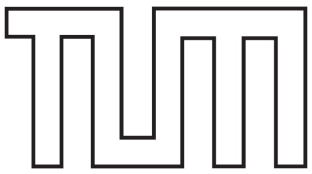


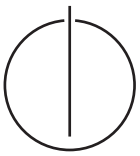
DEPARTMENT OF INFORMATICS

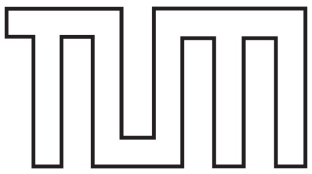
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Deriving Refactoring Suggestions from Quality Analysis Findings

Roman Haas





DEPARTMENT OF INFORMATICS

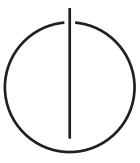
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Deriving Refactoring Suggestions from Quality Analysis Findings

Ableiten von Refactoring-Vorschlägen auf Basis von Qualitätsdefiziten

Author:	Roman Haas
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisors:	Dr. Elmar Jürgens
	Dr. Benjamin Hummel
	Dr. Christian Pfaller
Submission Date:	December 15, 2014



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, December 15, 2014 Roman Haas

Abstract

Software quality analysis results in findings to give hints where the quality of the code can be improved. Refactorings can be applied to solve some findings. To apply a refactoring, developers currently need to spend a considerable amount of time to find the most appropriate refactoring candidate, i.e. the most appropriate lines of code that should be refactored, before they can use tools that apply their preferred refactoring. This bachelor's thesis addresses the problem of finding the most appropriate refactoring candidate and presents an approach to derive extract method refactoring suggestions for C-like programming languages from quality analysis findings, especially long methods, with the aim to reduce code complexity. The approach first determines extractable candidates for too long methods. To find the most appropriate candidate, a scoring function is used that considers the following information about the candidates: length and nesting reduction, number of input and return parameters and the number of blank lines or comments before and after the candidate. The approach puts special stress on nesting for highly-nested methods. The prototype of this approach visualizes the best candidates using markers that are displayed beside the corresponding lines of code. The evaluation of this work on Java open source systems with ten participants in an online survey shows that the participants would actually apply 86% of the suggestions of the prototype for an extract method refactoring for the ten study objects.

Zusammenfassung

Findings sind das Ergebnis von Softwarequalitätsanalysen. Sie weisen auf Codestellen hin, bei denen die Codequalität verbessert werden kann. Einige dieser Findings können mithilfe von Refactorings behoben werden. Um ein Refactoring anwenden zu können, müssen Entwickler bei aktuellen Tools zunächst die Codezeilen ermitteln, auf denen sie ein Refactoring anwenden möchten. D. h. sie müssen den von ihnen favorisierten Refactoring-Kandidaten bestimmen, bevor das Refactoring dann mit Toolunterstützung automatisch durchgeführt werden kann. Diese Bachelorarbeit behandelt das Problem der Wahl des passendsten Refactoring-Kandidaten. Sie präsentiert einen Ansatz, um Vorschläge für das Extract Method Refactoring für C-ähnliche Programmiersprachen auf der Basis von Qualitätsdefiziten, insbesondere für zu lange Methoden, abzuleiten. Zunächst werden extrahierbare Kandidaten ermittelt und mit Punkten bewertet, nach denen die Kandidaten sortiert werden. Punkte werden für die Reduktion der Länge und der Verschachtelung, die Zahl der Ein- und Rückgabeparameter sowie für die Zahl der leeren bzw. kommentierten Zeilen vor und nach dem Kandidaten vergeben. Bei tief-verschachtelten Methoden werden besonders viele Punkte für die Reduktion der Verschachtelung vergeben. Es wurde ein Prototyp entwickelt, der zur Visualisierung der besten Kandidaten Markierungen neben den entsprechenden Codezeilen nutzt. Für die Evaluation dieser Arbeit wurde eine Onlineumfrage durchgeführt, bei der die zehn erfahrenen Teilnehmer Vorschläge für Java Open Source Systeme bewerten. 86% der Vorschläge für die zehn präsentierten Methoden aus den Open Source Systemen würden von den Teilnehmern für ein Extract Method Refactoring angewendet werden.

Contents

1. Introduction

Software quality plays an important role for the success of software projects. According to ISO/IEC 25010 (former ISO/IEC 9126) the most important software quality characteristics are functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability [ISO]. Lack of software quality may cause enormous costs during the whole software life cycle, e.g. during development, testing, and maintenance. Low-quality software is error-prone which means that additional time needs to be spent to solve problems. In addition, maintainability is very important for large software systems, as maintenance can make up to 90% of the total costs during the software life cycle [Erl00].

If no countermeasures are taken, code quality decays over time [Eic+01]. To be able to take countermeasures and improve the quality of software, one has to measure the quality and find code that lowers it. Measuring software quality is a big research area and there are many different approaches to measure the quality of software: One could rely on static analyzers, unit testing frameworks, metrics tools, violation checkers, and other techniques [Dei+08]. But doing only the measurement of software quality does not improve the quality.

[Fow99] introduced the concept of *bad smells*, which describes source code that can be improved such that maintenance and future (re)use become easier. *Refactoring* is a countermeasure against a bad smell: It is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing the observable behavior of the software. But doing refactorings manually is complicated, time-intensive, tedious, and error-prone, especially in large software systems [Opd92]. There are many tools available that support developers with semi-automated refactoring tools, like Eclipse, NetBeans or IntelliJ IDEA [Mar+10]. But even with tool support, a developer needs to select some source code within a method that he would like to extract and if the selected code is extractable, he enters a name for the new method. The refactoring itself will be done by the integrated development environment (IDE).

Quality Analysis tools provide information about code sequences that could be improved. These so-called *findings* have different categories and for each category there are common ways to fix the finding. For example, the open source software quality analysis tool ConQAT [1] can be used to find code anomalies. The finding severeness

[1] `http://www.conqat.org`

varies and for some findings it is harder to identify a solution than for others. Table A.1 in the appendix shows some findings that can be easily resolved. For example, to be able to solve the problem of empty blocks in a code segment, the developer only needs to delete the parenthesis ('{' and '}'). Similarly, it is easy to fix unused imports as only the `import`-statement needs to be removed. That means that there are findings for which tools, that fix these findings automatically, can be easily implemented.

The second table in the appendix, table A.2 shows examples of findings where several solutions may be appropriate, so usually a developer needs to review the finding. An example for that may be a *variable-assignment never read finding*. A static analysis tool can not distinguish between two cases: Either the module is loaded dynamically into a program and therefore the variable is actually read, so the finding is a false-positive. Or the variable is indeed never read during run time so that the finding should be removed. This distinction needs to be done by developers and tool-support for this task can not be implemented easily.

The last group of findings considered here, are those that are more severe than many other ones and should be fixed. Usually, it is not that trivial to decide *how* to fix such findings. The third table in the appendix, table A.3 gives some examples. Code clones may introduce copy&paste faults and make software harder to maintain. To resolve a clone, there is the option to extract the code into a method and to put this method in a utility class or to pull it up in the super class.

Applying refactorings is a good way to improve code quality which makes it suitable to resolve findings. To be able to use the semi-automated tools provided by the development environment, the developer needs to analyze the code and think about sensible refactorings. This work presents an approach to suggest extract method refactorings automatically based on software quality analysis findings.

1.1. Problem Statement

Current software quality analysis tools draw the developer's attention to bad smells and sometimes there is some general information about the finding that explains why this finding lowers software quality and what could be done to remove the bad smell. Long methods are an example for bad smells. They are hard to maintain because they can not be understood easily, they are reusable only at very coarse granularity, and result in imprecise profiling information [SE14]. To remove such a bad smell, developers have to find individual solutions for given problems, e.g. a method can be shortened by extracting some code lines into a new method, that is the *extract method* refactoring.

The extract method refactoring is one of the most important refactorings as it is often possible to do further refactorings after extracting methods [Fow99]. In practice,

it is one of the most often applied refactorings [WKK07] and is supported by many tools [MB08] in a semi-automated way. Semi-automated means in this context that the developer needs to identify a code segment that can be extracted, select it, and use a tool that extracts the selected code into a new method and replaces the selected code in the original method by a call of the new method.

The majority of development tools provide only semi-automated refactorings. That means that the first step to resolve a finding still needs to be done by developers: Before a refactoring can be applied, developers need to determine a valid and appropriate candidate. This candidate needs to fulfill several preconditions. Current research has shown that developers are not using those semi-automated refactoring tools as much as they could [MB07] because errors triggered by the refactoring progress are badly communicated [MB08] and developers still need to spend a considerable amount of time to find valid and appropriate refactoring candidates. To reduce the time needed to resolve findings, tools are needed that support developers by suggesting candidates that are valid and appropriate in the given context.

1.2. Contribution

This work presents an approach to find automatically refactoring suggestions based on software quality analysis findings, which makes it much easier for developers to find sensible refactorings. The approach focuses on *long method* findings and suggests extract method candidates to reduce code complexity by considering control flow and structural information of the specific method. The candidates are ranked using a scoring function that considers reduction of length and nesting, information about the parameters that are needed by the candidate, and commented or blank lines before and after the candidate. The approach is implemented as a prototype using the quality analysis tool ConQAT and is evaluated by ten experienced and independent programmers that are experts in the field of software quality. Therefore, an online survey that considers ten too long methods of three Java open source systems are used. The evaluation shows that for the survey objects 86% of the suggestions that were generated by the prototype would have been applied by the participants.

1.3. Outline

The remainder of this work is organized as follows. Foundations on the extract method refactoring and terms and definitions that are needed in the context of this bachelor's thesis are given in section 2. Section 3 concludes related work to refactoring recommendations, especially for extract method and presents several ways how such

recommendations can be generated. Section 4 presents an approach that combines several advantages of these approaches to be able to generate automatically extract method refactoring suggestions. Implementation details of the prototype are given in section 5. The approach is evaluated on the basis of seven open source systems in section 6. Finally, section 7 presents conclusions and suggests future work.

2. Foundations

This chapter will provide the foundations that are necessary for this work. First, general information about the extract method refactoring is provided to give an overview about the refactoring for which suggestions will be generated. In the second part of this chapter, terms and definitions that are needed in the context of this work are presented.

2.1. Extract Method Refactoring

Extract method means the extraction of a code fragment into a new method of the same class, replacing the extracted code fragment by a call of the extracted method. Detailed information about the extract method refactoring can be found in [Mar+10] and [Opd92], the most important pre- and postconditions and the general process of the refactoring are summarized below.

Preconditions. To be able to execute an extract method refactoring, the following preconditions must hold. The selected code fragment is valid (i.e. valid selection of statements and compilable code, even if the selected code is removed). If more than one local variable or parameter is assigned in the fragment, at most one of them is read in the control flow of the original method after the execution of the extracted code fragment. The fragment does not contain conditional returns: It always returns or always flows from the beginning to the end and it does not jump outside of itself.

