Size and cohesion metrics as indicators of the long method bad smell: An empirical study

Sofia Charalampidou, Apostolos Ampatzoglou, Paris Avgeriou

Department of Computer Science
University of Groningen
Groningen, the Netherlands
s.charalampidou@rug.nl, a.ampatzoglou@rug.nl, paris@cs.rug.nl

ABSTRACT
Source code bad smells are usually resolved through the application of well-defined solutions, i.e., refactorings. In the literature, software metrics are used as indicators of the existence and prioritization of resolving bad smells. In this paper, we focus on the long method smell (i.e. one of the most frequent and persistent bad smells) that can be resolved by the extract method refactoring. Until now, the identification of long methods or extract method opportunities has been performed based on cohesion, size or complexity metrics. However, the empirical validation of these metrics has exhibited relatively low accuracy with regard to their capacity to indicate the existence of long methods or extract method opportunities. Thus, we empirically explore the ability of size and cohesion metrics to predict the existence and the refactoring urgency of long method occurrences, through a case study on java open-source methods. The results of the study suggest that one size and four cohesion metrics are capable of characterizing the need and urgency for resolving the long method bad smell, with a higher accuracy compared to the previous studies. The obtained results are discussed by providing possible interpretations and implications to practitioners and researchers.

Categories and Subject Descriptors
D.2.3 [Coding Tools and Techniques]: *Object-Oriented Programming, Standards;* **D.2.8 [Metrics]:** *Product Metrics;* **D.2.10 [Design]:** *Methodologies.* The ACM Computing Classification Scheme: http://www.acm.org/class/1998/

General Terms
Measurement; Design; Experimentation

Keywords
Long method; metrics; cohesion; size; case study

1. INTRODUCTION
Code *bad smells* are symptoms indicating that a specific part of the source code is neglecting at least one programming principle [18]. By definition, bad smells are concrete problems closely related to applicable solutions, i.e. *refactorings*, which can alleviate the caused problems [22], [23]. However, the manual identification of such smells or refactorings opportunities is extremely challenging in large code bases; manual source code inspection would be time prohibitively expensive. To avoid manual source code inspection, several methods and tools aim at identifying code bad smells or refactoring opportunities. Nevertheless, the applicability of such methods and tools is usually limited due to their dependency to specific programming languages or IDEs. Additionally, in most of the cases, these methods and tools do not prioritize the identified bad smell occurrences; this is problematic due to the vast number of refactoring opportunities that can be identified in a single system. To tackle both problems, several generic-scope software metrics [15] have been proposed for recognizing parts of the code base that need refactoring, and for characterizing their urgency (see Section 2).

In this study we propose such a metric-based approach, focusing on one specific bad smell – the long method, which is resolved through the extract method refactoring, as defined by Fowler et al. [18]. The reasons for working with this smell are:

- ***The frequency of its occurrence***. Long method is a frequently occurring smell [10]. The case study reported in [10], aims to investigate the presence and evolution of four types of code smells, i.e., *Long Method, Feature Envy, State Checking* and *God Class*. The results indicated that the long methods were considerably more common compared to the other smells.
- ***Its persistence during evolution***. Long methods are of particular urgency as they often occur in the early versions of software and persist unless targeted refactoring activities are performed. Specifically, a case study on an open source project (jFlex) revealed that 89.8% of the long methods identified in that project remained unresolved in all the explored versions [10].
- ***The lack of metrics, related to long methods.*** So far only a few metrics have been assessed with respect to their capacity to predict the existence of long methods. To the best of our knowledge, current approaches achieve a recall rate of 59% and a precision rate between 39%-66% (see Section 2). Also the ability of metrics to prioritize the urgency of the long methods to be resolved has not been empirically investigated yet.

