Deriving Extract Method Refactoring Suggestions for Long Methods*

Roman Haas[1] and Benjamin Hummel[2]

[1] Technical University of Munich, Germany
[2] CQSE GmbH, Garching near Munich, Germany
haas@in.tum.de, hummel@cqse.eu

Abstract. The extract method is a common way to shorten long methods in software development. Before developers can use tools that support the extract method, they need to invest time in identifying a suitable refactoring candidate. This paper addresses the problem of finding the most appropriate refactoring candidate for long methods written in Java. The approach determines valid refactoring candidates and ranks them using a scoring function that aims to improve readability and reduce code complexity. We use length and nesting reduction as complexity indicators. The number of parameters needed by the candidate influences the score. To suggest candidates that are consistent with the structure of the code, information such as comments and blank lines are also considered by the scoring function. We evaluate our approach to three open source systems using a user study with ten experienced developers. Our results show that they would actually apply 86% of suggestions for an extract method refactoring.

Key words: Refactoring suggestion · Long method · Extract method

1 Introduction

Long methods are a bad smell in software systems [1]. This means that they do not influence the behavior of the code directly, but make it harder to understand and therefore harder to maintain [11].

A common way to treat long methods is to apply extract method refactorings, where parts of the long method are extracted and put into a new method. In practice, this is one of the most often applied refactorings [14].

The refactoring process using a modern IDE like Eclipse, NetBeans or IntelliJ IDEA [3] consists of the following steps. First, developers need to identify a sequence of statements that should be extracted. Second, they need to select the statements. Third, they call a tool that will, if possible, execute the refactoring. Fourth, before the refactoring can be applied by the tool, the

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

developer needs to specify a name for the new method. Finally, the refactoring is executed.

Tool support makes refactorings much easier, but a developer still needs to select some source code within the method that they would like to extract. This can be complicated, time-intensive, tedious, and error-prone [8].

Problem Statement. Developers need to invest a considerable amount of time in finding the sequence of statements best suited for an extract method refactoring.

The majority of development tools only provide support to execute refactorings that are specified by the developer. In the context of long methods this means that the most time consuming step in shortening a method still needs to be done by developers themselves, which is a reason refactoring tools are not used as much as they could be [6].

Sometimes, even experienced developers select invalid refactoring candidates because they have overlooked a violation of preconditions that must hold for the extract method. In such cases, current tools give poor information on why no refactoring is possible [7].

Extract method refactoring suggestions are helpful for developers because they save time and make fewer mistakes during the candidate selection.

Contribution. We present an approach to automatically finding extract method refactoring suggestions for long methods in Java projects. The approach generates a list of possible refactorings and ranks those using a scoring function. The ranking focuses on readability improvement and reduction of code complexity. The scoring function uses structural information given by the developer to reward bonus points. Additionally, it takes into account the number of parameters needed by the new method. We evaluate our approach on three open source systems using a user study with ten experienced developers.

2 Related Work

There are several ways to suggest extract method refactorings which can be divided into four categories. Some use program slicing techniques to find recommendations for extract method, while others try to find suitable suggestions from graph representations of the code. Some rely on scoring functions to find the most appropriate refactoring candidate. Refactoring prioritization tries to identify methods that are actually worth for refactoring.

Program Slicing Based Approaches. Maruyama [5] presents a semi-automated mechanism for refactoring suggestions. It decomposes the control flow graph using block-based program slicing. The approach is adapted and implemented by the tool JDeodorant by Tsantalis and Chatzigeorgiou [13] that improves behavior preservation. According to Sharma [9], approaches that use program slicing techniques cannot be fully automated as the user has to select a slicing criterion for every method that should be refactored. In addition, the suggestions

depend heavily on the user's input. As we wanted to have an approach that is able to find extract method refactorings automatically, we did not rely on a program slicer.

Graph Based Approaches. Sharma [9] provides a mechanism to propose extract method candidates based on a data and structure dependency graph. Their suggestions are obtained by deleting the longest dependency edge in the graph. The resulting two disconnected subgraphs represent the statements that stay within the original method or which will be extracted to a new method, respectively. They are able to suggest non-continuous statements for extract method. We use a control and data flow graph to represent methods. We do not obtain suggestions from operations on the graph but determine valid candidates that are ranked using a scoring function.