Process. During the refactoring process the following steps need to be done. First, add a method skeleton to the class with a name that is specified by the user, then add arguments to the new method (these are the local variables that are used but not defined in the fragment). Second, move the fragment from the original method to this new method (and if necessary, add a return type to the signature of the new method and a return instruction to the method body). Third, replace the extracted code fragment with the appropriate method call, passing the proper parameters.

Postconditions. After an extract method refactoring the following postconditions hold. The class has a new method, where the input parameters are those local variables

that are used but not defined in the extracted fragment. The extracted method has the correct return type and the remainder of the original method contains a call to the new method instead of the extracted code. If necessary, the return value of the extracted method is assigned to a variable in the remainder of the original method.

The focus of this work lies on the automated way of finding candidates that can be used to refactor a too long and therefore complex method by applying an extract method refactoring. This means that the most important aims of this work are first, finding valid candidates and second, suggesting the best ones, and therefore, the preconditions that must hold for the suggested candidates are much more important for this work than the general process of the refactoring or the postconditions. The refactoring itself can be done easily using existing tools (that are, for example, integrated in the IDE), which implement the process and ensure the postconditions.

2.2. Terms and Definitions

This section provides terms and definitions for three important categories. First, a naming convention for the relevant parts of the extract method refactoring is given. The second subsection shows which two metrics are considered to indicate complexity of methods. The last subsection will show which constructs are used to represent methods and statements.

2.2.1. Terms Considering the Extract Method Refactoring

Listing 2.1 shows an example (provided by [Fow99]) of a method that can be refactored using an extract method refactoring. Listing 2.2 shows the two methods that are the result of an extract method refactoring.

In this work, the *original method* is the method before any refactoring was applied, i.e. the method in Listing 2.1 is considered as original method. The *refactoring candidate* is the sequence of statements that will be extracted into a new method, in listing 2.1 these are the statements in line 5 and 6. The *remainder* of the (original) method are those statements that will last in the method, from which the refactoring candidate was extracted. The remainder includes the call of the extracted method (and, if necessary, assigning the return value of the method to some variable). In listing 2.2 the statements in line 2 and 3 are the remainder of the original method.

```
1 void printOwing() {
2    printBanner();
3
4    //print details
5    System.out.println ("name: " + _name);
6    System.out.println ("amount " + getOutstanding());
7 }
```

Listing 2.1: Code before an Extract Method Refactoring is applied

```
1 void printOwing() {
2    printBanner();
3    printDetails(getOutstanding());
4 }
5
6 void printDetails (double outstanding) {
7    System.out.println ("name: " + _name);
8    System.out.println ("amount " + outstanding);
9 }
```

Listing 2.2: Code after an Extract Method Refactoring is applied

2.2.2. Complexity Indicators

The approach presented in this work tries to suggest candidates for extract method refactorings that reduce complexity. [Hum] suggests length and nesting of methods as indicators for complex methods. Those two criteria will be used to quantify the complexity of a method before and after a possible refactoring.

Length. The *length* of a method is the number of lines of code (LoC) it has. This work does not consider the amount of statements as length because a method that has many long comments is probably also as complex and hard to understand as a method that has more statement and less comments but the same LoC. As far as complexity is concerned, an important question is, when a method actually is *too long*. Martin claims in his book [Mar09] that an old rule of thumb for the length of methods is that it should not be so long that it can not be displayed at once on a smaller monitor. As complexity of code in general increases with its length, he recommends to write methods as short as possible, which means that a method should not process more than one thing. As it is not easy to specify what actually one thing is, something else is needed to decide

whether a method is too long. In a study, [Män05] found out that LoC is the most important predictor variable for the smell long method, which makes it a considerable metric to decide whether a method is too long. Mäntylä does not provide a limit for the length but [SE14] do for their research concerning the prioritization of findings. They define a method as too long if it counts more than 40 LoC. This work follows their suggestion and assumes that methods in C-like programming languages are too long if they count more than 40 LoC.

Nesting. The second indicator of complexity considered by this work is nesting. Nesting depth is a common metric and describes the maximal value of nesting depth of control statements in a method (i.e. branching statements like loops, conditions, and gotos) [Piw82]. Therefore, for this work, the minimal nesting depth of a method is zero. An example for a method that has nesting depth 0 is the method in listing 2.1.

The next definition uses the nesting depth of statements. That is the value of nesting depth of the last branching statement before the considered statement in the control flow graph (CFG), or 0, if there is no such branching statement. CFGs were introduced by [All70] and a CFG is defined as a directed graph in which the nodes represent statements and the edges represent control flow paths (that result from branching or loops)

This work also considers the *nesting area* of a method as a metric for method complexity. Intuitively spoken, nesting area is the area under the single statements of pretty printed code.

Definition. *Let $NestDepth(S)$ be the nesting depth of a statement S. The nesting area of a method M with the statements $S_i, i \in (1, n)$, where n is the number of statements in the method, is defined as*

$$NestArea(M) = \sum_{i=1}^{n} NestDepth(S_i)$$

2.2.3. Representation

There are several ways that can be used to represent code for analysis. This work uses control and data flow graphs to represent methods and the nodes of the graph to represent statements. Refactoring candidates will consist of a set of (compound) statements.

Control and Data Flow Graph (CDFG). To represent methods this work uses control and data flow graphs (CDFGs). This work will use the implementation of CDFGs from Fabian Streitel. These CDFGs base on CFGs and as he points out in [Str14], he

adds edges to a CFG that connect the definition (or assignment) of a variable to all its uses and a use to all possible definitions. Based on this graph, it is possible to find out, which variables are when defined, read, and written. This information will be necessary to be able to compute the set of input and return parameters that are needed for an extract method refactoring candidate.

Compound Statements. The approach considers single *statements* and *compound statements*. Compound statements contain a (possibly empty) set of single statements and other compound statements. They represent branching and looping statements and even all statements in a method are represented by a compound statement. That means that a compound statement that represents a whole method usually has several child compound statements. For example, a compound statement that represents an `if`-statement contains always the statements that are executed if the condition of the `if`-statement evaluates to `true`.

3. Related Work

The field of code refactorings is a large research area. There is some research on refactoring recommendations, especially for extract method, that will be concluded in this chapter. The approaches in the literature can be divided into four categories. Some use program slicing techniques to find recommendations for refactoring (section 3.1), others try to find suitable suggestions from graph representations of the code (section 3.2). Some rely on scoring functions to find the most appropriate refactoring candidate (section 3.3) and finally, there is some research on the topic of refactoring prioritization (section 3.4).

3.1. Suggestions based on Program Slicing

Maruyama presents in [Mar01] a semi-automated mechanism for refactoring suggestions. First, the developer selects a variable of interest at a specific program point at the source method, his tool then calculates a set of refactoring candidates from which the user can choose one. Finally, the user can specify a name for the extracted method. The mechanism decomposes the control flow graph (CFG) using block-based program slicing. The approach is adapted and implemented in the tool JDeodorant by [TC09] that improves behavior preservation. An overview about different slicing techniques gives [MMK06].

According to [Sha12], program slicing has the following drawbacks: First, approaches that use program slicing techniques can not be fully automated as the user has to select a slicing criterion for every method that should be refactored. Second, the suggestions of program slicing based approaches depend completely on the user's input. If the user selects an unfavorable variable, suboptimal refactoring suggestions may be made such that the refactoring decreases code understandability and maintainability. This means that the success of the implemented approach depends on the developer's experience and the complexity of the software system. In addition, program slicing pays no attention to the fact that logical blocks of statements can be inter-related with each other and therefore, a program slicer may suggest a non-proper extraction.

Physical statement blocks may have multiple logical cohesive blocks and therefore, block based slicing may lead to no optimal extraction suggestion. Another drawback of program slicing based approaches is the need of a program slicer. Slicers are language

dependent and even harder to implement than a full-functional parser. Much time and effort are needed to have a solid program slicer that can be used to implement an approach for extract method suggestions. These are the reasons why the approach presented in this work does not rely on a slicer. But the algorithm to generate refactoring candidates uses the CDFG in a similar way like Maruyama.

3.2. Graph Based Approaches

Another way to find extract method refactoring candidates is to use information from graphs that represent the method which should be refactored. Sharma provides in [Sha12] a mechanism to propose extract method candidates based on a *data and structure dependency graph (DSD)*. The vertices $V = (V_d, V_s)$ consist of the set V_d of vertices that represent source code statements and the set V_s of vertices that represent blocks of statements. The set $E = (E_d, E_s)$ contains the edges of the DSD, where $E_d \in \{(u,v) \mid u,v \in V_d\}$ and $E_s \in \{(u,v) \mid (u \in V_d \wedge v \in V_s) \vee (u \in V_s \wedge v \in V_d)\}$. An edge $e_d \in E_d$ exists if there is a data dependency between the vertices u and v. An edge $e_s \in E_s$ exists if there is a structural dependency between the vertices u and v, i.e. the one statement depends on the structure of the other statement. The longer an edge, the lower is the likelihood that the statements represented by the vertices at the ends of the edge will stay together in a method. Sharma's approach identifies an extract method candidate by deleting the longest dependency edge in the DSD which results in two disconnected subgraphs. Those subgraphs represent the statements that stay within the original method or will be extracted to a new method, respectively.

The approach is not implemented by Sharma and hence not evaluated. An advantage of this representation of methods is that non-continuous refactoring candidates can be found. Sharma's approach considers, in addition to the structural information, the dependencies between single statements and recommends the set of statements that can be separated from the rest of the original method by deleting the lowest dependency. The approach does not take into account how many parameters are needed to extract the selected statements, that means candidates may have many parameters which introduces another code smell ("Long Parameter List", mentioned by [Fow99]) into the code. To avoid this effect the scoring function of the approach given in this work punishes candidates for having many parameters such that they are not recommended for extract method refactoring. This work focuses on other criteria to select candidates: Beside parameters it also considers the reduction of length and nesting, and structural information (comments and blank lines).

Kanemitsu et al. use in [KHK11] a program dependency graph (PDG), which represents statements as nodes and the dependencies between them as edges. Their approach was led by the design principle that one method should process only one thing. They implement the principle by setting the length of an edge as the intensity of data dependency. That means that the shorter an edge, the more probable it is that the two nodes stay together in the same method. The automatic recommendation was not part of the original implementation and is thus optional. It recommends to extract all nodes that are connected via edges that are not longer than a maximal length, which is specified by the user. A disadvantage of this approach is that it is not fully-automated as the user needs to specify the upper bound, which may also lead to non-optimal extract method recommendations. Kanemitsu's evaluation included a comparison between his tool ReAF and JDeodorant, developed by [TC09]. As a result of that comparison, Kanemitsu claims that the users are faster if they get refactoring suggestions automatically and it is helpful for users to see how the recommended method would look like as to decide whether they want to apply the refactoring. He implemented those improvements but did not evaluate them.

The approach presented in this work suggests a fully-automated way to find candidates for an extract method refactoring and suggests the best candidates that are determined by a scoring function. It tries to suggest candidates that reduce code complexity, taking structural information into account. That is more general than suggesting candidates such that methods fulfill the guideline of processing only one thing. Finally, the prototype presented in this work is evaluated by ten experienced developers on ten methods from three Java open source systems.