In order to use a metric-based approach for identifying bad smells or refactoring opportunities, one needs to specify unique characteristics for each bad smell (e.g. a 'god class' is large in size), leading to the selection of quality properties (i.e., concepts that can be directly evaluated by exploring the structure of software elements [5]), and subsequently metrics. By definition, the long method smell concerns methods large in size, which have a semantic distance between the major purpose of the method, with respect to a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PROMISE '15, October 21 2015, Beijing, China
© 2015 ACM. ISBN 978-1-4503-3715-1/15/10…$15.00
DOI: http://dx.doi.org/10.1145/2810146.2810155

specific functionality, and the degree to which its implementation serves this purpose [18]. In other words, we do not perceive all methods large in size as 'long', but only those whose large size is due to the implementation of multiple functionalities. Based on this definition, a property that can be used as an indicator of the number of functionalities that a module offers is cohesion [26], [31].

Therefore, we focus on two quality properties: method size and cohesion. Specifically, we will empirically investigate the ability of size and cohesion metrics: (a) to predict which methods suffer from the *long method bad smell*, and (b) to prioritize their *urgency to be resolved*, based on the extract method opportunities that they present. We note that we assess the urgency of a long method to be refactored based on the identified extract method opportunities, since Fowler et al. suggest that in 99% of the cases, the *extract method refactoring* (i.e. removal of code chunks from one method that can be turned into new methods, whose names explain their purpose [18]) is the solution to the long method bad smell.

In the next section, we present related work that used metrics for detecting refactoring opportunities or bad smells. In Section 3 we present the quality properties that are associated with the long method bad smell and the selected software metrics. In Section 4 we present the case study design for evaluating these metrics as indicators for the prediction and prioritization of long method smell. In Section 5 we report the findings of the case study, which are discussed in Section 6. Threats to validity are discussed in Section 7. Finally, in Section 8 we conclude this paper.

2. RELATED WORK

The related work section concerns studies that evaluate existing metrics, with respect to their ability of identifying and prioritizing refactoring opportunities or detecting bad smells. Therefore, studies which use other approaches for identifying software artifacts that suffer from bad smells or present refactoring opportunities (e.g., inspection of the revision change history [38], or exploration of the cohesion lattice structure [21], or exploitation of computational slices [44]), have been excluded from this section. Also, we omitted papers that propose new metrics for identifying refactoring opportunities or bad smells (e.g. [14] and [42]). Although such studies can be considered as indirect related work (our study uses existing metrics), we preferred not to present them, due to space limitations. While presenting related work, more emphasis is given to studies related to long methods.

Refactoring Identification & Prioritization: Yoshida et al. proposed the division of the source code into functional segments that could be used as the basis for extracting methods (fragments of code that concern a unique functionality) and the employment of a cohesion metric (NCCP, which is based on SCOM [16]) for their identification [45]. To validate the outcome of the proposed method the authors performed a single case study on one project with one of its developers as evaluator. In the optimum application of the approach, 51 out of 80 unique functionalities were correctly identified (recall: 59%).

Meananeatra et al. [34] presented a method for prioritizing five refactorings (*Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, and Decompose Conditional*) with respect to improving maintainability. The method employs three maintainability metrics (*complexity*, *size*, and *cohesion*) before and after applying the refactoring. Then a set of metrics, related to data and control flow graphs, is calculated for selecting applicable refactorings, based on some pre-defined criteria. The measurement applied after the refactoring, indicates which are the most capable for optimizing maintainability.

Zhao and Hayes developed a tool for identifying classes in need of refactorings, however, without specifying the bad smell that they suffer from [46]. Their approach was based on *size* and *complexity* metrics, combined through a weighted ranking method, for prioritizing the most urgent classes to be refactored. The tool was validated by comparing its results to those obtained through manual inspection. The outcome of the validation was that the results of the tool can be supportive for the software development teams.

Demeyer et al. proposed a refactoring detection method by applying lightweight, object-oriented metrics to successive versions of a software system [13]. The selected metrics concerned three major aspects: *method size* (number of message sends in method body, number of statements in method body, lines of code in method body), *class size* (number of methods in class, number of instance variables) and *inheritance* (hierarchy nesting level, number of immediate children of class, number of inherited methods, number of overridden methods). The detected refactorings were the following: *Split or Merge Superclass / Subclass*, *Move to other Class*, and *Split Method*. The approach was validated by three case studies, suggesting that the refactoring identification strategy supports reverse engineering, by focusing on the relevant parts of a system. The precision of this approach ranges from 38% to 66% for the *Split Method* refactoring.

