level at Munich Re.

References

[1] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka, "Tool support for continuous quality control," in *IEEE Software*, 2008.
[2] D. L. Parnas, "Software aging," in *ICSE '94*.
[3] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. Software Eng.*, 2001.
[4] P. Johnson, "Requirement and design trade-offs in hackystat: An in-process software engineering mea- surement and analysis system," in *ESEM'07*.
[5] F. Deissenboeck, M. Pizka, and T. Seifert, "Tool support for continuous quality assessmenet," in *STEP'05*.
[6] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software quality control in real-time," in *ICSE'14*.
[7] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008.

7.6 Tool Support for Finding Prioritization

The paper "Teamscale: Software Quality Control in Real-Time" (Paper **F**, [40]) focuses on the *tool support* to make the usage of static analysis applicable in practice.

Goal The paper presents the code analysis tool Teamscale which provides results from static analyses in real-time and integrates well into daily development.

Approach Teamscale is a quality analysis and management tool that overcomes the drawbacks of existing static analysis tools: Whereas existing tools operate in batch-mode, Teamscale provides up-to-date static analysis results within seconds. Based on incremental quality analyses, the tool processes each commit individually and, hence, provides real-time feedback to the developer. The tool has the full history of the code quality's evolution at its disposal, supporting efficient root cause analysis for specific quality problems. Combining incremental analysis with findings tracking, the tool distinguishes between old and new findings very accurately. This is a prerequisite for developers to tame the findings flood and to focus on newly created problems.

Evaluation We evaluated our tool with a development team from the German life insurance company LV 1871 which comprises about ten developers, among which six took part in the evaluation. We first gave the developers a short introduction to the tool. Afterwards, the developers were asked to solve specific tasks with Teamscale and to fill out an anonymous online survey about their user experience.

Results The evaluation showed that the developers were able to complete almost all tasks successfully and indicated a high user acceptance of the tool. The developers stated in the survey that they would like to use Teamscale for their own development: The tool provides fast enough feedback to integrate static analysis well into daily development and enables developers to accurately differentiate between old and new findings.

Thesis Contribution Teamscale provides the underlying tool support for this thesis: On the one hand, its incremental analysis engine allowed us to implement the case studies of Paper **B**, **A**, and **E**. On the other hand, it provides the necessary technical tool foundation to make the change-based prioritization and the quality control process applicable in practice (Section 6.5.3): this paper shows how drawbacks with existing static analysis tools can be overcome and how the tool helps to apply static analysis effectively in practice; its accurate findings tracking makes it possible to differentiate between new and old findings and, hence, helps the developers to manage the findings flood (Section 6.2.3).

Contribution of the Author As second author, we conducted the evaluation of the tool at our industrial partner and created the demonstration video[1] necessary for the submission as a tool paper. We contributed significantly to the writing but only partly to the implementation of Teamscale by providing the origin analysis [89] for the findings tracking.

[1] https://www.cqse.eu/en/products/teamscale/overview/

Teamscale: Software Quality Control in Real-Time *

Lars Heinemann Benjamin Hummel Daniela Steidl
CQSE GmbH, Garching b. München, Germany
{heinemann,hummel,steidl}@cqse.eu

ABSTRACT

When large software systems evolve, the quality of source code is essential for successful maintenance. Controlling code quality continuously requires adequate tool support. Current quality analysis tools operate in batch-mode and run up to several hours for large systems, which hampers the integration of quality control into daily development. In this paper, we present the incremental quality analysis tool Teamscale, providing feedback to developers within seconds after a commit and thus enabling real-time software quality control. We evaluated the tool within a development team of a German insurance company. A video demonstrates our tool: `http://www.youtube.com/watch?v=nnuqplu75Cg`.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*

General Terms

Management, Measurement

Keywords

Quality control, static analysis, incremental, real-time

1. INTRODUCTION

In many domains, software evolves over time and is often maintained for decades. Without effective counter measures, the software quality gradually decays [4,10], increasing maintenance costs. To avoid long-term maintenance costs, continuous quality control is necessary and comprises different activities: When developers commit changes, they should be aware of the impact on the system's quality and when introducing new quality defects, they should remove them appropriately. Project and quality managers should check the quality status in regular intervals and take actions to improve quality. Research has proposed metrics to assess software quality, including structural metrics (*e. g.*, nesting depth) or redundancy measurements (*e. g.*, code cloning), leading to a massive development of analysis tools.

However, existing tools have three major disadvantages. First, they require up to several hours of processing time for large systems [1]. Consequently, quality analyses are scheduled to off-work hours (e.g. nightly build) and developers are not notified of problems in their code until the next morning. However, by then, they often already switched context and work on a different task. Second, development teams produce hundreds of commits per day. Nightly analyses aggregate them and make it hard to identify the root causes for unexpected quality changes. Instead, developers must reverse engineer the root causes from the versioning system. Third, analysis techniques reveal thousands of quality problems for a long-lived software, making it infeasible to fix all quality issues at once. With existing tools, developers cannot differentiate between old and new quality defects, which often results in frustration and no improvement at all.

Problem Statement. *Existing quality analysis tools have long feedback cycles and cannot differentiate between old and new findings, hampering integration into daily development.*

Teamscale[1] overcomes these drawbacks with incremental quality analyses [1]. By processing each commit individually, it provides real-time feedback and reveals the impact of a single commit on the system's quality. With the full history of the code quality at its disposal, Teamscale supports efficient root cause analysis for specific quality problems. Combining incremental analysis with tracking [11] of quality issues, the tool distinguishes between old and new quality defects, encouraging developers to fix recently added findings.

Contribution. *Teamscale provides quality analysis in real-time, integrating into the daily development.*

With a web front-end and an IDE integration, Teamscale offers different perspectives on a system's quality. It provides an overview of all commits and their impacts on quality and a file-based view on source code annotated with quality defects. It also offers an aggregated view on the current quality status with dashboards and enables root cause analysis for specific quality problems.

*This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591068

[1] `http://www.teamscale.com`

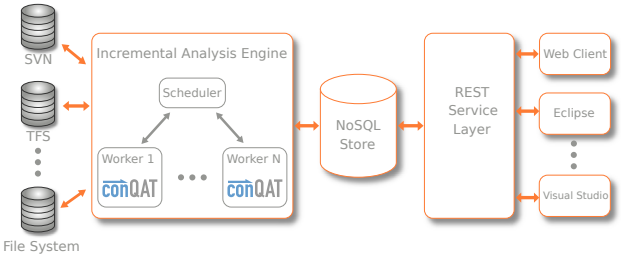


Figure 1: High-Level Architecture

2. RELATED WORK

In the current state of the art, many quality analysis tools already exist.There are numerous tools, for example, for the analysis of a specific quality metric or defect. While some of them are even incremental [5], Teamscale aims at providing a differentiated, all-encompassing perspective of software quality by using a broad spectrum of metrics and analyses.