3.3. Score Based Approaches

There are also some approaches that use a scoring function to determine extract method refactoring candidate suggestions. Silva et al. suggest in [STV14] an approach to automatically generate candidates for method extraction that is inspired by the *minimize coupling/ maximize cohesion* design guideline. The input is a method that should be refactored, the output a list of the most relevant refactoring candidates. At the beginning, their algorithm splits the given method into hierarchical blocks of continuous statements. These blocks are used to generate a list with sub-sequences of statements that can be extracted from each block. Checking whether the candidates can be extracted leads to a list of refactoring candidates, which is ranked by using a scoring function. Their scoring function considers the static dependencies between variables, types, and packages. Their evaluation shows that the approach achieves a precision and recall of at least 38% if only the best candidate is considered. An

advantage of their approach is that it is based on a well-known and accepted design guideline. A disadvantage on the other hand is that they are not able to extract non-continuous sequences of statements. The approach discussed in this work was inspired by Silva et al. as the general procedure of candidate generation is quite similar and the approach presented in this work is also not able to suggest candidates with non-continuous statements. But there is a big difference in the scoring function as the discussed approach does not consider dependencies but structural information and the reduction of length and nesting with the aim to reduce code complexity and increase maintainability.

Yang et al. consider in [YLN09] long methods and suggest an approach to recommend refactorings that have an as small coupling as possible, automatically. As basis they only use long methods. Each method is split up in hierarchical fragments whereas blank lines, iterations, and branches are splitting criteria. There is a maximal length for fragments which is not specified in the paper. For each fragment they collect the accessed variables, i.e. they compute sets of input and output variables. The declaration of variables is moved as far to the back as possible to reduce coupling. The recommended refactoring is the one which has the best benefit-cost ratio (that is their scoring function), where the benefit is the length of the extracted method and the costs are the numbers of needed input and output parameters. They evaluate their approach only on one open source system and declare a precision of up to 93%. To increase the precision, the maximal length could be ignored such that longer fragments can be extracted. The scoring function of the approach given in this work was influenced by the ideas of Yang et al. as the score of a candidate depends on structural information in the way that there are bonus points rewarded for blank lines or comments. But the most relevant factor for the score of a too long method is the reduction of complexity, i.e. of the length and nesting, where nesting is not considered by Yang et al. Whereas Yang et al. shift definitions as far back as possible before refactoring candidates are generated, this work considers only the source code as it is. They pointed out that the upper bound for the length of candidates prevented their approach to give recommendations that would have been preferred by the participants of their evaluation and this is why the approach presented in this work does not implement such an upper bound but other countermeasures are taken such that very long candidates are less attractive than shorter ones.

3.4. Prioritization of Extract Method Refactorings

Steidl and Eder focus in [SE14] the question of which findings should actually be resolved first. The question how to solve a given finding, i.e. a specific suggestion, is not addressed by their approach. They suggest a prioritization of quality defects that were found during a software quality analysis to maximize the developer's expected return of invest. That means that severe findings that can be easily removed should have a high prioritization. Exemplary, they suggest an approach for code clones and long methods. They implemented different heuristics that recommend the whole method for (manual) refactoring. To remove the long method finding, they use the following heuristics: The inline-comment heuristic recommends methods with a minimum number of inline comments. The extract-block heuristic tries to find an extractable block and recommends it. A similar heuristic is the extract-commented-block heuristic that recommends any commented block that begins with a `for`- or a `while`-loop, or an `if`-statement. The last heuristic, the method-split-heuristic, recommends methods that can be split into two methods with at most a certain number of parameters. In their evaluation they found out that the findings suggested by the inline-comment-heuristic are sometimes difficult to remove, the other heuristics seem to suggest methods that can be refactored easier. The authors argue that the inline-comment-heuristic is the only one that does not use any dataflow information and due to this not all suggested methods can be refactored easily. As a consequence of that, dataflow information should be taken into account to generate suggestions for extract method refactorings. Steidl and Eders' approach is not appropriate for automated refactorings as it only gives a hint, where a developer should start refactoring. It does not provide any information about which parts of the method can and should be refactored. The prototype of this work is implemented on the same system as Steidl and Eders' prototype. Both use the open source system ConQAT to perform the quality analysis and both use the same heuristic to get information about the dataflow in the considered methods. A fundamental difference is that this bachelor's thesis suggests only one heuristic but one that recommends specific candidates which can be used to perform refactorings on long methods instead of only recommending the method to be valuable for a refactoring.

4. Approach

The approach presented in this work recommends candidates for extract method refactorings. Usually, developers apply those refactorings on methods that are quite complex with respect to length or nesting depth. That is why the suggested approach focuses on methods that have too many lines of code (LoC) and suggests especially those candidates that reduce the code complexity by reducing length and nesting depth. The approach also considers the structural information, i.e. blank lines and comments, to derive suggestions that represent the original structure of the code. This chapter will describe in detail the approach of this work to suggest optimal candidates for the extract method refactoring automatically.

Figure 4.1 shows the process of suggesting extract method refactoring candidates from long method findings. To find the set of candidates that will be suggested by this approach, first all classes that are specified by the user are checked. Second, for all the too long methods in these classes a list is generated that contains all candidates which can be considered for an extract method refactoring. The candidates are ordered using a scoring function that depends on the criteria of length and nesting reduction, the number of input and return parameters and the number of blank lines and comments before and after a candidate. Finally, the best candidates are suggested for an extract method refactoring. This chapter will provide detailed information about the single steps of this approach. For illustrative purposes, a (shortened) method from the open source Java project AGILEFANT[1] (see listing 4.1) will be used.

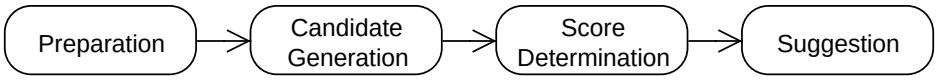


Figure 4.1.: Suggestion Process

[1] https://github.com/Agilefant/agilefant/

```
1  public void copyStory(Story otherStory, boolean moveFinishedTasks) {
2     this.setDescription(otherStory.getDescription());
3     this.setParent(otherStory.getParent());
4     if (otherStory.getParent() != null)
5        otherStory.getParent().getChildren().add(this);
6
7     // Copy the complex members: tasks, users, labels, parents
8     for (Task t : otherStory.getTasks()) {
9        if (moveFinishedTasks) {
10          if (t.getState() != TaskState.DONE && t.getState() !=
                TaskState.IMPLEMENTED) {
11             t.setStory(this);
12          }
13       } else {
14          t.setStory(this);
15          Task newTask = new Task(t);
16          t.setStory(otherStory); // set it back
17          this.getTasks().add(newTask);
18       }
19    }
20    this.getResponsibles().addAll(otherStory.getResponsibles());
21    for (StoryHourEntry entry : this.getHourEntries()) {
22       entry.setStory(this);
23       StoryHourEntry newEntry = new StoryHourEntry(entry);
24       this.getHourEntries().add(newEntry);
25       entry.setStory(otherStory);
26    }
27 }
```

Listing 4.1: Example Method

4.1. Preparation

The approach suggests to find extract method refactorings during a software quality analysis where long method findings are created. At the beginning of the analysis all specified files are read and their content is tokenized. The analysis uses the information contained in these tokens to construct CDFGs for each method in the source files. The CDFG construction was mainly developed by Fabian Streitel in his master's thesis [Str14]. For each class the set of global variables is stored as they are needed to compute the set of parameters that a refactoring candidate needs. For all long method findings, i.e. for all long methods that count more than 40 LoC (see subsection 2.2.2) extract method suggestions will be generated. The example method was cut by the author of this work to have a short and comprehensible example for the readers. Originally, the method counted 48 LoC, so it would be considered as too long.

4.2. Candidate Generation

For each method that is too long, a set of possible refactoring candidates is generated. To do so, the algorithm uses the compound statements that represent the statements in a method. Figure 4.2 illustrates the internal representation of the example method with compound statements. Each frame represents a (compound) statement that may have several child compound statements. The first compound statement represents the whole method and has several child statements. Some of these child statements (2, 3, 5, 11, 14 - 17, 20 and 22-25) represent only a single statement, where others (4, 8, 9, 10, 13 and 21) are compound statements. An extract method refactoring candidate now consists of a subtree of compound statements.

Preconditions

Algorithm 1 generates a list of candidates. To do so, continuous sub-sequences of compound statements are considered. Silva et al. refine the preconditions for the refactoring that were already mentioned in section 2.1 and point out, that not every possible sub-sequence is a candidate for extract method refactoring as three kinds of preconditions must hold: syntactical preconditions, behavior-preservation, and quality preconditions [STV14].

Syntactical Preconditions. The candidate generation algorithm guarantees that candidates attend the following three syntactical preconditions:

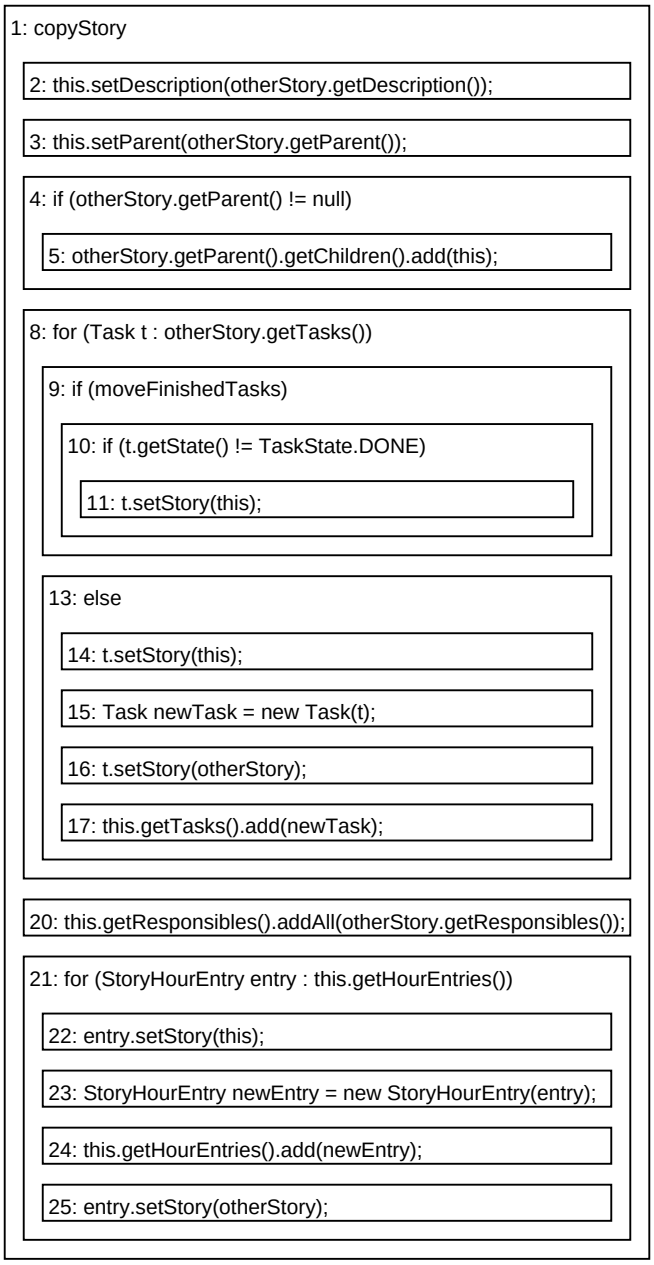


Figure 4.2.: Compound Statements representing the Example Method

Algorithm 1 Candidates Generation Algorithm

```
1: procedure getCandidates(Parent)                          ▷ Parent Statement
2:     Candidates ← ∅                          ▷ List that will contain the candidates
3:     Children ← ParentStmt.Children                  ▷ List of the child statements
4:     for k ← 0 to Children.size − 1 step k ← k + 1 do
5:         for j ← Children.size to 1 step j ← j − 1 do
6:             C ← ∅                                          ▷ Potential Candidate
7:             for i ← k to j − 1 step i ← i + 1 do
8:                 Statement ← Children.get(i)
9:                 Type ← Statement.Type
10:                if Type is if ∨ try then
11:                    C ← C + completeBlock(statement)
12:                else if (Type is catch ∨ finally) ∨ Type contains else then
                                                                   ▷ Do nothing
13:                else                                        ▷ Common Statement
14:                    C ← C + Statement
15:                end if
16:            end for
17:            if isLongEnough(C) ∧ isExtractable(C) then
18:                Candidates ← Candidates + C
19:            end if
20:        end for
21:        child ← Children.get(k)
22:        Candidates ← Candidates + getCandidates(child)
23:    end for
24:    return Candidates
25: end procedure
```