Bad Smell Identification: In 2004 Marinescu proposed a mechanism for detecting design problems [33]. In contrast to other studies that try to infer problems from a set of abnormal metric values, this approach defines metric-based rules that identify deviations from good design principles and heuristics. As a result, it is able to locate classes or methods affected by design flaws. The approach was validated experimentally on multiple case-studies by identifying nine design flaws *(*including *God Method)*. Concerning the identification of the *God Method* smell[1] Marinescu proposes the use of complexity metrics assuming that complexity should be uniformly distributed among methods. The precision of this approach on the identification of God method smells is 50%.

The aforementioned approach has been applied by Mihancea and Marinescu, for establishing metrics-based rules, which detect design flaws in object-oriented systems [35]. The method searches for thresholds that maximize the number of correctly classified entities, by combining existing metrics. For validation the *God Class* and *Data Class* flaws were detected. For the identification of both flaws complexity, cohesion and coupling metrics were used. Next, in 2006 Lanza and Marinescu collected in a book entitled "Object Oriented Metrics in Practice" six well-known bad smells (*God Class, Feature Envy, Data Class, Brain Method, Brain Class* and *Significant Duplication*), which they present in details, along with strategies for detecting them. These strategies included the use of 24 of metrics, in total, and thresholds, which are different for each smell. Thus, their detailed presentation is out of the scope of this section [27].Mäntylä et al. investigated the identification of bad smells based on possible correlation between human critics and metrics provided by existing tools [30]. The bad smells under investigation were the *large class*, *long parameter list* and *duplicate code*. The results showed no correlation between the two sources. Khomh et al. proposed a Bayesian network ap-

[1] Although this approach is able to identify *God* instead of *Long* Methods, we consider it comparable approach to ours, due to smells' similarity. The difference between these smells is: "*Long Methods have a large number of LoC. In addition to being long, God Methods have many branches and use many attributes, parameters, local variables.*" [43]

proach for handling the inherent uncertainty in the process of identifying code or design smells [24]. This study was based on the detection rules proposed by Moha et al. [36] for the identification of the *Blob antipattern*. As an example, they suggest that classes with more than 90% of *accessor methods* can be characterized as *data classes*.

Salehie et al. proposed a metric-based heuristic framework for detecting and locating object-oriented design flaws [40]. The framework assesses the design quality of internal and external structure of a system, at the class level, in two phases. In the first phase, hotspots are detected using metrics aiming at indicating a design feature (e.g., high *complexity*). In the second phase, individual design flaws are detected using a proper set of metrics. The use of the framework is presented for the *God Class* and the *Shotgun Surgery* flaws, by employing coupling, cohesion and complexity metrics. The framework was applied on the JBoss Application Server, i.e., a large size system with pure object-oriented structure, in order to set threshold values to the used metrics.

Related Work Overview & Contributions: From the aforementioned related work, it becomes clear that only three studies (i.e., [13], [33], [45]) have explored the identification of long methods, through metrics. From these studies only the approach of Yoshida et al. employs cohesion metrics for this purpose, however by focusing only on one metric. Therefore the contributions of this study can be summarized as follows:

- It relates a variety of cohesion metrics with the existence of long methods.
- It relates cohesion metrics to the prioritization of resolving long methods.
- It compares size / cohesion metrics, as predictors of the existence of long methods and their urgency for refactoring.
- It provides a method of higher accuracy (precision and recall), compared to the state of the art.
- It is one of the few tools that perform identification of long methods, instead of extract methods opportunities (e.g., JDeodorand [44], JExtract [41], etc.).