Distributed Quality Analysis Tools. Shang et al. [12] propose to address scalability by adapting tools which originally were designed for a non-distributed environment and deploying them on a distributed platform such as MapReduce. Using enough computing ressources, their approach can reduce the feedback to developers to minutes. However, their approach focusses on individual analyses and does not provide a comprehensive front-end to the quality analysis data. Additionally, our approach of incremental analysis requires significantly less computing resources.

Quality Dashboards. Multiple tools provide the user with a dashboard containing various aggregations and visualizations of a system's source code quality. Tools include ConQAT [3], SonarQube[2], or the Bauhaus Suite[3]. All of these tools provide a custom analysis engine, integrate the results of other existing analysis tools, and provide configurable means of displaying the results. All of them, however, require multiple hours on large systems, making real-time feedback and fine-grained root cause analysis impossible.

Incremental Quality Analysis Tools. The conceptually most similar tool to ours is SQUANER [7]. It also proceeds incrementally, and updates are triggered by commits. It also aims to provide rapid developer feedback. However, the calculated metrics are file-based and thus limited to local analyses (see [1] for more details). Unfortunately, the corresponding web site is unreachable, but the paper suggests that the data representation does not support root cause analysis and that no tracking of quality defects is provided.

3. ARCHITECTURE

Teamscale is a client/server application: the quality analyses are performed on a central server and different user clients present the results. Figure 1 shows an overview of Teamscale's architecture: The incremental analysis engine in the back-end connects to various different data sources, *e. g.*, version control systems. A REST Service Layer provides the interface of the analysis results for different front-end clients.

Data Sources. Teamscale directly connects to different version control system (VCS), such as Subversion, GIT and Microsoft's TFS. It constantly monitors the activity in the VCS and directly fetches its changes. Each commit in the VCS triggers the update of all quality analyses. Consequently, in contrast to batch tools, Teamscale does not need to be integrated with continuous integration tools. Teamscale is able to analyze multi-language projects, supporting all widely used languages, such as Java, C#, C/C++, JavaScript, and ABAP. Teamscale further relates quality analysis results with bug or change request data as it connects to bug trackers such as Jira, Redmine and Bugzilla.

Incremental Analysis Engine. In the back-end, the incremental analysis engine [1] is built on top of the tool ConQAT [3]. Instead of analyzing a complete software system in batch-mode, it updates quality metrics and findings with each commit to the VCS. Our study in [1] showed that our incremental analyses perform significantly faster than batch analyses. As commits typically consist of only few files, Teamscale computes their impact on the system's source code quality within seconds, providing rapid feedback for the developer. Monitoring the quality changes based on single commits enables a fine-grained root cause analysis.

Storage System. Teamscale has a generic data storage interface supporting various noSQL data stores (such as Apache Cassandra[4]). Incremental storage of the analysis results over the system's history [1] allows to store the full analysis data for every single commit as well as the complete history of every source code file within reasonable space. A medium-sized system with 500 kLOC and a history of 5 years typically requires 2–4 GB of disk space.

REST Service Layer. A REST service layer provides an interface for retrieving all stored quality analysis results, supporting a variety of clients on multiple platforms.

Clients. Multiple user clients present the quality analysis' results. A JavaScript-based web front-end allows platform independent access to Teamscale. For developers, IDE clients integrate quality analysis results directly in their development environment, such as Eclipse or Visual Studio.

4. CODE QUALITY ANALYSES

Teamscale offers many common quality analyses. For the remainder of this paper, we refer to the term *finding* for all quality defects that can be associated with a specific code region. A finding can, *e. g.*, result from a violation of a metric threshold (such as a file which is too long) or be revealed by a specific analysis such as clone detection or bug pattern search. In the following, we will give an overview of the provided analyses:

Structure Metrics. As major structural metrics, Teamscale comprises file size, method length, and nesting depth. These metrics are aggregated based on the directory structure and result in findings when thresholds are violated.

Clone Detection. Teamscale uses an incremental, index-based clone detection algorithm [8] to reveal code duplication by copy and paste. It can be also configured to use an incremental gapped clone detection to find inconsistent clones which are likely to contain incomplete bug fixes [9].

Code Anomalies. Teamscale analyzes many different code anomalies, *e. g.*, naming convention violations, coding guideline violations, and bug patterns: we integrate state-of-the-art bug pattern detection tools such as FindBugs[5],

[2]Formerly called *Sonar*, `http://www.sonarqube.org`
[3]`http://www.axivion.com/products.html`
[4]`http://cassandra.apache.org`
[5]`http://findbugs.sourceforge.net`

PMD[6], and StyleCop[7]. Tools that do not work incrementally (*e. g.*, FindBugs due to non-trivial cross-file analysis) are integrated via a finding import from the nightly build.

Architecture Conformance. Architecture conformance assessment compares a high-level architecture specification model against the actual dependencies in the code. Teamscale uses the ConQAT architecture conformance assessment [2] which provides a simple component dependency model. The system is modeled as a hierarchical decomposition into components with dependency policies between them. The assessment result is an annotated version of the component model that rates each dependency as satisfied or violated.

Source Code Comments. The code comment analysis verifies if documentation guidelines for the existence of code comments are met, which may for instance prescribe that every public type and public method must have an interface comment. In addition, a machine-learning based algorithm reveals commenting deficits, such as trivial or inconsistent comments which do not promote system understanding [13].

Test Quality. From the nightly build, Teamscale uses test coverage data and visualizes which parts of the source code are covered by tests.

5. USER CLIENTS

The results of all quality analyses are available in either a web front-end or the IDE.

5.1 Web Front End

The web client is organized in *perspectives*, which provide different views of the system's quality: The *code perspective* allows to browse the source code, its quality metrics and findings. The *activity perspective* displays all commits in the system's history, the *finding perspective* presents all current quality findings whereas the *dashboard perspective* offers an aggregated overview of the quality status. The web front-end supports various different use cases (UC):

UC1: Inspecting quality defects in specific components. In the code perspective, Teamscale offers a file-system based view on source code, metrics and findings. The user can navigate through the directory structure to find quality defects in specific parts of the system. As all source code and quality analysis results are held in Teamscale for the entire history, the code perspective also functions as a *time machine* to browse previous versions of the code and its findings. On file level, the source code is annotated with findings indicated with colored bars and tool tips next to the code. For each directory level and quality metric, Teamscale provides trend charts showing the evolution of the metric for the selected directory, such as the growth in lines of code or the evolution of the clone coverage.

UC2: Inspecting the quality impact of the last commit. The activity perspective shows all commits. Besides the information on the change itself (*i. e.*, revision number, author, message, affected files), Teamscale reveals the impact on the quality status for each change: It indicates how many quality problems were introduced or removed with this change. In addition, the bug tracker integration relates the commits to change requests or bugs.