1. The highest level of compound statements that are selected as candidate have the same parent compound statement, which can be seen by the structure of the algorithm: It always considers a parent compound statement (see line 1) as starting point (at first the highest level compound statement that represents the whole method) with its children and after that for all its children the procedure is repeated (see line 22). As a consequence of this, it is not possible to get a refactoring candidate for the example method that only covers the statements in line 11 and 14 because they have different parents (10 and 13, respectively).

2. A candidate covers a continuous sequence of statements. The most inner `for` loop (line 7) ensures that always a continuous sequence of statements is considered as candidate. For the example method that means that there can not be a candidate which recommends the statements 14, 16 and 17 but not statement 15 as these are not a continuous sequence.

3. If a compound statement is part of a refactoring candidate, all its child statements are also part of the candidate, automatically. That is guaranteed by the structure of the compound statements as their children are always part of them. For example, if statement 8 of the example method is part of a candidate, all its children (9 and 13) are also part of the candidate as well as all grandchildren (10, 11 and 14 - 17).

Remark that an `if` and a corresponding `else` block are not represented by the same compound statement. The algorithm takes care of the fact that `if`-blocks can only be extracted if all depending `else if` and `else`-blocks are extracted. The same holds for `try-catch-finally`-blocks (see lines 10 and 12).

Behavior Preservation. The second precondition is behavior-preservation, that means that the behavior of the refactored method M must not change in any case after an extract method refactoring was applied. As [TC09] point out, `continue`, `break` or `return` statements violate behavior-preservation because they constitute unstructured control flow and would change the behavior if they were extracted. In addition, it is not possible to extract a candidate, for which more than one local variable is read (before rewriting it) after the candidate because then more than one return parameter would be needed. But in C-like languages it is not possible to have more than one return parameter. All these checks are performed by the `isExtractable` method in line 17.

Quality Preconditions. The third kind of preconditions that are mentioned by Silva et al. are quality preconditions. A refactoring candidate should have a minimal length and the remainder should also have at least that minimal length. [STV14] designed an

exploratory study that leads to the result that the minimal length should be set to 3 such that their refactoring recommendations have a maximal recall. With respect to the general procedure the approach by Silva et al. is similar to the approach presented in this work and thus the implementation of this approach also sets a minimal length of 3, for both, the candidate and the remainder, in the `isLongEnough` method in line 17.

4.3. Scoring Function

The next step after generating a list of candidates is ranking them. To rank the candidates, a scoring function determines a value for each candidate that depends on several kinds of criteria which are presented in figure 4.3. Each criterion results in some amount of points that will be summed up and result in the score of a candidate. The best candidate is the one with the highest score.

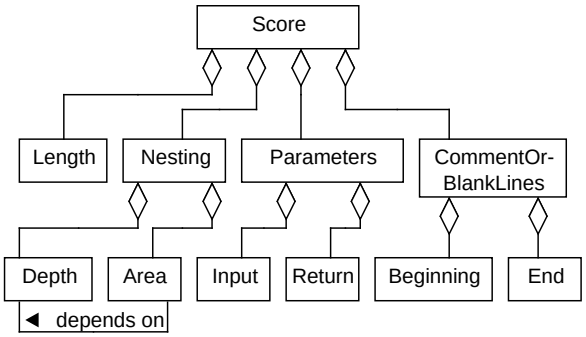


Figure 4.3.: Score Elements

4.3.1. Length

As the approach considers long methods with the aim of reducing their length and their complexity, the length of a refactoring candidate should influence its ranking. Nevertheless, the length of the candidate is not sufficient as a candidate that covers nearly the whole method (such that a method with the minimal length remains) would have the maximal length and a first implementation might rank the longest candidate always at the top of the candidate list. To avoid the effect of recommending nearly the whole method, the length score S_{length} depends on the length of the candidate L_c and the length of the remainder L_r. For each line a constant amount of points c_l is

rewarded, up to the upper bound $MAX_{scoreLength}$. This upper bound ensures that very long candidates (that might lead to another too long method) are not ranked higher just because they are extraordinary long. In contrast to [YLN09] there is no upper bound such that very long candidates are not considered for recommendation, but these are not extra rewarded for being very long. The approach sets

$$S_{length} = \min\left(c_l \cdot \min\left(L_c, L_r\right), MAX_{scoreLength}\right),$$

where in the implementation of the approach $c_l = 0.1$ and $MAX_{scoreLength}$ is set to 3, i.e. the maximal length score is achieved by 30 or more lines of code.

4.3.2. Nesting Depth

This approach aims to reduce complexity of long methods and [Hum] states that nesting depth is an important indicator of code complexity. That is why reduction of nesting depth by a value R is considered for the ranking of the refactoring candidates. Let D_m be the nesting depth of the original method, D_r be the nesting depth of the remainder, and D_c be the nesting depth of the refactoring candidate. The approach sets the score of a candidate obtained for reducing the nesting depth to

$$S_{nestDepth} = R = \min\left(D_m - D_r, D_m - D_c\right),$$

which means that (theoretically) there is no upper bound for $S_{nestDepth}$, but remark that given a method with a nesting depth D_m the maximal reduction of nesting depth is $\left\lfloor \frac{D_m}{2} \right\rfloor$ and so, $S_{nestDepth} \leq \left\lfloor \frac{D_m}{2} \right\rfloor$ always holds. The example method in listing 4.1 has a nesting depth of 3. By applying an extract method refactoring one can achieve a reduction of nesting depth of at most 1, and to reduce nesting depth at all, line 10 must be extracted without extracting the whole `for` loop as the maximal nesting depth of the remainder and the extracted method would then be 3, so no reduction would have taken place. Even though, it might make sense to extract the whole `for` loop. So, reducing the nesting depth should influence the score but there is another criterion with respect to code nesting that can be always used to rank refactoring candidates, that is the reduction of nesting area.

4.3.3. Nesting Area

Nesting area is, intuitively spoken, the area under the single statements of pretty printed code. As shown in the previous section, nesting depth is not always a suitable criterion to determine reduction of complexity as it might be complicated or even impossible to extract all deeply nested statements to reduce the maximal nesting

depth of the remainder and the candidate. But even if nesting depth is not reduced, reduction of code complexity is possible by extracting nested statements. That is why this approach also considers the reduction of nesting area. If nesting is reduced by a candidate more than by another one, the first one should be ranked higher. The same is true if a candidate extracts statements that are nested deeper than the statements covered by another candidate. A candidate is optimal with respect to nesting area reduction if the nesting area reduction is maximized, or in other words the maximal nesting area of the remainder (A_r) and the candidate (A_c) is minimized: $A_{reduction} = \min(A_m - A_c, A_m - A_r)$, where A_m is the nesting area of the original method. The higher $A_{reduction}$ is the better the candidate. For a given method with nesting area A_m an optimal candidate can achieve a reduction $A_{reduction}$ of at most $\left\lfloor \frac{A_m}{2} \right\rfloor$, similar to the maximal nesting depth reduction.

This approach assumes that reducing the nesting area becomes more and more important the higher the nesting depth of the original method D_m is and that is why the upper bound of the score achievable for reducing the nesting area depends on D_m:

$$S_{nestArea} = 2 \cdot D_m \cdot \frac{A_{reduction}}{A_m}$$

The factor 2 is taken into account to obtain a score for the nesting area that is at most as high as D_m. As reduction of nesting area is nearly always possible, the achievable score for nesting area reduction is higher than the achievable score for reducing nesting depth (which is only in some cases possible). If nesting depth (i.e. complexity) is high, the other criteria have less relevance for the scoring of the candidates because nesting scores are not bounded while the other scoring criteria are bounded.

For the example method we have seen that $D_m = 3$. The nesting area of the original method is $A_m = 20$. If we consider the candidate that covers the whole first `for` loop (i.e. the statements 8 - 17), the nesting depth reduction would be 0 as $D_m = D_c$. But the nesting area of the candidate $A_c = 15$, the nesting area for the remainder $A_r = 5$ and therefore, for this candidate $S_{nestArea} = 2 \cdot 3 \cdot \frac{5}{20} = \frac{3}{2}$.

4.3.4. Parameters

To obtain the most independent candidates with respect to coupling, the approach considers the number of parameters that are needed for each candidate. The more parameters are needed to extract the candidate from the original method, the higher is the data dependency between the original and the extracted method. That is why [YLN09] set the costs of a candidate to the number of parameters it needs. Similarly, this approach assumes that the more parameters are needed the lower the score of the candidate. For the parameter score S_{param} there is an upper bound $MAX_{scoreParam}$.

Usually, the number of needed parameters (n_{input} and n_{output}, where $n_{output} \leq 1$) will reduce the score, each parameter decreases the score by one:

$$S_{param} = MAX_{scoreParam} - n_{input} - n_{output}$$

Fowler claims in [Fow99] that having a long parameter list is another bad smell. He proposes to not have more than three input parameters. As we have in C-like languages at most one return parameter, this approach sets $MAX_{scoreParam} = 4$. That means that a candidate that needs three input parameters and one output parameter will get no points for its parameters.

The approach tries to avoid introducing new smells into the code by suggesting refactorings that smell. That is why candidates with at least $MAX_{scoreParam}$ many input parameters will receive a parameter score S_{param} of -10,000. This value does not mean anything else than that this candidate is quite bad and too many parameters can not be excused by being very good in any other criterion. There is one exception from this rule: If the original method already had too many input parameters[2] the score will be calculated using the following formula:

$$S_{param} = MAX_{scoreParam} \cdot \frac{\left(n_{methodInput} - n_{input} - n_{output}\right)}{n_{methodInput}},$$

with $n_{methodInput}$ being the number of input parameters of the original method.