Kanemitsu et al. [2] use a program dependency graph and recommend that users extract all nodes that are connected via edges not longer than a user-defined maximal length. Their approach was led by the design principle that one method should process only one thing. We consider the same design principle by rewarding bonus points to candidates that have comments or blank lines at the beginning or the end because they are often indicators that something new has been processed by the preceding or following lines. Our scoring function also considers code complexity reduction and the number of needed parameters.

Score Based Approaches. Silva et al. [10] suggest an approach to automatically generate candidates for method extraction. Their scoring function ranks candidates with respect to static dependencies between variables, types, and packages. Our approach was inspired by Silva et al. as the general procedure of candidate generation is similar and we were also not able to suggest candidates with non-continuous statements. The scoring function of our approach does not consider dependencies but mainly the reduction of length and nesting with the aim of reducing code complexity and increasing maintainability.

Yang et al. [15] consider long methods and suggest an approach to recommending refactorings that lead to as small a coupling as possible, automatically. Their scoring function is the benefit-cost ratio of the length of the extracted method and the numbers of needed input and output parameters. In contrast to Yang et al. we do not move variable declarations as far back as possible. Our scoring function also considers the number of parameters needed and we reward bonus points for comments or blank lines (which are a splitting criterion for Yang et al. to obtain their candidates). Additionally, reduction of code complexity has a high impact on the ranking.

Prioritization. Steidl and Eder [11] focus on 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.

Steidl and Eder's approach is not appropriate for automated refactorings as it only gives a hint of where a developer should start refactoring.

3 Approach

We present an approach to finding extract method refactorings for long methods automatically. There are two fundamental steps: first, the generation of all possible refactoring candidates (i.e. all sequences of statements that can be extracted). Second, ranking all of them by applying a scoring function that considers reduction of complexity and structural information of the candidates. The candidates with the highest ranking will be suggested for an extract method refactoring.

3.1 Candidate Generation

The procedure of generating all possible candidates is quite similar to the one that Silva et al. [10] presented. They introduced a minimal number of statements, K, that ensures suggestions do actually have some benefit. In their evaluation they found that $K = 3$ is optimal and therefore, our approach also sets a minimal number of statements $K = 3$, which must hold for the number of statements of a candidate and the corresponding remainder of the long method.

To obtain valid refactoring candidates we use the software quality analysis tool ConQAT[2] and Streitel's implementation of control and data flow graphs [12] to check that several preconditions hold. Most importantly, an extract method candidate may not need more than one return parameter (see [7] for details).

3.2 Scoring Function

All valid extract method refactoring candidates obtained in the first step are ranked using a scoring function that focuses on code complexity reduction. We rely on length and nesting metrics as complexity indicators. The scoring function also takes structural information, like blank lines or lines with comments, and the number of parameters into account. For each scoring element (see figure 1) a score value is determined and all score values summed up lead to the total score of a candidate. The candidate with the highest score will be our first suggestion.

Length We aim at suggestions that reduce complexity and consider length as an complexity indicator. Therefore, the length of a refactoring candidate influences its ranking. 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 remainder L_r. For each line a constant number of points c_l is awarded, up to the upper bound $MAX_{scoreLength}$. This upper bound ensures that very long candidates are not ranked higher just because they are extraordinarily long. We set

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

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

[2] www.conqat.org

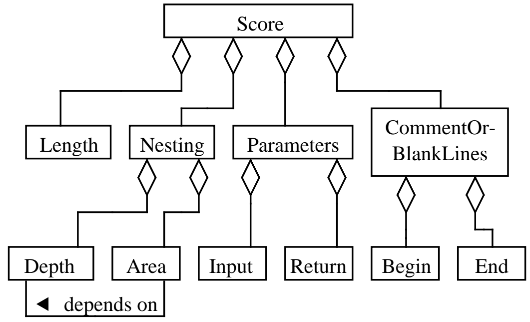

Fig. 1. Score Elements

Nesting Depth We use nesting depth as another indicator of code complexity. Let D_m be the nesting depth of the original method, D_r the nesting depth of the remainder, and D_c the nesting depth of the refactoring candidate. The score of a candidate obtained for reducing the nesting depth is set to

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

which means that (theoretically) there is no upper bound for $S_{nestDepth}$. But note 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.