UC3: Comparison of two versions of the system. Teamscale can compute the *change delta* between two different versions of the system, which shows the changes within the time frame between the two versions: The change delta contains information about added, deleted, or changed files, the version control system commits within the time frame, metric value changes, as well as added or removed findings.

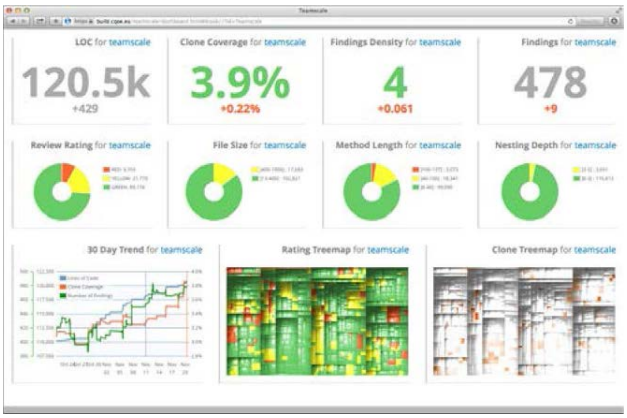


Figure 2: Dashboard Perspective

UC4: Assessment of architecture conformance. Teamscale provides a visualization of the architecture specification annotated with its assessment according to the current dependencies in the code. In case a dependency policy is violated, it allows to identify the problematic dependency in the code, i.e. navigate from the model down to individual files that contain the architecture violation.

UC5: Identification and management of findings. In the findings perspective, the user can browse quality defects along two dimensions—the directory hierarchy and the finding categories. Teamscale is capable of tracking these findings during the software evolution, in which the location of a finding can change: the finding may move within a file or even across files due to edit operations, moving code snippets, or file renames. Teamscale can track this, using a combination of ideas from [6] and [11]. Based on findings tracking, individual findings, such as false positives, can be *blacklisted*. A blacklisted finding will not be further reported. Teamscale also offers the opportunity to only inspect findings that were added since a configurable revision, *e. g.*, the last release. This way, when developers clean up code, they can focus on newly added findings.

UC6: The system's quality at a glance. The dashboard perspective gives an at-a-glance view on the overall quality status. The users can freely build personal dashboards based on a variety of configurable widgets. With his personal dashboard, the user can focus on specific quality aspects that are relevant for him. Teamscale provides widgets for displaying metric values with delta indicators, metric distributions, tree maps, trend charts, and quality deficit hotspots. Figure 2 shows an example dashboard.

5.2 IDE Integration

Teamscale provides plug-ins for two widely used IDEs: Eclipse and Visual Studio. The IDE clients annotate the code with findings from the Teamscale server, making developers aware of existing quality issues in their code.

[6] `http://pmd.sourceforge.net`
[7] `https://stylecop.codeplex.com`

6. QUALITY CONTROL SUPPORT

Teamscale is designed to be used in a continuous quality control process during daily development. Continuous control requires constant monitoring of the quality, a transparent communication about the current status as well as specific tasks on how to improve the quality. Teamscale supports various usage scenarios for different stakeholders in this process: The project-manager uses the dashboard to check the quality status of his system in regular intervals. For each commit, a developer receives feedback about findings that were introduced or removed with this commit. A quality manager can monitor the evolution of the quality, detect the root-cause for quality problems and create specific tasks to improve the system's quality.

7. EVALUATION

We evaluated our tool with a development team from a German insurance company, LV 1871 which comprises about ten developers, among which six took part in the evaluation. The developers had an average of 18.5 years of programming experience and are hence considered suitable for evaluation.

Set Up. We first gave the developers a short introduction to the tool. Afterwards, the developers were asked to solve specific tasks with Teamscale and to fill out an anonymous online survey about their user experience. The tasks comprised nine questions related to the quality of the team's software and were guided by the use cases presented in Section 5.1, except of UC4 (as no architecture specification was available). Developers were also asked to commit live to the tool and evaluate the feedback cycle. Solving the tasks ensured that the developers were familiar with the tool before taking part in the survey and showed whether developers succeeded to use the tool to find the required information.

Results. Of all 9 tasks, all developers were able to complete 7 tasks correctly. Two tasks regarding the dashboard perspective were not solved correctly by all, two developers each made a mistake. Problems with those two tasks were related to a lacking piece of information about a specific dashboard widget. In the survey, developers rated several statements about Teamscale on a Likert scale from 1 (I totally agree) to 6 (I do not agree at all). Table 1 shows the average developer response per statement (μ) and the standard deviation (σ). The results show a very positive user experience. All developers agreed that they want to use Teamscale for their own development ($\mu = 1.2$) and that Teamscale provides fast feedback in practice ($\mu = 2$). They also agreed that they were able to differentiate between new and old quality problems ($\mu = 1.5$) and that Teamscale enables root-cause analysis ($\mu = 1.67$). To summarize, the developers showed that they were able to solve specific tasks successfully using the tool and that they would like to be supported by the tool in their daily development.

Table 1: Average Developers' Agreement

Statement	μ	σ
Teamscale provides a good overview of the quality of my software.	1.5	0.5
Teamscale helps to identify the root cause of quality problems.	1.67	0.47
Teamscale supports me in a quality-driven software development	1.67	0.47
Teamscale provides feedback fast enough in practice.	2	0.57
Teamscale makes it easy to differentiate between new and old quality problems.	1.5	0.5
Teamscale's UI is intuitive.	2	0
I'd like to use Teamscale for my own development.	1.2	0.37

8. CONCLUSION

In this paper, we presented a comprehensive quality analysis tool which performs in real-time. Giving feedback about the impact of a change on the quality of a system within seconds, the tool smoothly integrates into the daily development process, enabling successful continuous quality control in practice. Providing root cause analysis, developers can identify the cause of a specific quality problem and address it appropriately. With the help of findings tracking, developers can decide which findings should be reported, helping them to manage and remove findings such that quality actually improves. With the tool, developers, quality managers, and project managers get their own perspective on the system's quality, adapted to the specific needs of each role. A survey among developers showed a high user acceptance.

Acknowledgements

We thank our evaluation partner LV 1871 for their support.

9. REFERENCES

[1] V. Bauer, L. Heinemann, B. Hummel, E. Juergens, and M. Conradt. A framework for incremental quality analysis of large software systems. In *ICSM'12*, 2012.

[2] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with ConQAT. In *ICSE'10*, 2010.

[3] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Softw.*, 2008.

[4] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Softw.*, 2001.

[5] N. Göde and R. Koschke. Incremental clone detection. In *CSMR'09*, 2009.

[6] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *ICSE '11*, 2011.

[7] N. Haderer, F. Khomh, and G. Antoniol. SQUANER: A framework for monitoring the quality of software systems. In *ICSM'10*, 2010.