4.3.5. Comments and Blank Lines

As [SE14] and [YLN09] point out, the visual structure of code, i.e. blank lines and comments, is also an important indicator for refactoring candidates. Developers often have comments that give information about the next source code line(s), especially if these perform something else than the previous ones. But this is a violation of the design principle that one method should process only one thing (see [Mar09]) and therefore, the following lines might be a good candidate for an extract method refactoring. Hence, this approach suggests a scoring criterion that considers blank lines and comments at the beginning and at the end of a candidate. For each such line (and the fact that these lines exist) c_p many points are rewarded. The approach assumes that blank lines and comments at the beginning of a candidate are more relevant to identify the most suitable extract method refactoring candidate than the ones at the end of a candidate because they give more information about the candidate itself. In

[2] Sometimes more than three parameters are needed or useful, e.g. a C++ method from a Windows library that creates a pop up and needs twelve parameters:
`http://msdn.microsoft.com/en-us/library/windows/desktop/ms632680%28v=vs.85%29.aspx`

the score formula the higher relevance is represented by the factor f_b, which should be greater than 1. In addition, several lines of comments before a sequence of statements may indicate a more complex explanation which is more likely to describe a new functionality and therefore, more blank or commented lines should get more points.

The score depends on four variables: The existence of blank lines or comments first, at the beginning (e_b) and second, at the end (e_e) of a refactoring candidate. Third, the number of blank lines or comments at the beginning (n_b) and fourth, at the end of a candidate (n_e), where $e_x \in \{0,1\}, n_x \in \{0,1,2,3\}$ and $x \in \{b,e\}$. Clearly, $e_x = 0 \Rightarrow n_x = 0$ and $e_x \neq 0 \Rightarrow n_x \neq 0$. If there are more than three blank lines or comments the same amount of points is rewarded as if there were three blank lines or comments. This approach suggests that

$$S_{commentsBlankLines} = f_b \cdot c_p \cdot (e_b + n_b) + c_p \cdot (e_e + n_e)$$

For the prototype, it was assumed that comments at the beginning should result in twice as many points as comments at the end, i.e. $f_b = 2$. c_p was set to 0.25 s.t. a candidate may get up to 2 points for having at least three comments or blank lines at the beginning and up to 1 point for having at least three comments or blank lines at the end.

4.3.6. Scoring Elements Intervals

The previous subsections gave detailed information about the single criteria for the score of a refactoring candidate. Table 4.1 shows the intervals of the single scoring elements. D_m stands for the nesting depth of the original method.

Score Element	**Min**	**Max**
S_{length}	0	3
$S_{nestDepth}$	0	$\left\lfloor \frac{D_m}{2} \right\rfloor$
$S_{nestArea}$	0	D_m
S_{param}	-10,000	4
$S_{commentsBlankLines}$	0	3

Table 4.1.: Scoring Elements and their Intervals

As already mentioned, for methods with a high nesting depth, the elements that are not considering code nesting are less relevant for the total score to obtain candidates

that reduce complexity. For methods that do not have such a high nesting depth, the other criteria are quite relevant.

Remember that the minimum for S_{param} of -10,000 is only achieved by methods that have too many input parameters. If the number of parameters is not too big, the minimum is 0. Section 5.3 provides further details on the question, when a candidate with too many input parameters may be suggested for an extract method refactoring by the prototype of this approach.

4.3.7. Total Score

The candidates will be compared using the total score S. For each candidate the total score is the sum of all single scoring elements:

$$S = S_{length} + S_{nestDepth} + S_{nestArea} + S_{param} + S_{commentsBlankLines}$$

That means, that the total score of a valid candidate is between -10,000 and $D_m + \left\lfloor \frac{D_m}{2} \right\rfloor + 10$.

After having calculated the score for each candidate, the list of candidates is sorted (best candidate first).

Please note that, although the implementation of this approach is evaluated at the end of this work, the scoring function itself and the importance of their single parts to the score should be investigated during another research.

4.4. Suggestion

After all candidates where ranked using the described scoring function, the best candidates are suggested for an extract method refactoring.

The implementation details (chapter 5) will explain how many candidates are suggested by the prototype. The evaluation (chapter 6) also addresses the question, if more than one suggestion is sensible.

5. Implementation Details

This chapter will provide some more detailed information about the prototype that implements the approach, that address some extreme cases. These details are not relevant to understand the idea of the approach but are needed to get satisfying results of the analysis and appropriate suggestions.

5.1. Analysis Parameters

The prototype allows some additional parameters to those that are described in chapter 4. The user may specify a length threshold for long methods (which is by default 40).

But there are two other parameters that were not mentioned in chapter 4. As default, at least three candidates are recommended. There is a parameter that specifies the least amount of candidates which should be recommended. The question if several candidates should actually be recommended will be discussed in chapter 6.

There is no absolute value for the maximal number of candidates that should be suggested, but via a parameter the number of candidates per line of code can be specified to get more suggestions for very long methods than for shorter ones. The default value for that parameter is $\frac{1}{15}$, which means that for each 15 lines of code an additional candidate is recommended. If there should always be a fixed number of candidates suggested, one should set the parameter that specifies the minimal number of suggested candidates to that fixed number and the parameter for the number of candidates per line to 0.

5.2. Generated Number of Candidates

The maximal number of candidates that is generated by Algorithm 1 depends on n, the lines of code in the input method. The algorithm generates at most n^2 candidates (see lines 4 and 5 of Algorithm 1) and therefore considers up to $\frac{n*(n+1)*(n+2)}{6}$ many (compound) statements. As there are $\mathcal{O}(n^2)$ many candidates, even for not very big n the number of candidates is quite big. The worst case of considered statements only appears if there are only sequential statements with no branching and no comments or

blank lines etc. For example, for a method with 100 sequential statements (i.e. $n = 100$) the number of candidates is at most 10,000 which cover a total of 171,700 statements. For $n = 500$ (which appears from time to time) the number of candidates can be up to 250,000 (which cover a total of 20,958,500 statements). In real-world systems there are some methods that have quite a lot of lines of code without any branching, for example to set up the graphical user interface (GUI) or to call initialization methods.

For methods, that have such a low nesting, often quite big fragments can be extracted with just some input parameters, i.e. it does not make sense to generate all possible candidates but only the longest ones to have a fair trade-off between performance and the number of candidates. That is why the implementation of this approach has an upper bound for the number of candidates that are generated by algorithm 1:

$$MAX_{candidates} = 10,000 = n^2 \text{ for } n = 100$$

That means that for all methods that are not longer than 100 LoC all possible candidates are generated.

As the algorithm starts with generating the longest candidates and then reduces the length of the next candidates, the first $MAX_{candidates}$ are always the longest ones.

5.3. Negative Scored Candidates

Chapter 4 points out that the presented approach tries to avoid introducing new smells into the code by suggesting refactorings that smell. That is why candidates with at least $MAX_{scoreParam}$ many input parameters receive a parameter score S_{param} of -10,000, which will lead to a negative total score S. Before candidates are recommended, there is a check if there is at least one candidate with a positive total score S. If there is, all candidates with a negative total score will be deleted from the list of refactoring candidates and this is why it can happen that not enough candidates are recommended (with respect to the parameters mentioned for the number of suggested candidates), even though there were enough valid candidates.

5.4. Embracing Candidates

A candidate is embraced by another candidate if all its statements are also covered by the other candidate. If a candidate is embraced by another candidate, which has the same set of input and return parameters and a better score, the embraced candidate will not be suggested. This means for the prototype of the presented approach that two candidates, which differ only in some lines and have the same set of input and

return parameters and the shorter one has a smaller total score S than the longer one, the shorter one will be deleted (and therefore surely not suggested).

That feature was implemented to get a wide range of really different suggestions as small adaptions on suggested candidates can be easily done by the developer who wants to apply an extract method refactoring.

5.5. Visualization

The prototype of the approach shows suggestions for extract method refactorings in a code view. After all candidates were ranked, for the best candidates markers are generated. The number of recommended candidates (and therefore markers) depends on parameters of the ConQAT analysis (for details see section 5.1). The markers will be displayed as vertical lines on the left of the covered statements in the result file of the analysis. The best candidate has a light green marker, the last recommended candidate has a black marker. The candidates between those have green markers where the brightness of the color of the marker decreases.

Figure 5.1 shows three recommended candidates for the example method. The best candidate (determined by the scoring function explained above) covers the first `for` loop and the statement after it (in figure 5.1, ll. 65-78). The second candidate also covers the second `for` loop and has a lower score because the nesting area reduction is less. The third highlighted candidate suggests to extract the first statements including the first `if` statement. The orange bar highlights the whole method and means that it is too long. Please note that the example method usually would not be considered as too long, suggestions were only generated as this method was the example throughout this and the previous chapter.

The user gets more information about a candidate when he hovers the mouse over its marker. The prototype displays the ranking, input and output parameters, length, and scoring details. What other kind of information should be displayed will be evaluated in chapter 6.

5.6. Restrictions

Although the general approach suits all C-like programming languages, the prototype is restricted to Java. It can be easily adapted to other C-like languages by specifying suitable constants for the scoring function.

Due to the implementation of the CDFGs the prototype is not able to generate valid extract method refactoring candidates for methods that have methods in (anonymous) inner classes. This is because they have read access to the variables of the outer method,

```
52:     public void copyStory(Story otherStory, boolean moveFinishedTasks) {
53:         this.setDescription(otherStory.getDescription());
54:         this.setStoryValue(otherStory.getStoryValue());
55:         this.setName(otherStory.getName());
56:         this.setBacklog(otherStory.getBacklog());
57:         this.setTreeRank(otherStory.getTreeRank() - 1);
58:         this.setState(otherStory.getState());
59:         this.setStoryPoints(otherStory.getStoryPoints());
60:         this.setParent(otherStory.getParent());
61:         if (otherStory.getParent() != null)
62:             otherStory.getParent().getChildren().add(this);
63:
64:         // Copy the complex members: tasks, users, labels, parents
65:         for (Task t : otherStory.getTasks()) {
66:             if (moveFinishedTasks) {
67:                 if (t.getState() != TaskState.DONE
68:                         && t.getState() != TaskState.IMPLEMENTED) {
69:                     t.setStory(this);
70:                 }
71:             } else {
72:                 t.setStory(this);
73:                 Task newTask = new Task(t);
74:                 t.setStory(otherStory); // set it back
75:                 this.getTasks().add(newTask);
76:             }
77:         }
78:         this.getResponsibles().addAll(otherStory.getResponsibles());
79:         for (StoryHourEntry entry : this.getHourEntries()) {
80:             entry.setStory(this);
81:             StoryHourEntry newEntry = new StoryHourEntry(entry);
82:             this.getHourEntries().add(newEntry);
83:             entry.setStory(otherStory);
84:         }
85:     }
```

Figure 5.1.: Marked Recommendations for the Example Method

but their CDFGs stores no (dataflow) information about that. Therefore, the prototype may determine not all parameters that would actually be needed for a refactoring, especially more return parameters could be necessary. The aim of the prototype is to suggest only candidates that are valid (with respect to the preconditions in section 4.2). That is why in the case of inner classes no candidates are generated at all. Chapter 6 shows for seven study objects how many long methods are concerned by this restriction.