3. METRICS SELECTION

The first step towards relating long methods and existing software metrics is to find out which quality properties could be related to it and subsequently, which metrics could be used for quantifying those quality properties. According to the definition provided by Fowler [18], a long method is characterized by: (a) its size, and (b) the functional distance of the lines of code of its body. First, the size of a method is a quality property per se, and one way to measure it is by counting the uncommented lines of code (LOC). Second, the functional distance is related to cohesion, which is defined as the functional relatedness of the elements of a module [31]. We note that the relation between cohesion and functional distance is inverse (i.e., when functional distance increases, cohesion decreases). However, the selection of cohesion metrics that would be useful indicators of the functional distance in the body of a method is a complex task, for two reasons:

- ***The plethora of available cohesion metrics***. Al Dallal has reported 16 class-level metrics [3]. As a result there is a need for an empirical evaluation of the ability of each metric to indicate the existence of long methods and their refactoring priority. This need becomes even more evident by taking into account that each one of these metrics addresses a different notion of cohesion [37]. Therefore, it is beneficial to also investigate if these different aspects of cohesion lead to different capabilities for long method identification and prioritization.
- ***The lack of cohesion metrics that can be calculated inside the method body***. According to Al Dallal, cohesion metrics are applicable at class level and are classified into two categories, namely *high-level* and *low-level* cohesion metrics [3]. For the purpose of this study none of these metrics is directly applicable, in the sense that they cannot assess cohesion inside the method body. Specifically, the high-level metrics calculate cohesion, based on methods' parameters, and thus they cannot be mapped to the method body level. On the contrary, the low-level metrics, which calculate cohesion by characterizing pairs or sets of methods as cohesive, can be transformed to assess the cohesion inside the method body.

The application of low-level cohesion metrics at the method-level (i.e., inside the method body) was also discussed by Yoshida et al. [45], when introducing the NCCP metric, i.e., a new, method-level cohesion measure, which has been derived from the transformation of the SCOM metric [16]. In our approach, we have applied a process, similar to the one proposed by Yoshida et al. [45], for all 13 low-level cohesion metrics collected by Al Dallal [2]. The main principles for this process are the mapping of:

- Lines of code to methods, and
- All variables within the scope of the method (i.e., attributes, local variables, or parameters) to attributes.

In Table 1 we present the 13 cohesion metrics used in this study, accompanied by their definitions, after they were transformed to apply to the method-level. Also, we name the original study in which the class-level cohesion metric was introduced.

Table 1. Method Level Cohesion Metrics

Cohesion Metric	Application on method level
LCOM1 [11]	LCOM1 = P, where `P` is the number of pairs of lines that do not share variables.
LCOM2 [12]	LCOM2 = P – Q, if P − Q ≥ 0 / otherwise LCOM2 = 0, where `P` is the number of pairs of lines that do not share variables, and `Q` is the number of pairs of lines that share variables.
LCOM3 [28]	Number of connected components in a graph, where each node represents a line of code and each edge the common use of at least one variable.
LCOM4 [20]	Similar to LCOM3. Method calls are treated as edges.
LCOM5 [19]	LCOM5 = (a - nl) / (l - nl) where `n` is the number of lines, `a` is the number of variables used in a line, and `l` is the total number of variables.
Coh [9]	Coh = 1 – (1 – 1/n) LCOM5 where `n` is the number of lines
Tight Class Cohesion (TCC) [7]	TCC = NDC / NP where `NDC` the number of directly connected pairs of lines (i.e. accessing a common variable either within the line or within the body of a method invoked in that line directly or transitively), and `NP` the maximum possible number of direct connections in a method.