[8] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: Incremental, distributed, scalable. In *ICSM'10*, 2010.

[9] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE '09*, 2009.

[10] D. L. Parnas. Software aging. In *ICSE '94*, 1994.

[11] S. P. Reiss. Tracking source locations. In *ICSE '08*, 2008.

[12] W. Shang, B. Adams, and A. E. Hassan. An experience report on scaling tools for mining software repositories using mapreduce. In *ASE'10*, 2010.

[13] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *ICPC'13*, 2013.

"The scientific man does not aim at an immediate result. His duty is to lay the foundation for those who are to come, and point the way."

Nikola Tesla

8 Conclusion

The final chapter summarizes the contributions of our work (Section 8.1), and outlines directions for future research (Section 8.2).

8.1 Summary

Automated static analyses are a powerful quality assurance technique. As they analyze the code only, they are easy to perform; they can be applied without the runtime environment of the software and even before a system is tested. Yet, many companies struggle to apply them successfully in practice. Even though they have static analysis tools installed, the number of findings often keeps increasing and code quality often decays further.

Based on our industry experience of three years as software quality consultants working with customers from various different domains, we identified five major challenges developers usually face when applying static analyses. While these claims were certainly not generalizable to all software engineering, we believed they yet apply to a substantially large subset of long-lived software across different domains. The five challenges comprised the overly large number of findings, the different benefit of their removal, the varying relevance of findings as well as controlling their evolution and obtaining management priority.

In this thesis, we presented an approach to apply static analyses effectively in practice, i. e. to reduce the numbers of findings in the long-term. Our approach is based on a notion of quality as defined in the ISO-25010 definition of software product quality. The ISO definition covers a large variety of quality aspects—from functional suitability over security to maintainability (Chapter 3). For each quality aspect, a variety of static analysis exists. Independent of the quality aspect, their findings have in common that they likely increase the non-conformance costs of the system. For example, programming faults can create costs for hotfixes or redeployment, security vulnerabilities can create costs for lost data, and low-quality code can increase costs for change requests.

The approach was designed such that its general idea can be applied to different static analyses, in principle. The specific instantiations in this thesis, however, focused on static analyses addressing correctness and maintainability as these two analysis types are most frequently used in practice. To derive our approach to cost-effectively use static analysis, several intermediate steps were required. Figure 8.1 repeats these steps. It further annotates which chapter or paper contained the corresponding contributions. In the following, we will summarize each step and its corresponding contribution.

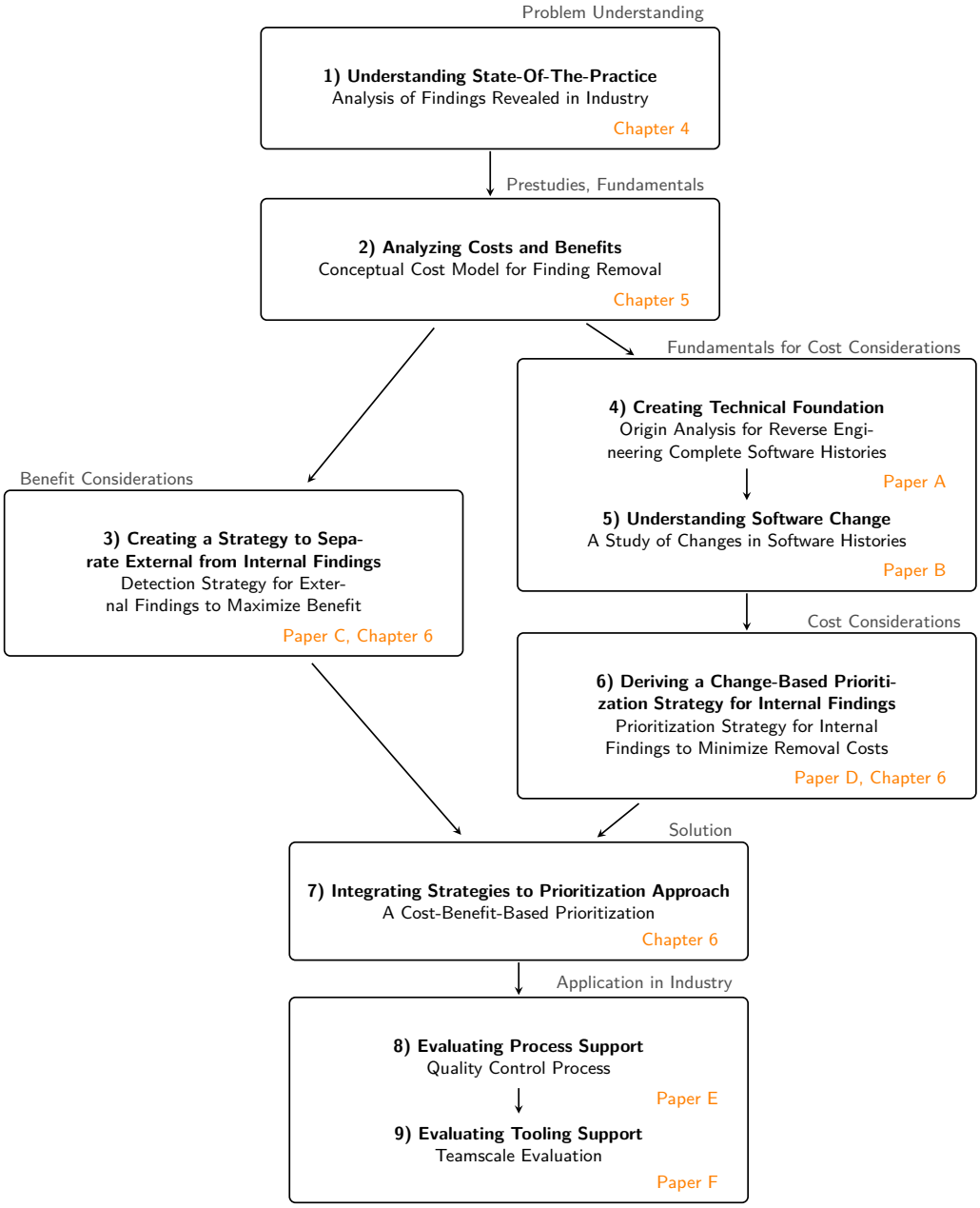


Figure 8.1: Summary of Approach

Understanding State-Of-The-Practice As a first step, we conducted an empirical study to analyze the state-of-the-practice of using static analysis in industry (Chapter 4). The study showed that most industry systems reveal only few external findings which can potentially lead to user-visible failure of the system. In contrast, only few maintainability analyses already resulted in thousands of internal findings addressing the (non-user-visible) internal quality. Overall, the number of findings in practice is too large to be fixed at once because quality assurance budget is usually limited. To effectively use the available

budget, we targeted to select those findings for removal which reveal the best cost-benefit ratio. To better understand costs and benefits associated with finding removal, we provided a conceptual cost model.