When the implementation of CDFGs is adapted, the approach is implemented in such a way that suggestions also work for methods within inner (anonymous) classes.

6. Evaluation

In this chapter, the usefulness and restrictions of this approach will be evaluated using the prototype which is implemented as a ConQAT analysis for Java projects. As the approach aims to support developers and software quality analysts (SQAs), the evaluation focuses on the usefulness of the approach from their perspective. To evaluate the approach, an online survey was constructed and ten developers and SQAs participated in that survey. This chapter is structured as follows: At first, the research questions are explained, second, an overview about the study objects is given. Third, the study setup including an online survey is presented. After these introductory steps, the results of the evaluation will be shown and discussed. Finally, threats to validity of this evaluation are mentioned.

6.1. Research Questions

RQ1: How often do too long methods with inner classes appear? In section 5.6 it was mentioned that the prototype of this approach is currently restricted to long methods not containing inner classes. This research question tries to figure out, how many too long methods are actually affected by this restriction.

RQ2: Are suggestions better than a random (valid) refactoring candidate? This question considers a first criterion to have a useful scoring function. If random candidates are not significantly less preferred by developers and SQAs than the suggestions of this approach, the approach with its scoring function would not be useful.

RQ3: Do developers follow the suggestions of this approach? This question considers a much stronger criterion of usefulness than RQ2. The evaluation of this work tries to find out, whether (and how often) this approach is able to suggest candidates that are taken as refactoring candidate from developers and SQAs. The more often developers follow the suggestions of the prototype, the closer is the scoring function on the intuition of the developers.

RQ4: Should several suggestions be made? This questions addresses a result of an evaluation in [STV14]. They claimed that an implementation of their approach should preferably suggest only the best candidate. As the approach of this work is structurally

similar to their approach, this work tries to find out whether their result also holds for the prototype of this work.

RQ5: Which information is relevant for developers to rate a refactoring candidate? To make a tool that implements this approach as useful as possible, this evaluation tries to figure out which criteria of refactoring candidates, the original method, and remainders are the most important ones for developers and SQAs to identify the most appropriate refactoring candidate. These criteria should then be displayed by tools that suggest extract method refactorings for each candidate.

6.2. Study Objects

Table 6.1 shows the study objects used to answer the research questions. All seven study objects are Java open source projects. They have a size of approx. 10,000 to 184,000 LoC and have different domains. They all have too long methods but some have – in relative terms – more long methods than others. Atunes and Agilefant for example have only 1.1% of too long methods, JabRef counts 428 too long methods, i.e. 7.6% and Mediocre Chess has 29.2% of too long methods. The longest method (with respect to lines of code) can be found in JabRef and counts 1,305 lines. All projects were selected for the evaluation because they are well-known Java open source systems and have a five star ranking (based on voluntary feedback from the users) on the open source distribution platform sourceforge[1]. To answer RQ1, all these projects were considered. For the survey, which aims to answer RQ2-RQ5, only too long methods from Agilefant, JabRef and JChart2D were selected to have several examples from each project with the trade-off between time needed to answer the survey and the number of examples that were presented during the survey.

6.3. Study Setup

To answer RQ1, for all seven study objects the number of too long methods that have several CDFGs, i.e. that have an inner class, are set in relation to the total number of too long methods. Remember that a method is considered as too long if it counts more than 40 LoC.

An online survey was constructed to answer the other research questions, RQ2-RQ5. For the survey, all participants received an html-file that contained ten methods (they will be called survey objects) that were considered during the survey. All these methods had between 48 and 73 LoC. For each survey object there were three candidates that

[1] `http://sourceforge.net/`

Name	Domain	Size (LoC)	# Methods	# Long Methods (%)	Longest Method (LoC)
Agilefant	Backlog Tool	36,116	2,841	31 (1.09%)	143
Atunes	Mediaplayer	184,305	9,823	105 (1.06%)	138
Freemind	Mindmapping Tool	74,421	4,356	133 (3.05%)	871
JabRef	Reference Manager	128,145	5,665	428 (7.56%)	1,305
JChart2D	Charting Library	50,728	1,849	72 (3.89%)	641
LogiSim	Circuit Editor	80,540	6,197	160 (2.58%)	170
Mediocre Chess	Chess Engine	9,666	154	45 (29.22%)	500

Table 6.1.: Study Objects

were all highlighted using a light green bar beside the code lines of the corresponding candidate. One of the candidates was always the first suggestion of the prototype, which will be called *TOP1*. Another one was the second or third suggestion of the prototype (which one it actually was, was determined randomly during the analysis but then was the same for each participant), called *TOP2/3*. The third candidate was a randomly selected valid candidate that was not considered as being one of the TOP3 candidates of the scoring function, called *Random*. The order of the highlighting bars was only determined by the start line of the candidate, not by their order of the scoring function. The displayed information when hovering with the mouse over a marker contained only the input and return parameters, the length of the candidate, and the reduction of nesting depth, not the ranking of the candidate.

Survey Questions

During the survey, for all survey objects the following survey questions were asked:

SQ1: Which candidate would you use more likely for an extract method refactoring? The participants were asked to "[c]onsider the method evaluationXY in the class EvaluationXY". To do so, they should "[t]ake a look at the three candidates that are highlighted with a green bar." They always had the options "Candidate 1", "Candidate 2" or "Candidate 3" with the corresponding start and end lines.

SQ2: Would you use the selected candidate for an extract method refactoring? The participants were asked to "assume that [they] want to refactor the given method". They could answer "Yes", "Yes, but slightly shifted (by 1-2 lines)" or "No".

SQ3: Would you have applied an extract method refactoring on this method? For the third question the participants should "[d]ecide whether it is worth to shorten the method by

applying a refactoring (i.e. using an extract method refactoring tool)". There were two options: "Yes" or "No".

To answer RQ5, at the end of the survey the participants were asked to answer the following survey question:

SQ4: How important is it to display the following criteria for each candidate (e.g. in a small text box) to be able to decide which candidate is the most preferable one? The importance could be characterized using integer values between 1 (very unimportant) and 5 (very important). There were the following criteria:

- Ranking (determined by a candidate suggestion tool)
- Input parameters of the candidate (or the information that none is needed)
- Return parameters of the candidate (or the information that none is needed)
- Length of the candidate, the original method, and the remainder (each individually rated)
- Nesting depth of the candidate, the original method, and the remainder (each individually rated)
- Reduction of the maximal nesting depth (after the refactoring)
- Reduction of the length (absolute value)
- Reduction of the length (relative value)

SQ1 will be used to answer RQ2, SQ2 and SQ3 will be needed to answer and discuss RQ3 and RQ4. The last survey question, SQ4, forms the basis to answer RQ5.

6.4. Results

The following subsections summarize the results of the research questions. Two developers and eight SQAs participated in the survey that is used to answer RQ2-RQ5. All of them were experienced developers as they have between 6 and 24 (in average 12) years of development experience. All SQAs are using software quality tools frequently, where both developers are not using such tools regularly. All participants (except one developer) are regularly applying extract method refactorings with help of a tool (that might be integrated in their favorite IDE). The developer that does not use tools for extract method refactorings applies those manually as there is no tool support for refactorings for the programming languages he uses. All the participants that use tools to apply extract method refactorings have to select the code that should be extracted manually.

How often do too long methods with inner classes appear? (RQ1)

To answer RQ1, the total number of too long methods was determined and the fraction of methods that had only one CDFG, i.e. that had no inner classes, was calculated. For all these methods, the prototype was able to generate extract method refactoring suggestions, if there were any extractable candidates. In Table 6.2 one can see that for 52.4% to 97.8% of the too long methods from the study objects suggestions were generated. That means that for less than $\frac{1}{2}$ of the too long methods, no suggestions are available due to the existence of several CDFGs. For all but one study object for more than $\frac{2}{3}$ of the too long methods suggestions were generated if there were extractable candidates.

Name	Domain	# Long Methods	Only one CDFG (%)
Agilefant	Backlog Tool	31	29 (95.55%)
Atunes	Mediaplayer	105	55 (52.38%)
Freemind	Mindmapping Tool	133	93 (69.25%)
JabRef	Reference Manager	428	294 (68.96%)
JChart2D	Charting Library	72	49 (68.06%)
LogiSim	Circuit Editor	160	156 (97.50%)
Mediocre Chess	Chess Engine	45	44 (97.78%)

Table 6.2.: Available Suggestions

Are suggestions better than a random (valid) refactoring candidate? (RQ2)

The diagram in Figure 6.1 shows the results for SQ1 after the mapping from the candidates number (1, 2, 3) to their actual position in the ranking of the scoring function (i.e. *TOP1*, *TOP2/3* and *Random*). 74% of the selected candidates were the one that was ranked top most by the scoring function. The other 26% were the second suggestion, i.e. the TOP2/3 candidate.

For all ten survey objects all participants selected (under the assumption that they want to refactor the given method) either the TOP1 or the TOP2/3 candidate. The random candidate was never selected by any of the participants and therefore one can assume that for the ten survey objects the recommendations generated by the prototype are much better than the selection of a valid random candidate.

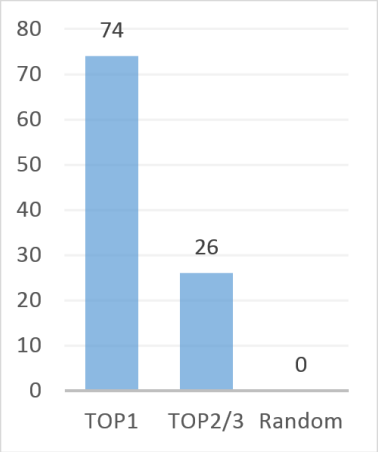


Figure 6.1.: Results of SQ1

Do developers follow the suggestions of this approach? (RQ3)

The diagram in Figure 6.2 shows the results of SQ2. Again, 74% would apply the selected refactoring candidate (i.e. one of the suggestions of this approach) without modifications, 12% would apply a quite similar refactoring by only shifting the selected candidate about one or two lines. Only 14% claim that they would not have applied the selected refactoring, even if they assumed that they want to refactor the given method.

The results of SQ3 are presented in the diagram in Figure 6.3. One can clearly see that for most of the cases (93%) developers and SQAs would apply an extract method refactoring on the presented method, i.e. they think that it is worth to spend some time on a refactoring. Five of the seven "No" answers concerned the last survey object, which was a method that contained a test case.

Should several suggestions be made? (RQ4)

As seen in the diagram of Figure 6.1, in 74% the selected candidate was the best one with respect to the order determined by the scoring function of this approach. The other 26% were the TOP2/3 candidate. These values of course do only represent the average distribution. It is remarkable that for none of the survey objects a similar distribution appeared: For half of them nearly all participants (nine out of ten) selected the TOP1 candidate and for the other half of survey objects the distribution was quite mixed, i.e. five participants selected TOP1 and the other five selected TOP2/3 or six selected TOP1 and four TOP2/3 (or vice versa).