Cohesion Metric	Application on method level
Loose Class Cohesion (LCC) [7]	LCC = (NDC + NIC) / NP where NDC and NP as defined above, and NIC the number of indirectly connected pairs of lines. A pair of lines is indirectly connected, if they access no common variables, but there is a line directly connected to both lines of the pair.
Degree of Cohesion-Direct (DC_D) [4]	$DC_D = \|E_D\| / [n * (n – 1) / 2]$ where E_D the number of edges in a graph connecting directly related lines of code (i.e. as defined for TCC or in cases that the lines directly or transitively invoke the same method), and n the number of lines of a method.
Degree of Cohesion-Indirect (DC_I) [4]	$DC_I = \|E_I\| / [n * (n – 1) / 2]$ where E_I the number of edges in a graph connecting indirectly related lines of code (i.e. as defined for LCC or in cases that the lines directly or transitively invoke the same method), and n the number of lines of a method.
Class Cohesion (CC) [8]	$CC = \frac{1(n-2)!}{n!} \sum_{i=1}^{\frac{n!}{2(n-2)!}} \frac{\|IV\|_c}{\|IV\|_t} i$ where n the number of lines of a method, $\|IV\|_t$ is the total number of variables used by two lines and $\|IV\|_c$ the number of common variables used by both lines.
Class Cohesion Metric (SCOM) [16]	$SCOM = \frac{2}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{card(I_i \cap I_j) * card(I_i \cup I_j)}{\min(card\ (I_i), card\ (I_j)) * a}$ where n is the number of lines of a method, $card(I_i \cap I_j)$= $\|IV\|_c$ as defined for CC, $card(I_i \cup I_j)$= $\|IV\|_t$ as defined for CC, $\min(card\ (I_i), card\ (I_j))$ is the minimum number of variables accessed between the two lines, and a is the number of variables accessed in the method.
Low-level design Similarity-based Class Cohesion (LSCC) [3]	$LSCC = \begin{cases} 0 & \text{if } l=0 \text{ and } n>1 \\ 1 & \text{if } (l>0 \text{ and } n=0) \text{ or } n=1 \\ \frac{2}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} ns(i,j) & \text{otherwise} \end{cases}$ where n is the number of lines, l the number of variables in the method of interest, and ns the normalized similarity between a pair of lines.

4. CASE STUDY DESIGN

The objective of this case study is to investigate the ability of one size (lines of code – LOC) and 13 cohesion metrics (presented in Section 3) to provide indications on the existence of long methods, and their urgency to be resolved. The case study has been designed and reported according to the template suggested by Runeson et al. [39]. The next sections contain the four parts of the research design, i.e., *Objectives and Research Questions*, *Case Selection and Units of Analysis*, *Data Collection, Pre-Processing*, and *Analysis*.

4.1 Objectives and Research Questions

The goal of the study is described using the Goal-Question-Metric (GQM) approach [6], as follows: "***analyze*** *thirteen cohesion and one size metric* ***for the purpose of*** *evaluation,* ***with respect to*** *their ability to: (a) predict the existence of long method, and (b) prioritize the urgency for applying the extract method refactoring on them,* ***from the viewpoint of*** *software engineers,* ***in the context of*** *java open source software*". According to the aforementioned goal, we have derived two research questions that will guide the case study design and the reporting of the results.

RQ_1: *Which metrics can be used to predict the existence of the long method smell?*

This research question aims at identifying metrics that could potentially be used for predicting the existence of long methods in the complete codebase of software projects. In large codebases, the manual identification of long methods, might be a time consuming or even unrealistic task.

RQ_2: *Which metrics can be used for prioritizing long methods, with respect to their urgency for applying the extract method refactoring (in terms of extracted lines)?*

This research question aims at investigating which metrics could be used for prioritizing the identified long method smells, according to their urgency to get refactored. In large-scale software systems, it is likely that many methods could benefit from an extract method refactoring. However, applying all these refactoring opportunities is not feasible and maybe even unnecessary. Answering this research question can provide guidance on which of the existing long method bad smells should be initially refactored.

As urgency we define the average number of lines to be extracted by applying one extract method opportunity in the long method. We expect that the larger the methods to be extracted, the more important it is to refactor the long method. For example consider the two extract method opportunities of Figure 1 and the use of the *LCOM1* metric (see Table 1). For simplicity, in Figure 1, we denote sets of lines of code that are 100% cohesive (i.e., all lines all cohesive to each other), with the same fill pattern. Also, we consider that lines with different fill patterns are 100% non-cohesive (i.e., no variable is shared). In this case, *LCOM1* for the left method is 38, and we compare two extract method opportunities: (a) which extracts the block of 4 LoC, and (b) which extracts the block of 2 LoC. The outcome of (a) is method of LCOM1 equals 10, whereas the outcome of (b) is a method of *LCOM1* equals 20. Therefore, the benefit from extracting a larger number of cohesive lines of code is higher. Although in this example we describe an extreme scenario, the effect is similar in other cases.