Contribution: Our large-scale quantitative study revealed state-of-the-practice of using static analysis and can serve other researchers as reference.

Analyzing Costs and Benefits To obtain a prioritization strategy for external and internal findings, we provided a cost-benefit analysis for finding removal in form of a conceptual cost model (Chapter 5). With an empirical study, we enriched the conceptual model with industrial evidence. Combined with results from existing work, this empirical data enabled an exemplary instantiation of the model. The instantiation provided an order of magnitude by showing that the refactoring of an overly large class (as one finding example) would pay off after few months based on our assumptions. However, some assumptions were weak. Thus, the results needs to be treated with caution.

Based on the cost model, we derived the following reasoning to prioritize findings: The benefit of removing external findings, i. e. eliminating a potential system failure, was quite obvious. Further, the model revealed that external findings are usually fast to remove. Due to the immediate reward, our approach prioritized the removal of external findings over the removal of internal ones. The small number of external findings and their low removal costs make this prioritization feasible. However, treating external and internal findings differently required an accurate differentiation mechanism between the two as a next step.

For internal findings, the model showed that they take more time to be removed than external findings. On the benefit side, the prediction was more difficult as the benefit of removing internal findings depends on future maintenance. With an empirical study, we showed that static analyses have limited impact in predicting future maintenance. We assumed that future maintenance is rather derived from changing requirements than from the existing code base or other artifacts. While we could not reliably predict the benefit automatically based on existing artifacts, we chose removal costs for internal findings to serve as prioritization basis: In the absence of immediate reward, developers are much more motivated to remove internal findings if the removal costs do not create additional burden. As outlined in our cost model, we can reduce costs if we closely align finding removal with ongoing changes during development. This saves overhead costs in terms of code understanding and testing as performing a change request already comprises both.

To align finding removal with software changes, we strove to better understand how developers perform them and to analyze the change history in an open source and industry case study. However, we found that history as recorded in version control systems is often not complete due to refactorings. To reverse engineer complete histories despite moves or refactorings and to obtain accurate results, we first had to lay out the technical foundation. We addressed this in a separate step.

Contribution: Our conceptual cost model provided first empirical data about costs and benefits associated with finding removal. Its instantiation is a first step towards a return-on-invest calculation for project management in industry. For researchers, it evaluates

current state-of-the-art and reveals weak assumptions when justifying static analysis as quality assurance technique that require future work.

Creating a Strategy to Separate External from Internal Findings Based on the results from our cost model, our approach prioritized the removal of external findings over the removal of internal findings. Hence, we needed to accurately differentiate between the two. In this thesis, we showed that differentiating between external and internal findings can be mostly done trivially based on the finding's category. However, we further showed that for some categories, e. g. code clones, the differentiation becomes more complex because findings of the same category can represent both internal and external findings. Hence, the differentiation has to be made for each finding individually. For the specific example of code clones, we provided a classification algorithm and evaluated it on an industrial data set (Paper **C**). The evaluation showed that our classification algorithm has higher precision than random sampling. However, the separation of external from internal clone findings still requires a manual inspection.

Having detected external findings, our approach suggested them for instant removal due to the immediate benefit. It remained to prioritize internal findings. Based on the results from the cost model, we targeted to align the prioritization of internal findings with ongoing software changes and, hence, needed a better understanding of how software is changed.

Contribution: We provided a strategy for developers to separate external from internal findings which helps to prioritize findings based on benefit.

Creating Technical Foundation To perform an accurate analysis of software changes, we first provided an origin analysis as technical foundation (Paper **A**). The origin analysis allowed us to obtain complete histories on file and method level even if renames, moves, or other refactorings occurred. The evaluation of precision and recall showed that the analysis was highly accurate. Additionally, it revealed that up to 38% of analyzed file histories in the version control system were incomplete due to at least one refactoring. Without our origin analysis, any analysis on the history of these files would not be accurate.

Contribution: Our origin-analysis is substantial for other researchers mining software histories to gain accurate results. Further, it enriches tool support for developers as it facilitates to obtain complete histories for code reviews, metric trend inspections, or change logs.

Understanding Software Change Based on the origin analysis, we conducted a large-scale empirical study to better understand how software evolves (Paper **B**). The study showed that software systems grow primarily by adding new methods. They do not grow through modification in existing methods. Many of the existing methods, in fact, remain unchanged. Based on these results, we provided a change-based prioritization for internal findings to reduce removal costs.

Contribution: We provided a theory of how software systems grow which can serve as foundation for other researchers.

Deriving a Changed-Based Prioritization Strategy for Internal Findings As software grows through new methods, we prioritized internal findings first in newly added code and, second, in lately modified code. As many existing methods remain unchanged, this prioritization, in fact, reduces the number of internal findings significantly. For highly dynamic development for which the number of findings in new code is still too large, we prioritized findings in new or changed code that can be removed with a fast refactoring. This minimizes the removal costs as outlined in our cost model. In particular, we provided a case study for the exemplary findings of code clones and long methods, showed how fast-to-refactor findings can be detected, and evaluated their acceptance amongst developers (Paper **D**): During individual interviews, developers confirmed that findings detected by our approach can be removed fast. Nevertheless, developers considered these removals not only fast, but still useful to achieve better maintainability.

Contribution: We provided a useful strategy for developers to prioritize internal findings based on their removal costs.

Integrating Strategies to a Prioritization Approach Based on our derived benefit-based strategy for external findings on the one hand and cost-based strategy for internal findings on the other, we formulated a prioritization approach (Chapter 6). The core of the approach comprises an automatic recommendation which prioritizes findings based on the previously gained insights: External findings should be removed immediately in each iteration as their removal provides immediate benefit of preventing costly system failures. Internal findings were prioritized based on their removal costs. We used the change-based prioritization to reduce removal costs by saving overhead in code understanding and testing. Additionally, for highly dynamic systems, we focused only on these findings that can be refactored fast—minimizing the overall removal costs.

The automatic recommendation of a subset of findings reduced the flood of findings to a reasonable size. In a second, manual inspection step, false positives and non-relevant findings were eliminated to not disturb and distract developers. Third, our approach was to be applied iteratively during development. This provides several advantages: One the one hand, newly created findings are considered. On the other hand, the set of selected analyses can be adjusted flexibly in between two iterations and also the current quality assurance budget can be taken into account for each iteration.

Main Contribution: We provided a prioritization approach that addresses current challenges of using static analysis. The approach helps developers to apply static analysis cost-effectively in practice.

Evaluating Process Support Our approach helps developers to select a subset of findings for removal. However, if management prioritizes change requests over quality improvements, developers still lack resources to actually remove findings. We suggested a quality-control process which provides the necessary transparency and impact analysis to convince management of the benefit of allocating resources for quality improvement. With an industrial evaluation, we showed quantitatively that the number of findings can be controlled and decreased even if the systems keep growing in size (Paper **E**). Qualitatively, the management

in our case study was convinced by the benefits of the process and decided to extend it to almost all of their development projects. This decision revealed the applicability, benefit, and impact of our process.