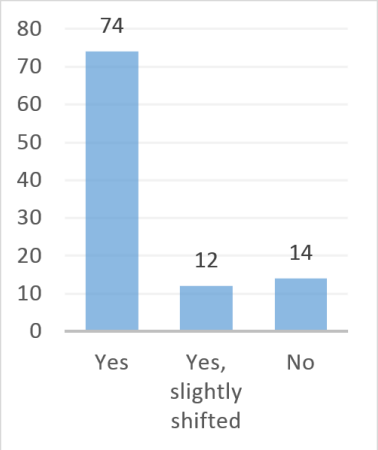


Figure 6.2.: Results of SQ2

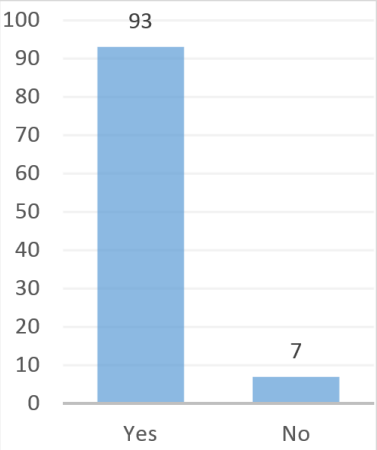


Figure 6.3.: Results of SQ3

Which information is relevant for developers to rate a refactoring candidate? (RQ5)

The diagram in Figure 6.4 shows the results of SQ4. In the diagram, 1 means very unimportant and 5 means very important. It shows the variance of the answers using arrows for the max and min values.

There is a high agreement on the importance of the input and return parameters (in average they are rated with a 4.8 or 4.9, respectively). According to the survey answers, the length of the original method is rather unimportant.

The other criteria are not very clear because there is often a big variance between the single answers. For example, the nesting depth of the original method has an average score of 2.9 and for this criterion, all possible answers were selected by the participants, i.e. for some it is very important, for others very unimportant. The intuition of the importance of the ranking varies a lot, too. That means that the results concerning criteria like the nesting depth of the original method or the ranking are inconclusive.

6.5. Discussion

This section discusses the results of the analysis of the survey objects and the survey itself. Many participants gave additional and individual feedback and reasons for their answers which will also be included in the discussions.

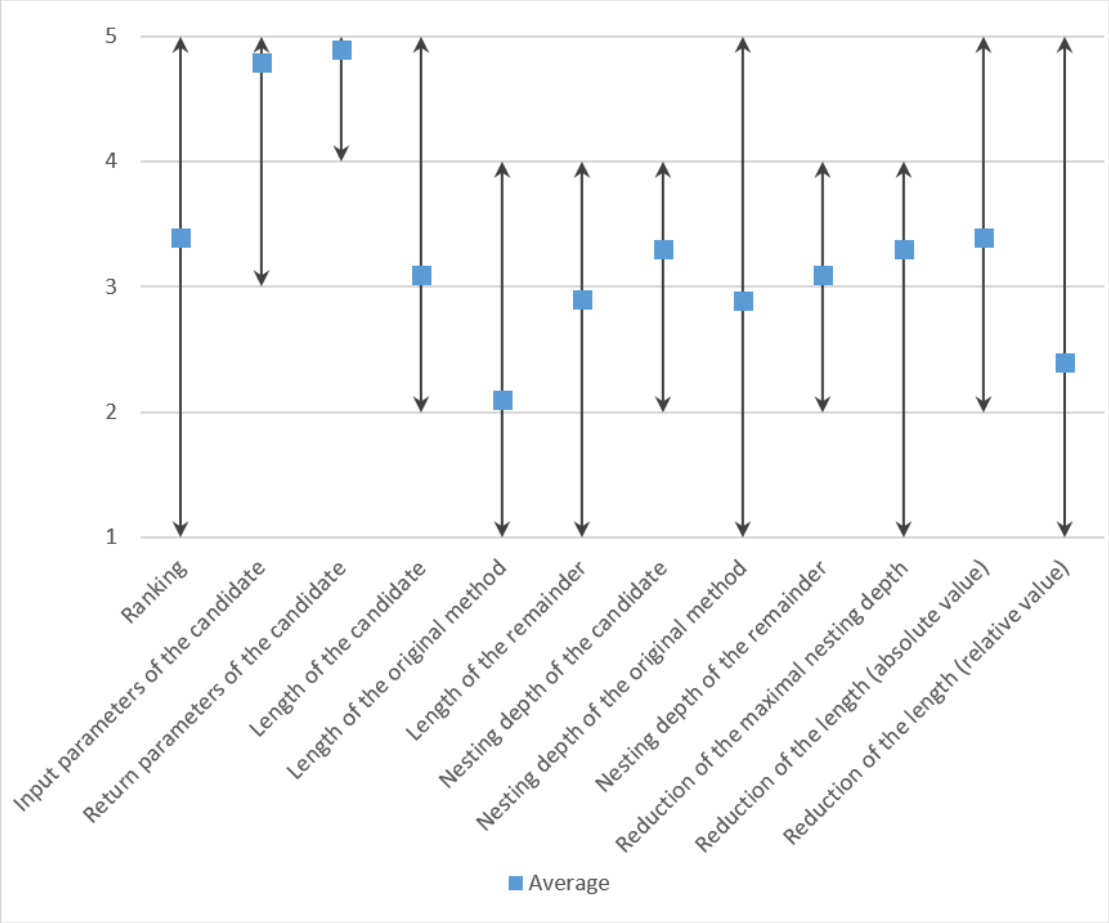


Figure 6.4.: Results of SQ4

How often do too long methods with inner classes appear? (RQ1)

There is one software project, where for nearly every second too long method no suggestions can be generated due to the existence of inner classes. Even though, for most of the study objects, the prototype was able to generate refactoring suggestions for more than $\frac{2}{3}$ of too long methods. That means that for most of the study objects suggestions for a considerable amount of too long methods were generated.

In addition, inner classes often appear in methods that set up the GUI or are used for initialization purposes, which are therefore usually not the aim of quality improvement sessions. In Atunes, for example, there are many too long methods that create buttons using inner classes or setup multi-threading by instantiating threats using inner classes.

As mentioned in section 5.6, the prototype is able to generate suggestions for methods with inner classes when the implementation of CDFGs is adapted. Then of course, for all too long methods that have extractable candidates, the prototype is able to generate suggestions for extract method refactorings.

Do developers follow the suggestions of this approach? (RQ3)

For the survey objects, 86% of the selected candidates (maybe slightly shifted) would have been chosen for an extract method refactoring. All of them were suggestions of the prototype.

Several participants claimed (in their individual feedback) that for some survey objects there were redundancies in the code such that they would first try to eliminate the redundancies and then they would refactor the resulting method. But as they would not start with an extract method refactoring, they answered in such cases with "No" to SQ3. As already mentioned, half of participants would not have refactored the last survey object because that method covered a test case. Thus, the participants claimed that it was better to keep the whole test case in the same method to have a better overview about the functionality that is tested by the given test case.

That means that not all methods that are considered as too long (and therefore complex) by the approach are candidates for developers and SQAs for extract method refactorings (at least they would not start with an extract method refactoring to increase code quality). In general, the suggestions seem to be helpful and if developers want to refactor a given method using an extract method refactoring, they often follow the suggestions of the prototype.

Should several suggestions be made? (RQ4)

Many participants mentioned in their feedback that there were some methods where they were quite sure which candidate is the best one and that they would apply only

this one extract method refactoring on the given method. For other survey objects, they would have applied both suggestions, the TOP1 and the TOP2/3 candidate for extract method refactorings. So, to answer the question in the survey, where they could select only one option, they had to select their answer more or less randomly between those two candidates. That might be an explanation for the mixed answers.

In practice, of course, several refactorings can be applied and it often is the best solution to refactor a too long method by extracting several pieces of code into new methods. Hence, in some cases it really makes sense to suggest several candidates, at least the TOP3 candidates with respect to the ordering of the scoring function. In chapter 7, a suggestion is made that addresses the problem of recommending several candidates, that should all be applied as extract method refactoring.

Which information is relevant for developers to rate a refactoring candidate? (RQ5)

The answers to SQ4 were not that clear. From the results presented in the diagram in Figure 6.4 one can see that the importance of the ranking of candidates is quite unclear. A reason for that might be that participants did not know whether the ranking is meaningful and further experience would have been needed to be gathered to see if the ranking is a useful information. The participants themselves had no experience with the tool and as mentioned before, the suggestions were made anonymous for the evaluation as there was a random candidate involved. That means that during the survey, no ranking information was given.

The length of the original method was considered as rather unimportant by the participants. One of them pointed out that it is not necessary to give this information for each single candidate. On the other hand, the information that a method is actually too long might be relevant. In software quality analysis tools, one can rely on the highlighting of long method findings.

Finally, the results of SQ4 imply that at least the information about input and return parameters are highly relevant for developers to identify the most appropriate candidate for an extract method refactoring.

6.6. Threats to Validity

As usual, there are some threats to validity of the evaluation of this work, that are summarized in the following.

Resolution of too long methods is subjective (see [YLN09]). First, there is no consense in science when a method is actually too long. This means that some may treat a given method as too long where others do not. Second, there is no commonly accepted

algorithm that splits a too long method into suitable smaller ones. That actually is a threat to validity of this evaluation as several participants were asked which candidate according to their opinion is best for an extract method refactoring. Other participants might have selected other candidates. To handle this risk, ten experienced developers and software quality analysts, that have deep knowledge about findings and their resolution, took part in the survey.

A threat of external validity is, as usual in software engineering topics, that the results of the evaluation need not necessarily hold for other software systems. The evaluation gave an impression of the severeness of the restrictions mentioned in section 5.6 for seven Java open source systems. The survey objects were taken from three of these systems. It can not be assured that the results of this work are valid for other software systems.

Of course, the ten survey objects do not represent the whole spectrum of methods that should be refactored using an extract method refactoring. They all covered (for too long methods) only a few lines of code and did not represent all possible ways of designing a method. To have a fair overview the survey objects were selected from several Java open source systems and there from different packages. They have quite different code structures such that a wide range of ways how methods can be structured are covered by the evaluation.

Although the approach fits for C-like languages, the evaluation of this work considers only Java open source systems because the prototype of this work is currently restricted to Java. From the results of this evaluation, it is not clear whether an implementation of the approach for other C-like languages will lead to similar results.

7. Conclusion

The last chapter of this bachelor's thesis provides a short summary about the approach, the prototype of the approach and the most important answers to the research questions that were addressed by the evaluation of this thesis. Finally, some suggestions for future work are presented.

7.1. Summary

Approach. This work presented an approach to derive extract method refactoring suggestions for C-like languages from software quality analysis findings, especially long and nested methods, to reduce code complexity. The approach determines valid extractable candidates from the CDFG of a method. A refactoring candidate needs to fulfill several kinds of preconditions: Syntactical preconditions, behavior-preservation, and quality preconditions. Each candidate is ranked using a scoring function that considers the following criteria: Length reduction, nesting depth and area reduction, number of input and return parameters. For methods that have a high nesting depth the reduction of code nesting is most important.