Unfortunately, nesting depth often cannot be reduced by extract method if one considers both the remaining method and the candidate. It is often the case that there are several deeply nested statements that cannot be extracted at once without extracting the whole nesting structure: either one suggests a candidate that includes only some of these statements (which will not reduce nesting depth of the original method) or one chooses a candidate that extracts the whole deeply nested structure, leading to a new candidate which is as deeply nested as the original method was.

Nesting Area To have a measure for nesting reduction that is more often applicable we consider the reduction of nesting area.

In formal terms, the nesting area of a sequence of statements S_1 to S_n, each having a nesting depth of d_{S_i}, is defined as $\sum_{i=1}^{n} d_{S_i}$. Intuitively spoken, it is 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 at once 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 we consider the reduction of nesting area. If nesting structure is simplified by extracting parts of it, we claim that complexity is reduced. The deeper the extracted statements are nested, the bigger the benefit is

in terms of complexity reduction. We aim for maximizing nesting area reduction ($A_{reduction}$). That is 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. For a given method with nesting area A_m an optimal candidate can achieve a reduction of at most $\lfloor \frac{A_m}{2} \rfloor$, similar to the maximal nesting depth reduction.

We assume that reducing the nesting area becomes more important as the nesting depth of the original method D_m becomes higher. Therefore, 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. Remark that if nesting depth (i.e. complexity) is high, the other criteria have less relevance for the scoring of the candidates as nesting scores are not bounded while the other scoring criteria are bounded.

Parameters To obtain the most independent candidates with respect to coupling, we consider 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. For the parameter score S_{param} there is an upper bound $MAX_{scoreParam}$. The number of needed parameters (n_{in} and n_{out}, where $n_{out} \leq 1$) will reduce the score, and each parameter decreases the score by one:

$$S_{param} = MAX_{scoreParam} - n_{in} - n_{out}$$

Fowler [1] claims that having a long parameter list is a bad smell. He proposes to not have more than three input parameters. As we have in Java at most one return parameter, we set $MAX_{scoreParam} = 4$.

Comments and Blank Lines To capture additional developer's knowledge, we award bonus points for comments and blank lines. Developers often have comments that give information about the next source code line(s), especially if these perform something different than the previous ones. In other cases, blank lines separate such different tasks. But this is a violation of the design principle that one method should process only one thing (see [4]) and therefore, the following lines might be a good candidate for an extract method refactoring. The bonus we award for candidates that have such lines with comments or blank lines at the beginning or the end is as follows: for each such line (and the fact that these lines exist) c_p many points are obtained. In our experiments we saw 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 because they give more information about itself. In the score formula the

higher relevance is represented by the factor $f_b > 1$. In addition, several lines of comments before a sequence of statements indicate a more complex explanation which is more likely to describe a new functionality and therefore, more lines with comments or blank lines get more points.

The score depends on four variables: the existence of blank lines or comments 1) at the beginning (e_b) and 2) at the end (e_e) of a refactoring candidate. 3) the number of blank lines or comments at the beginning (n_b) and 4) at the end of a candidate (n_e), where $e_x \in \{0,1\}, n_x \in \{0,1,2,3\}$ and $x \in \{b,e\}$. If there are more than three blank lines or comments the same amount of points is awarded as if there were only three blank lines or comments. We set

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

For our prototype $f_b = 2$ holds, i.e. preceding comments result in twice as many points as comments at the end. c_p was set to 0.25 such that a candidate may get up to 2 points for having at least three comment or blank lines at the beginning, and up to 1 point for having at least three of such lines at the end.

Scoring Elements Intervals The previous subsections gave detailed information about the single criteria for the score of a refactoring candidate. Table 1 shows the intervals of the single scoring elements. D_m stands for the nesting depth of the original method.

Table 1. Scoring Elements and their Intervals

Score Element	**Max Score**
S_{length}	3
$S_{nestDepth}$	$\lfloor \frac{D_m}{2} \rfloor$
$S_{nestArea}$	D_m
S_{param}	4
$S_{commentsBlankLines}$	3

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:

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

3.3 Pruning

At the end of our suggestion algorithm, the list of candidates is optimized. As all possible candidates are generated, there are several ones that differ only in one or two statements at the beginning or the end. Those often have similar scores because they refer to nearly the same piece of code and the differing statements do not change the score that much. To obtain a wide range of suggestions, candidates are removed from the list if there is another candidate containing all of their statements, having the same input and return parameters, and having a better score.