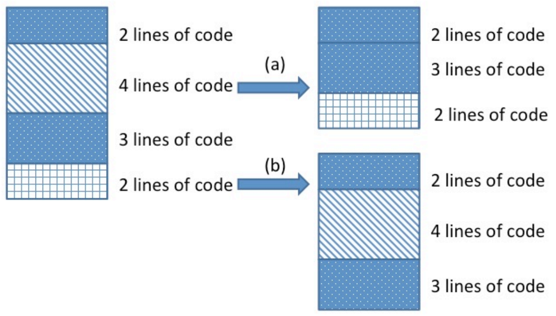

Figure 1. Extract Method Benefit

4.2 Case Selection and Units of Analysis

This study is a holistic multiple case study, in the sense that methods are both the cases and the units of analysis. As subjects for this study, we selected Java projects (listed in Table 2), based on our accessibility to their developers. In particular all selected projects are research tools for which we could ask one of their developers to indicate the existing long methods. Due to the effort required for manual long method detection, we investigated a rather small number of software projects, for which manual code inspection was feasible. The reason for restricting our case selection to Java projects was a limitation of the used tools for identifying extract method opportunities (see Section 4.3). Specifically, the tool that we used for identifying the extract method opportunities is able of parsing only Eclipse projects. On the completion of the process, we ended up with a dataset of four Java projects, which provided us with 1,850 methods.

Table 2. OSS Project Selection Outcome

Project	Project Description	#Methods
CKJM[2]	Calculates quality metrics for Java projects.	173
ClassInstability[3]	Calculates the REM metric for Java projects.	1,389
lm_tool[4]	Parses the abstract syntax tree of Java projects and identifies functionally related code chunks.	128
SSA[5]	Detects design pattern instances from Java binary classes.	160

4.3 Data Collection and Pre-Processing

The dataset used in this study consists of 1,850 rows, which correspond to methods of the selected Java projects. For every method, we recorded the following variables:

- **V1 – V3**: Method demographics (project name, class name, method name). This set of variables is not used in the analysis, but only for characterization purposes.
- **V4 – V16**: Cohesion metrics (see metrics described in Table 1). This set consists of the independent variables to be analyzed.
- **V17**: Method Size (uncommented lines of code inside the method). This variable is also used as an independent variable.
- **V18:** Long Method (yes / no). This variable was assigned a binary score from the developer of each project. This variable is going to be used as dependent variable in RQ_1.
- **V19**: Extract Method Urgency. This variable corresponds to the average number of lines to be extracted if the extract method refactoring is applied. The variable is extracted by using a tool (see below). This variable is going to be used as dependent variable in RQ_2.

The software metrics (V4 – V17) were calculated by a tool developed by the authors for the needs of this study[6], whereas, as mentioned earlier, variable V18 was manually recorded, based on experts' opinion (developers of the subject Java projects). The extract method opportunities (V19) were obtained using an existing tool, namely, JDeodorant [44]. JDeodorant is an Eclipse plugin that detects four types of refactoring opportunities, including extract method. The tool identifies refactoring opportunities by applying two different techniques for calculating static slices and uses the union of their results. The first technique calculates the complete computation slice of primitive data types or object references, which concerns a given variable, whose value is modified throughout the original method. The second technique calculates the object state slice, which consists of all statements modifying the state of a given object in the original method. For this purpose a set of slice-based metrics have been used (i.e., tightness, overlap, and coverage). These metrics are not directly related to any of the cohesion or size metrics used in our approach. Therefore, they do not affect the results of this case study.

Additionally, we need to clarify the basic difference between JDeodorant and our method is that they serve different goals: one identifying *long methods* and the other identifying *extract method opportunities*. Although the two goals are related, they differ in the sense that the existence of an extract method opportunity does not automatically constitute the method as 'long'. Details on the validation of JDeodorant are presented in Section 7. Finally, we note that we preferred to assess *urgency for refactoring* through the outcome of an automated tool, rather than expert opinion for two reasons: (a) the cohesion benefit obtained from extracting larger parts of code is an objective success criterion, and (b) the comparison of refactoring opportunities from different projects is not feasible in the sense that no developer had an overview of all examined projects. We preferred not to split the dataset into four sub-datasets (one dataset for each developer), as this would reduce the size of our sample, and consequently confidence in the obtained results.