Prototype. A prototype implementing the approach presented in this work can be easily adapted to change the parameters that are suggested in this work. For performance reasons, for very long and low-nested methods, only the most important, i.e. the longer candidates are generated. The prototype takes care of embracing candidates: If a candidate is embraced by another candidate, that has the same set of parameters and a better score, it will not be suggested. The best candidates are suggested in a code view by highlighting the candidates with markers beside the corresponding code lines. Depending on the rank of the candidate, the color of the marker is between light green and black. If the mouse hovers over a marker, a text box is displayed that provides further information about the candidate. The prototype that was implemented for this bachelor's thesis is not able to suggest refactorings for methods with inner classes.

Evaluation. The prototype was used to evaluate the approach on seven Java open source systems with an online survey, where ten experienced developers and soft-

ware quality analysts participated. The evaluation aimed in answering five research questions.

RQ1 aimed to find out how severe the restriction of the prototype is, that it is not able to generate suggestions for methods with inner classes. For the seven open source systems that are considered for the evaluation, usually for more than $\frac{2}{3}$ of the too long methods, suggestions could be generated.

The evaluation showed that the suggestions of the approach for the survey objects are always better than a random candidate (RQ2).

The evaluation also addressed the question whether several candidates should be recommended (RQ3). For the half of the survey objects nearly all participants selected the same candidate and claimed that they would use it for an extract method refactoring – in these cases one suggestion might be sufficient. But for the other half of survey objects, the participants would apply several extract method refactorings that were suggested by the prototype. So, for 50% of the survey objects, at least the three best candidates should be suggested.

For 86% of the suggestions for the study objects, the developer would follow the suggestions made by the prototype (RQ4). That means that, at least for the study objects, the suggestion of the prototype are usually useful.

The last research question that is addressed by the evaluation (RQ5) is what information about refactoring candidates is actually necessary for developers to determine the most appropriate candidate. The results of the evaluation show that the input and output parameters of the candidates are very important for developers.

7.2. Future Work

This bachelor's thesis concludes with some suggestions on work that can be done in the future, that concern the approach, the prototype and the evaluation.

Approach. In the future, to raise usefulness for developers, the approach should be implemented as plug-ins for IDEs, as [MB07] suggest. They claimed that current refactoring tools are under-utilized by programmers and there is a need of tools that give more support on refactorings. In addition, the approach could be implemented in software quality analysis tools to give suggestions on how long method findings can be resolved.

The evaluation has shown that developers and SQAs often want to extract several candidates. To improve the approach, several candidates could be combined (if they do not intersect with each other) to obtain suggestions for several extract method refactorings in one "candidate". Therefore, several sets of extract method refactorings

could be compared with respect to the elements of the scoring function presented in this work. The prototype of the approach can easily be adapted in a way to get suggestions that cover several candidates.

The implementation of the approach uses weights for the single score elements that were influenced by the intuition of the author and his experience gathered while implementing and testing the prototype. The evaluation of this work has shown that these weights and parameters are fine, but probably are not optimal. To get a deeper understanding of the dependencies between the single criteria and to obtain more reliable weights and parameters one could use machine learning.

Like in the approach of [STV14] one could add an additional score element that represents the dependency between the candidate and the remainder of the method on class and package level to improve the quality of suggestions.

The approach may be useful as basis for tools that try to generate suggestions for other findings. For example, one could expand the approach to find refactoring suggestions for nested methods. For that, the candidate generation algorithm presented in this work can be used, only the scoring function should be adapted to get appropriate suggestions to resolve nested method findings.

Prototype. The prototype was implemented only for Java software systems and the evaluation covered only Java open source systems. The approach should work for all C-like languages with small adaptions on the weights and parameters of the scoring function elements. Therefore, tools that implement the approach might also consider other languages than Java.

Evaluation. The evaluation itself could be extended: There were only ten participants, two developers and eight SQAs, that evaluated the approach on ten too long methods that were picked from three open source systems. To get more reliable data on the usefulness of the approach, the evaluation should consider more participants, a higher number of survey objects that should be picked from systems, that are used in different domains and optimally are implemented in different programming languages.

The evaluation considered the question which information should be provided to developers and SQAs such that they can quickly select the most appropriate candidate for an extract method refactoring. A result was that information about the needed input and return parameters should be displayed. Further research needs to be done and more experience with tools that implement this approach need to be gathered to get clearer results for that question. The parameters could be highlighted, like [MB08] suggest in their paper, to get an even better overview about the dependencies between the candidate and the remainder of the original method.

Bibliography

[All70] F. E. Allen. "Control Flow Analysis." In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19.

[Dei+08] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka. "Tool Support for Continuous Quality Control." In: *Software, IEEE* 25.5 (Sept. 2008), pp. 60–67.

[Eic+01] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. "Does code decay? Assessing the evidence from change management data." In: *Software Engineering, IEEE Transactions on* 27.1 (Jan. 2001), pp. 1–12.

[Erl00] L. Erlikh. "Leveraging legacy system dollars for e-business." In: *IT Professional* 2.3 (May 2000), pp. 17–23.

[Fow99] M. Fowler. *Refactoring : Improving the design of existing code*. Addison-Wesley object technology series. Reading, Pennsylvania: Addison-Wesley, 1999.

[Hum] B. Hummel. *McCabe's Cyclomatic Complexity and Why We Don't Use It*. `https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/`, 2014. Accessed: 2014-10-23.

[ISO] ISO/IEC 25010:2011. *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*.

[KHK11] T. Kanemitsu, Y. Higo, and S. Kusumoto. "A Visualization Method of Program Dependency Graph for Identifying Extract Method Opportunity." In: *Proceedings of the 4th Workshop on Refactoring Tools*. WRT '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 8–14.

[Män05] M. Mäntylä. "An experiment on subjective evolvability evaluation of object-oriented software: Explaining factors and interrater agreement." In: *Empirical Software Engineering, 2005. International Symposium on*. Nov. 2005, pp. 287–296.

[Mar+10] R. Marticorena, C. López, Y. Crespo, and F. Pérez. "Refactoring Generics in JAVA: A Case Study on Extract Method." In: *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. Mar. 2010, pp. 212–221.

[Mar01] K. Maruyama. "Automated Method-extraction Refactoring by Using Block-based Slicing." In: *SIGSOFT Software Engineering Notes* 26.3 (May 2001), pp. 31–40.

[Mar09] R. C. Martin. *Clean code : A handbook of agile software craftsmanship*. Ed. by M. C. Feathers. Robert C. Martin series. Upper Saddle River, NJ [u.a.]: Prentice Hall, 2009.

[MB07] E. Murphy-Hill and A. P. Black. "Why don't people use refactoring tools?" In: *Proceedings of the 1st Workshop on Refactoring Tools*. ECOOP '07. TU Berlin. 2007, p. 61.

[MB08] E. Murphy-Hill and A. P. Black. "Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method." In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 421–430.

[MMK06] D. P. Mohapatra, R. Mall, and R. Kumar. "An Overview of Slicing Techniques for Object-Oriented Programs." In: *Informatica* 30 (2006), pp. 253–277.

[Opd92] W. F. Opdyke. "Refactoring object-oriented frameworks." PhD thesis. University of Illinois at Urbana-Champaign, 1992.

[Piw82] P. Piwowarski. "A Nesting Level Complexity Measure." In: *SIGPLAN Not.* 17.9 (Sept. 1982), pp. 44–50.

[SE14] D. Steidl and S. Eder. "Prioritizing Maintainability Defects Based on Refactoring Recommendations." In: *Proceedings of the 22Nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India: ACM, 2014, pp. 168–176.

[Sha12] T. Sharma. "Identifying Extract-method Refactoring Candidates Automatically." In: *Proceedings of the Fifth Workshop on Refactoring Tools*. WRT '12. Rapperswil, Switzerland: ACM, 2012, pp. 50–53.

[Str14] F. Streitel. "Incremental Language Independent Static Data Flow Analysis." Master's Thesis. Technische Universität München, 2014.

[STV14] D. Silva, R. Terra, and M. T. Valente. "Recommending Automated Extract Method Refactorings." In: *Proceedings of the 22Nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India: ACM, 2014, pp. 146–156.

[TC09] N. Tsantalis and A. Chatzigeorgiou. "Identification of Extract Method Refactoring Opportunities." In: *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*. Mar. 2009, pp. 119–128.

[WKK07] D. Wilking, U. F. Kahn, and S. Kowalewski. "An Empirical Evaluation of Refactoring." In: *e-Informatica* 1.1 (2007), pp. 27–42.

[YLN09] L. Yang, H. Liu, and Z. Niu. "Identifying Fragments to be Extracted from Long Methods." In: *Software Engineering Conference, 2009. APSEC '09. Asia-Pacific*. Dec. 2009, pp. 43–49.

A. Findings

Category	Finding	Example
General Checks	Empty blocks	`if(localName.equals("a")){}`
	Commented out code	`//object.setValue(1);`
Java Checks	Star import	`import org.apache.jempbox.xmp.*;`

	More than one top-level class in file	`public Class A {...}`
		`//-------------`
		`public Class B {...}`

Unused Code	Unused imports	`import import net.sf.jabref.Util // Util not used`
	Unused parameters	`public Button(String type) {...} // type not used`
Documentation	Interface comment missing	`public String t2String() throws IOException {...}`
	Task tags	`// TODO: what if this takes long time?`
Formatting	Multiple statements in single line	`try { in.close(); } catch (Exception ignored) {}`
	Multiple declarations in statement	`boolean file1Found = false, file2Found = false;`
	Condition or loop without braces	`if (fileWriter != null)` `  fileWriter.close();`

Other	Comment quality: unrelated member comment	`/**`
		` * Delete the temporary file.`
		` */`
		`public void tearDown() {...}`

Table A.1.: Examples for Findings from different Categories that can be easily resolved

Category	Finding	Example
Java Checks	Private field that could be made final	`private ZipFile zipFile = null;`
	Non-constant public attribute	`public int selectedInt = -1;`
Unused Code	Unused private methods/ private fields/ local variables	`private void setTop(){...} // not used`
Other	The value assigned to variable x was never read	`final String[] wrns = pr.warnings(); // never read`

Table A.2.: Examples for Findings where several Solutions – depending on the System Context – may be appropriate

Category	Finding	Example
Cloning	Clone with *x* instances	`A method that was cloned several times`
Naming	Violation of the Java naming conventions	`private URL ADSUrl = ...;`
Structure	Too big file size	`very long file`
	Too long method	`very long method`

	High nesting depth	`if ( keys != null ) {`
		`  int keyCount = keys.length;`
		`  for (String dummyStr : keys) {`
		`  if (dummyStr != null) {`
		`  mySet.add(dummyStr.trim());`
		`  }`
		`  }`
		`}`

Other	Null pointer dereference	`Object obj = null;`
		`...`
		`obj.toString();`

Table A.3.: Examples for Findings where Suggestions can not be easily generated