4 Evaluation

This section evaluates our approach using a prototype that is implemented as a ConQAT analysis for Java projects. We constructed an online survey that presented ten long methods from open source Java projects with extract method refactoring suggestions.

RQ1: 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 than the suggestions of this approach, the approach with its scoring function would be useless.

RQ2: Do developers follow the suggestions of this approach? This question considers a much stronger criterion of usefulness than RQ1. The evaluation of this paper tries to find out, whether (and how often) this approach is able to suggest candidates that are taken as refactoring candidate from developers. The more often developers follow the suggestions of the prototype, the closer is the scoring function on their intuition.

RQ3: Should several suggestions be made? This question addresses a result of Silva et al. [10]. They claimed that an implementation of their approach should preferably suggest only the best candidate. As the approach of this paper is structurally similar to their approach, we try to find out whether their result also holds for our prototype.

4.1 Design

For the survey, all participants received an HTML file that contained ten methods (survey object) that were considered during the survey. All these methods had between 48 and 73 lines. For each survey object there were three highlighted candidates. One of the candidates was always the first suggestion of the prototype, called *TOP1*. Another one was the second or third suggestion of the prototype (which one 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 one of the TOP3 candidates determined by the scoring function, called *Random*. All suggested candidates were highlighted in the same way such that the participants could not differentiate them.

Table 2 shows the study objects from which ten long methods were presented in the survey. We consider a method as long if it counts more than 40 lines. All study objects are Java open source projects. They all have long methods but some have – in relative terms – more long methods than others. 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[3].

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

Table 2. Study Objects

Name	Domain	Size (LoC)	# Methods	# Long Methods	LoC of Longest Method
Agilefant	Backlog Tool	36,116	2,841	31 (1.09%)	143
JabRef	Reference Manager	128,145	5,665	428 (7.56%)	1,305
JChart2D	Charting Library	50,728	1,849	72 (3.89%)	641

Our online survey asked for each survey object the following questions:

1. Which candidate would you use more likely for an extract method refactoring? The participants could select exactly one suggestion.
2. Would you use the selected candidate for an extract method refactoring? In addition to "Yes" and "No", the participants could select "Yes, with slight modification (of 1-2 lines)".
3. Would you have applied an extract method refactoring on this method? Answering options were "Yes" and "No".

4.2 Results

Ten experienced developers participated in the survey that is used to answer RQ1-RQ3. All of them have between 6 and 24 (on average 12) years of development experience.

Are suggestions better than a random (valid) refactoring candidate? (RQ1) 74% of the selected candidates in the first survey question were the one that was ranked top most by the scoring function. The other 26% were 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 suggestions generated by the prototype are much better than the selection of a valid random candidate.

Do developers follow the suggestions of this approach? (RQ2) 74% of the selected candidates would have been applied without modifications (according to the answers to the second survey question). For other 12% a quite similar refactoring would be applied (by only shifting the selected candidate about one or two lines). For the remaining 14% the developers claimed that they would not have applied the selected refactoring. For 93% of the survey objects developers would apply an extract method refactoring on the presented method. 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? (RQ3) 74% of the selected candidates were 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. 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).

4.3 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.

Do developers follow the suggestions of this approach? (RQ2) 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. Nevertheless, 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 those and then refactor the resulting method. But as they would not start with an extract method refactoring, they answered in such cases the third survey question with "No". 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.

This means that not all methods that are considered lengthy by our prototype are candidates for developers for extract method refactorings. In general, the suggestions are helpful: 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? (RQ3) Many participants mentioned in their feedback that there were some methods where they were quite sure which of the suggested candidates 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 often it is the best solution to refactor a 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.

4.4 Threats to Validity

There are some threats to validity of the evaluation, which are summarized in the following.

Resolution of long methods is subjective (see [15]). First, there is no consensus in science when a method is actually long. This means that some may treat a given method as long where others do not. Second, there is no commonly accepted algorithm that splits a 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 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.