Contribution: With our quality control process, we showed how our prioritization can be applied effectively in industry and lead to measurable quality improvements.

Evaluating Tooling Support The quality control process revealed numerous requirements for static analysis tools to be able to support the process. As one example, our change-based prioritization required the differentiation between old findings and findings in new or modified code. Many existing tools analyze only a snapshot of a system and were, thus, not designed to provide this distinction. In this thesis, we evaluated Teamscale, a tool developed by our company, which provides findings tracking through all commits (Paper **F**). Hence, Teamscale can differentiate between newly added and old findings. Our evaluation based on a guided developer survey showed a high user acceptance rate of the tool. Many developers stated that the tool integrates well into their development process and would help them to write cleaner code.

Contribution: The evaluation of corresponding tool support complements the applicability of our quality control process in industry. Additionally, it provides directions for the development of future static analysis tools.

8.2 Future Work

The main focus of this thesis was the effective application of static analyses as quality assurance technique for long-lived systems. However, as discussed in Chapter 3 and 5, static analyses also have certain limitations. Also, our cost model revealed certain shortcomings in current state-of-the-art when arguing about the return-on-invest of static analyses. Based on these shortcomings, we derive the following directions for future work:

8.2.1 Benefit of Removing Internal Findings

For internal findings, our prioritization approach focused on the costs for removal. In order to better predict their benefit, we need more empirical evidence and prediction abilities for the variables in our benefit model (see Equation 5.7). Based on our model, removing findings is beneficial for the three main maintenance activities code reading, code modification, and code testing.

Increase in Readability and Changeability To better predict the benefit of removing a single finding, investigating the speed up of code reading and code modification requires future work. While existing work has tried to find empirical evidence for both, it was yet not able to reveal it consistently. Some case studies revealed an increase in developer's performance on cleaner code, while others did not. Even though we are convinced that this

speed up exists based on several anecdotal evidence from our customers, it is extremely difficult to prove scientifically.

Large research grants could enable controlled experiments with two functional equivalent systems—one being quality assured with static analysis and the other not. However, to analyze the speed up of removing certain findings in isolation, the two system would ideally need to have an identical setup in terms of development process, tools, testing, developers' skills, etc. To achieve scientifically sound results, this makes the study setup very complex and costly.

Prediction of Future Maintenance The benefit of removing internal findings does not only depend on the speed up per maintenance activity, but also on the number of maintenance activities to be performed in the future. Hence, sophisticated maintenance prediction models are required. In this thesis, we stated the assumption that existing code bases and their history can only provide limited information for the prediction of future changes because these are rather derived from changing requirements. It remains to investigate if different static information can improve the prediction model. Additionally, the biggest question is which other artifacts from areas of requirement engineering or marketing analysis can be included in the prediction. As this was out of scope of this thesis, we had to leave it for future work.

Increase in Testability Additionally, we stated in our cost model that removing findings can also be beneficial for subsequent testing activities, e. g. shorter methods might be easier to test with unit tests or less redundancy might reduce the number of required manual test runs. However, within the scope of this thesis, we were not able to provide any evidence for this. Future work is required to investigate the impact of cleaner application code on the testing expenses for a system.

Interference Effects between Findings Our model formalized costs and benefits of removing a single finding. However, first existing work indicates that the hampering impact on maintainability increases significantly if multiple findings affect the same code. Our model does not yet take these interference effects into account. Future work is required to investigate how much multiple findings amplify the negative impact on maintainability.

8.2.2 Other Prioritization Aspects

We provided a prioritization based on a cost-benefit consideration for removing external and internal findings. While external findings were to be removed immediately (maximizing benefit), the removal of internal findings was change-based (minimizing costs). Yet, for future work, we could consider other dimensions for the findings prioritization as well.

Social Aspects For future work, it might be worth investigating other prioritization aspects such as the criticality of the code, the centrality of the code [90], ownership of the code etc. A lot of questions are still to be answered here. To outline just a few: Are developers more willing to remove findings in critical and central code? Do certain parts of the code belong only to one developer? Would this developer be more willing to remove the findings? Do other social aspects influence the developer's motivation to produce clean code?

Learning from History With the origin analysis, our thesis provides a large technical contribution to track findings over time. Accurate findings tracking provides developers precise information about truly new findings as opposed to existing findings that were modified or moved. This is a big step towards using historical information from the repository to support developers containing the findings flood. Yet, the repository history can provide more useful information beyond the technical findings history. We can potentially learn from the history, when static analysis results correlated to software failures and when and why developers decide to remove certain findings.

A Appendix

A.1 Notes On Copyrights

IEEE Publications According to the consent of IEEE[1] to reuse papers in a PhD thesis, the author of the thesis is allowed to include Papers **B**, **C**, and **E** in the thesis:

> The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:
>
> Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:
>
> 1. The following IEEE copyright/ credit notice should be placed prominently in the references: ©[year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
>
> 2. Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
>
> 3. In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.
>
> If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

[1] http://ieeexplore.ieee.org/

ACM Publications According to the consent of ACM[2] to reuse papers in a PhD thesis, the author of the thesis is allowed to include Papers **A**, **D**, and **F** in the thesis. For each of these three papers, we obtained a license through the Rightslink portal of ACM. The obtained license states the following:

1. The publisher of this copyrighted material is Association for Computing Machinery, Inc. (ACM). By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at).

2. ACM reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.

3. ACM hereby grants to licensee a non-exclusive license to use or republish this ACM-copyrighted material[3] in secondary works (especially for commercial distribution) with the stipulation that consent of the lead author has been obtained independently. Unless otherwise stipulated in a license, grants are for one-time use in a single edition of the work, only with a maximum distribution equal to the number that you identified in the licensing process. Any additional form of republication must be specified according to the terms included at the time of licensing.

4. Any form of republication or redistribution must be used within 180 days from the date stated on the license and any electronic posting is limited to a period of six months unless an extended term is selected during the licensing process. Separate subsidiary and subsequent republication licenses must be purchased to redistribute copyrighted material on an extranet. These licenses may be exercised anywhere in the world.

5. Licensee may not alter or modify the material in any manner (except that you may use, within the scope of the license granted, one or more excerpts from the copyrighted material, provided that the process of excerpting does not alter the meaning of the material or in any way reflect negatively on the publisher or any writer of the material).

6. Licensee must include the following copyright and permission notice in connection with any reproduction of the licensed material: "[Citation] ©YEAR Association for Computing Machinery, Inc. Reprinted by permission." Include the article DOI as a link to the definitive version in the ACM Digital Library.

