Prioritizing Maintainability Defects based on Refactoring Recommendations[*]

Daniela Steidl
CQSE GmbH
steidl@cqse.eu

Sebastian Eder
Technische Universität München
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.

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.

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.

[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.

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.

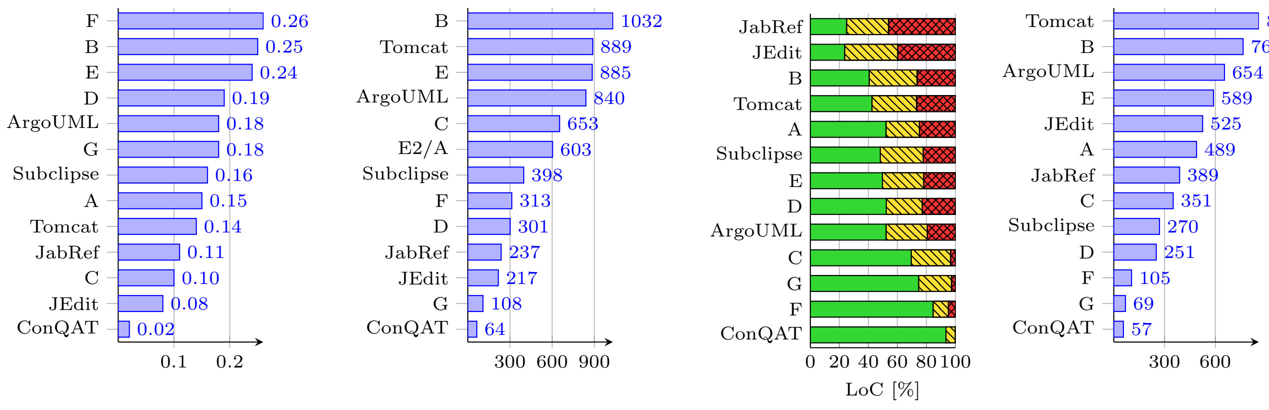
5. RESULTS

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.