Ten survey objects do not represent the whole spectrum of methods that should be refactored using an extract method refactoring. They all covered (for 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 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.

5 Conclusion and Future Work

We proposed an approach to derive extract method refactoring suggestions for long methods in Java to improve maintainability and reduce code complexity. The approach determines extractable candidates from the control and data flow graph of a method. A refactoring candidate needs to fulfill syntactical preconditions, have an equivalent data flow and a minimal length. Each candidate is ranked using a scoring function that considers the following criteria: length, nesting depth and area reduction, and the number of input and return parameters. Bonus points are awarded for candidates having comments or blank lines at their beginning or end.

We used a prototype to evaluate our work in a survey. It showed that the suggestions of the approach for the survey objects are always better than a random candidate (RQ1). For 86% of the suggestions for the study objects, the developers follow the suggestions made by the prototype (RQ2). This means that, at least for the study objects, the suggestions of the prototype are usually useful. We also addressed the question whether several candidates should be recommended (RQ3). For one 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.

We think that our approach also works for methods with a high nesting depth (which is another code smell). In the future, we want to conduct another case study to test the validity of this hypothesis. We plan to do further research on the choice and weights of our scoring parameter.

Instead of suggesting several candidates from which developers can choose at least one, one could suggest a set of disjoint extract method refactorings. The scoring function then could consider the benefit of applying all these refactorings instead of ranking single suggestions.

References

1. Fowler, M.: Refactoring: Improving the design of existing code. Addison-Wesley, Reading (1999)
2. Kanemitsu, T., Higo, Y., Kusumoto, S.: A Visualization Method of Program Dependency Graph for Identifying Extract Method Opportunity. In: Proceedings of the 4th Workshop on Refactoring Tools, pp. 8–14. ACM (2011)
3. Marticorena, R., Lpez, C., Crespo, Y., Prez, F. J.: Refactoring Generics in JAVA: A Case Study on Extract Method. In: 14th European Conference on Software Maintenance and Reengineering (CSMR), pp. 212–221. IEEE (2010)
4. Martin, R.C.: Clean code: A handbook of agile software craftsmanship. Prentice Hall, Upper Saddle River (2009)
5. Maruyama, K.: Automated Method-extraction Refactoring by Using Block-based Slicing. In: ACM SIGSOFT Software Engineering Notes, vol. 26, pp. 31–40. ACM (2001)
6. Murphy-Hill, E., Black, A.P.: Why don't people use refactoring tools?. In: Proceedings of the 1st Workshop on Refactoring Tools, pp. 60–61 (2007)
7. Murphy-Hill, E., Black, A.P.: Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In: Proceedings of the 30th International Conference on Software Engineering, pp. 421–430. IEEE (2008)
8. Opdyke, W.F.: Refactoring object-oriented frameworks. PhD thesis. University of Illinois at Urbana-Champaign (1992)
9. Sharma, T.: Identifying Extract-method Refactoring Candidates Automatically. In: Proceedings of the 5th Workshop on Refactoring Tools, pp. 50–53. ACM (2012)
10. Silva, D., Terra, R., Valente, M.T.: Recommending Automated Extract Method Refactorings. In: Proceedings of the 22nd International Conference on Program Comprehension, pp. 146–156. ACM (2014)
11. Steidl, D., Eder, S.: Prioritizing Maintainability Defects Based on Refactoring Recommendations. In: Proceedings of the 22nd International Conference on Program Comprehension, pp. 168–176. ACM (2014)
12. Streitel, F.: Incremental Language Independent Static Data Flow Analysis. Master's thesis. Technical University of Munich (2014)
13. Tsantalis, N. Chatzigeorgiou, A.: Identification of Extract Method Refactoring Opportunities. In: 13th European Conference on Software Maintenance and Reengineering, pp. 119–128. IEEE (2009)
14. Wilking, D., Kahn, U.F., Kowalewski, S.: An Empirical Evaluation of Refactoring. e-Informatica 1(1), pp. 27–42 (2007)
15. Yang, L., Liu, H., Niu, Z.: Identifying Fragments to be Extracted from Long Methods. In: Asia-Pacific Software Engineering Conference, pp. 43–49. IEEE (2009)