[2] http://dl.acm.org/

[3] Please note that ACM cannot grant republication or distribution licenses for embedded third-party material. You must confirm the ownership of figures, drawings and artwork prior to use.

7. Translation of the material in any language requires an explicit license identified during the licensing process. Due to the error-prone nature of language translations, Licensee must include the following copyright and permission notice and disclaimer in connection with any reproduction of the licensed material in translation: "This translation is a derivative of ACM-copyrighted material. ACM did not prepare this translation and does not guarantee that it is an accurate copy of the originally published work. The original intellectual property contained in this work remains the property of ACM."

8. You may exercise the rights licensed immediately upon issuance of the license at the end of the licensing transaction, provided that you have disclosed complete and accurate details of your proposed use. No license is finally effective unless and until full payment is received from you (either by CCC or ACM) as provided in CCC's Billing and Payment terms and conditions.

9. If full payment is not received within 90 days from the grant of license transaction, then any license preliminarily granted shall be deemed automatically revoked and shall be void as if never granted. Further, in the event that you breach any of these terms and conditions or any of CCC's Billing and Payment terms and conditions, the license is automatically revoked and shall be void as if never granted.

10. Use of materials as described in a revoked license, as well as any use of the materials beyond the scope of an unrevoked license, may constitute copyright infringement and publisher reserves the right to take any and all action to protect its copyright in the materials.

11. ACM makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

12. You hereby indemnify and agree to hold harmless ACM and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

13. This license is personal to the requestor and may not be sublicensed, assigned, or transferred by you to any other person without publisher's written permission.

14. This license may not be amended except in a writing signed by both parties (or, in the case of ACM, by CCC on its behalf).

15. ACM hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and ACM (and CCC) concerning

this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

16. This license transaction shall be governed by and construed in accordance with the laws of New York State. You hereby agree to submit to the jurisdiction of the federal and state courts located in New York for purposes of resolving any disputes that may arise in connection with this licensing transaction.

17. There are additional terms and conditions, established by Copyright Clearance Center, Inc. ("CCC") as the administrator of this licensing service that relate to billing and payment for licenses provided through this service. Those terms and conditions apply to each transaction as if they were restated here. As a user of this service, you agreed to those terms and conditions at the time that you established your account, and you may see them again at any time at http://myaccount.copyright.com

18. Thesis/Dissertation: This type of use requires only the minimum administrative fee. It is not a fee for permission. Further reuse of ACM content, by ProQuest/UMI or other document delivery providers, or in republication requires a separate permission license and fee. Commercial resellers of your dissertation containing this article must acquire a separate license.

Bibliography

[1] Definition: Thought experiment. https://en.wikipedia.org/wiki/Thought_experiment. Accessed: 2015-09-07.

[2] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[3] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011.

[4] E. Ammerlaan, W. Veninga, and A. Zaidman. Old Habits Die Hard: Why Refactoring for Understandability Does Not Give Immediate Benefits. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2015.

[5] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11), 2014.

[6] N. Ayewah and W. Pugh. The google findbugs fixit. In *Proceedings of the 19th ACM International Symposium on Software Testing and Analysis*, 2010.

[7] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2007.

[8] R. Baggen, J. P. Correia, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2), 2012.

[9] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014.

[10] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, 2000.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2), 2010.

[12] B. W. Boehm. *Software Engineering Economics*. 1981.

[13] B. W. Boehm, J. Brown, H. Kaspar, M. Lipow, G. McLeod, and M.Merritt. *Characteristics of Software Quality*. 1978.

[14] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative Evaluation of Software Quality. In *Proceedings of the 2Nd International Conference on Software Engineering (ICSE)*, 1976.

[15] C. Boogerd and L. Moonen. Prioritizing software inspection results using static profiling. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 2006.

[16] C. Boogerd and L. Moonen. Ranking software inspection results using execution likelihood. In *Proceedings Philips Software Conference*, 2006.

[17] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the ACM FSE/SDP Workshop on Future of Software Engineering Research*, 2010.

[18] F. Buschmann. To pay or not to pay technical debt. *IEEE Software*, 28(6), 2011.

[19] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, (6), 2004.

[20] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, 21(2), 2013.

[21] P. B. Crosby. *Quality is free: the art of making quality certain*. 1979.

[22] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2), 1993.

[23] M. D'Ambros, A. Bacchelli, and M. Lanza. On the impact of design flaws on software defects. In *Proceedings of the 10th International Conference on Quality Software (ICQS)*, 2010.

[24] F. Deissenboeck. *Continuous Quality Control of Long-Lived Software Systems*. PhD thesis, Technische Universität München, 2009.

[25] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 2008.

[26] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3), 2006.

[27] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J. Girard. An activity-based quality model for maintainability. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2007.

[28] G. R. Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2), 1995.

[29] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman. Does god class decomposition affect comprehensibility? In *Proceedings of the IASTED Conference on Software Engineering*, 2006.

[30] S. Eder, M. Junker, E. Jurgens, B. Hauptmann, R. Vaas, and K. Prommer. How much does unused code matter for maintenance? In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[31] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Software*, 2001.

[32] P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217, 2008.

[33] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3), 2000.

[34] D. Falessi and A. Voegele. Validating and prioritizing quality rules for managing techncial debt: An industrial case study. *MTD 2015-UNDER REVISION*, 2015.

[35] W. Fenske and S. Schulze. Code smells revisited: A variability perspective. In *Proceedings of the 9th ACM International Workshop on Variability Modelling of Software-intensive Systems*, 2015.

[36] D. A. Garvin. What does product quality really mean. *Sloan management review*, 26(1), 1984.

[37] R. L. Glass. Maintenance: Less is not more. *IEEE Software*, 15(4), 1998.

[38] S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2008.

[39] S. Heckman and L. Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4), 2011.

[40] L. Heinemann, B. Hummel, and D. Steidl. Teamscale: Software Quality Control in Real-Time. In *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE)*, 2014. ©2014 Association for Computing Machinery, Inc. Reprinted by permission. http://doi.acm.org/10.1145/2591062.2591068.

[41] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. An empirical study of global software development: distance and speed. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 2001.

[42] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-Based Code Clone Detection: Incremental, Distributed, Scalable. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2010.

[43] ISO. Standard 9126 for software engineering – Product quality – Part 4: Quality in use metrics. Technical report, ISO, 2004.

[44] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technical report, 2010.

[45] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th IEEE International Conference on Software Engineering (ICSE)*, 2013.

[46] P. Johnson. Requirement and design trade-offs in hackystat: An in-process software engineering mea- surement and analysis system. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*, 2007.

[47] E. Juergens. *Why and How to Control Cloning in Software Artifacts*. PhD thesis, Technische Universität München, 2011.

[48] E. Juergens and F. Deissenboeck. How much is a clone? In *Proceedings of the 4th International Workshop on Software Quality and Maintainability (WoSQ)*, 2010.

[49] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[50] H.-W. Jung, S.-G. Kim, and C.-S. Chung. Measuring software product quality: A survey of iso/iec 9126. *IEEE Software*, (5), 2004.

[51] J. Juran and F. Gryna. *Quality Control Handbook*. 1988.

[52] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, (3), 2012.

[53] F. Khomh, M. D. Penta, and Y.-G. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, 2009.

[54] S. Kim and M. D. Ernst. Which warnings should i fix first? In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 45–54. ACM, 2007.

[55] B. A. Kitchenham, G. H. Travassos, A. von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. Towards an ontology of software maintenance. *Journal of Software Maintenance*, 11(6), 1999.

[56] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *TSE*, 32(12), 2006.

[57] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, 2006.

[58] A. Kovács and K. Szabados. Test software quality issues and connections to international standards. *Acta Universitatis Sapientiae, Informatica*, 5(1), 2013.

[59] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: from metaphor to theory and practice. *IEEE Software*, (6), 2012.

[60] R. Kumar and A. V. Nori. The Economics of Static Analysis Tools. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[61] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the European Workshop on Software Process Technology (EWSPT)*, 1996.

[62] J.-L. Letouzey. The sqale method for evaluating technical debt. In *Proceedings of the IEEE Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[63] B. P. Lientz, P. Bennet, E. B. Swanson, and E. Burton. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. 1980.

[64] K. Lochmann and L. Heinemann. Integrating Quality Models and Static Analysis for Comprehensive Quality Assessment. In *Proceedings of 2nd International Workshop on Emerging Trends in Software Metrics (WETSoM)*, 2011.

[65] R. Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5), 2012.

[66] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 2008.

[67] T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings of the 4th ACM International Workshop on Principles of Software Evolution (IWPSE)*, 2001.

[68] H. H. Mui, A. Zaidman, and M. Pinzger. Studying late propagations in code clone evolution using software repository mining. *Electronic Communications of the EASST*, 63, 2014.

[69] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th International Conference on Software engineering (ICSE)*, 2005.

[70] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk. Preliminary results on using static analysis tools for software inspection. In *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2004.

[71] J. T. Nosek and P. Palvia. Software maintenance management: changes in the last decade. *Journal of Software Maintenance*, 2(3), 1990.

[72] A. Nugroho, J. Visser, and T. Kuipers. An empirical model of technical debt and interest. In *Proceedings of the 2nd ACM Workshop on Managing Technical Debt (MTD)*, 2011.

[73] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2015.

[74] D. L. Parnas. Software aging. In *Proceedings of the 16th International Conference of Software Engineering (ICSE)*, 1994.

[75] G. Pomberger and W. Pree. *Software Engineering - Architektur-Design und Prozessorientierung (3. Aufl.)*. 2007.

[76] A. Porter, H. P. Siy, C. Toman, L. G. Votta, et al. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Transactions on Software Engineering*, 23(6), 1997.

[77] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2004.

[78] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th ACM International Conference on Software Engineering*, 2008.

[79] J. Sappidi and S. A. Curtis, Bill. The crash report - 2011/12 (cast report on application software health).

[80] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul. Software engineering metrics for COTS-based systems. *Computer*, 34(5), 2001.

[81] H. Siy and L. Votta. Does the modern code inspection have value? In *Proceedings of the IEEE international Conference on Software Maintenance (ICSM)*, 2001.

[82] S. A. Slaughter, D. E. Harter, and M. S. Krishnan. Evaluating the cost of software quality. *Commun. ACM*, 41(8), 1998.

[83] Q. D. Soetens, J. Pérez, S. Demeyer, and A. Zaidman. Circumventing refactoring masking using fine-grained change recording. In *Proceedings of the 14th ACM International Workshop on Principles of Software Evolution (IWPSE)*, 2015.

[84] D. Steidl and F. Deissenboeck. How do java methods grow? In *Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2015. Copyright ©2015 IEEE. Reprinted, with permission.

[85] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler. Continuous Software Quality Control in Practice. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014. Copyright ©2014 IEEE. Reprinted, with permission.

[86] D. Steidl and S. Eder. Prioritizing Maintainability Defects by Refactoring Recommendations. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, 2014. ©2014 Association for Computing Machinery, Inc. Reprinted by permission. http://doi.acm.org/10.1145/2597008.2597805.

[87] D. Steidl and N. Göde. Feature-based detection of bugs in clones. In *Proceedings of the 7th ICSE International Workshop on Software Clones (IWSC)*, 2013. Copyright ©2013 IEEE. Reprinted, with permission.

[88] D. Steidl, B. Hummel, and E. Juergens. Quality Analysis of Source Code Comments. In *Proceedings of the 21st International Conference of Program Comprehension (ICPC)*, 2013.

[89] D. Steidl, B. Hummel, and E. Juergens. Incremental Origin Analysis of Source Code Files. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014. ©2014 Association for Computing Machinery, Inc. Reprinted by permission. http://doi.acm.org/10.1145/2597073.2597111.

[90] D. Steidl, B. Hummel, and E. Jürgens. Using network analysis for recommendation of central software classes. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, 2012.

[91] W. Suryn, A. Abran, and A. April. Iso/iec square. the second generation of standards for software product quality, 2003.

[92] A. K. Tripathi and A. Gupta. A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for java programs. In *Proceedings of the 18th ACM International Conference on Evaluation and Assessment in Software Engineering*, 2014.

[93] A. van Deursen and L. Moonen. The video store revisited – thoughts on refactoring and testing. In *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2002.

[94] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. *Refactoring test code.* 2001.

[95] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, 2002.

[96] R. D. Venkatasubramanyam and S. G. R. Why is dynamic analysis not used as extensively as static analysis: An industrial study. In *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices (SER&IPs)*, 2014.

[97] A. Vetro, M. Torchiano, and M. Morisio. Assessing the precision of findbugs by mining java projects developed at a university. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010.

[98] A. Von Mayrhauser et al. Program comprehension during software maintenance and evolution. *IEEE Computer*, (8), 1995.

[99] S. Wagner. A literature survey of the quality economics of defect-detection techniques. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, 2006.

[100] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. *Lecture Notes in Computer Science*, 3502, 2005.

[101] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb, and J. Streit. The quamoco product quality modelling and assessment approach. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[102] F. Wedyan, D. Alrmuny, and J. Bieman. The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction. In *Proceedings of the International Conference on Software Testing Verification and Validation (ICST)*, 2009.

[103] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012.

[104] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, 2013.

[105] A. Yamashita and L. Moonen. To what extent can maintenance problems be predicted by code smell detection? - an empirical study. *Inf. Software Technology*, 55(12), 2013.

[106] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME.

[107] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Proceedings of the IEEE Computer Society's Second International Computer Software and Applications Conference (COMPSAC)*, 1978.

[108] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, M. Vouk, et al. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4), 2006.