On the completion of data collection, a pre-processing step took place. In particular, we filtered out of the dataset methods that were less prone to suffer from the long method bad smell. The rationale for this decision was to have a balanced dataset with respect to the number of methods that are in need of refactoring and those that are not. Having a balanced dataset makes the null model (i.e., a model without any independent variable) to provide a classification accuracy near 50% (i.e., close to the probability of a random guessing). According to King and Zeng [25], applying predictive models (e.g. regression) in rare events datasets (in our case 10%), can benefit from case selection strategies that reduce the number of negative events (in our cases methods that are not in need for refactoring). Therefore, we filtered out methods of size smaller than 30 lines of code, in alignment with Lippert and Roock [29], who suggest that a method is prone to suffer from bad smells if its size exceeds 30 lines of code. After applying this filter, the dataset was comprised of 79 units of analysis (including 40.5% of negative events). Although the number of cases seems rather small for a study in the domain of source code analysis, the number of cases is limited due to the involvement of human experts and the manual processing of source code.

4.4 Data Analysis

In order to answer the research questions set in Section 4.1, we will statistically analyze the collected data, through regression analysis, correlation analysis and visualization [17].

To answer RQ_1 we will investigate the ability of cohesion and size metrics (see Section 4.3) to act as potential predictors of the long methods. To this end, we will perform a logistic regression, which is used for predicting the value of a binary variable (in this case: V18 – *long method*), from a set of numerical predictors (in this case: *metric score* [V4–V17]). We note that although, some related work employs metrics combinations instead of using metrics in isolation, we believe that treating each metric separately is the first step towards creating a more complex model for the identification of long methods or extract method opportunities, in the sense that the most fitting metrics can be fed to such models. The generic form of a logistic regression equation is as follows:

$$f(metric\_score) = \frac{1}{1 + e^{-(b_0 + b_1 * metric\_score)}}$$

For each metric, the equation coefficients b_0 and b_1 will be calculated by performing the regression analysis [17]. Next, in order to use the regression equation the *metric score* has to be substituted, and the value of *f(metric_score)*, will assess the probability of the method to be in need of refactoring. Specifically, the closer the value of *f(metric_score)* is to 1.0, the larger the probability of the

2 http://www.spinellis.gr/sw/ckjm/
3 http://iwi.eldoc.ub.rug.nl/root/2014/ClassInstability/
4 www.cs.rug.nl/search/uploads/Resources/lm_tool.zip
5 http://java.uom.gr/~nikos/pattern-detection.html
6 www.cs.rug.nl/search/uploads/Resources/SEMI.zip

method to be long[7]. After creating the equations, the fitness of the models (i.e., the ability of each metric to predict the need for refactoring), will be assessed by three well-known measures: *accuracy*, *precision*, and *recall* [17]. Accuracy evaluates the ratio of correctly classified methods either positively or negatively (i.e., TP + TN) against all classified methods (n), precision quantifies the positive predictive power of the model (i.e., TP / (TP + FP), and recall evaluates the extent to which the model captures all long methods (i.e., TP / (TP + FN)[8].

To answer RQ_2 we will apply a correlation test between the cohesion/size metrics, and the average number of lines to be extracted, when a refactoring is applied. These tests will aim at identifying indicators on the urgency of refactoring a method, with respect to the average number of extracted lines of code. As explained above, it is expected that the benefit of extracting larger, cohesive methods should help reduce the negative effect of the long method smell. Even in cases that the smell is not totally mitigated (e.g., extraction of a relatively small code fragment, from a large method) the method is improved with respect to its long size.

The decision to apply a correlation test (i.e., Spearman correlation), is based on the 1061 IEEE Standard for Software Quality Metrics Methodology [1], which suggests that a sufficiently strong correlation "*determines whether a metric can accurately rank, by quality, a set of products or processes* (in the case of this study: a set of methods)". We note that we performed a Spearman rather than a Pearson correlation, since our data were not normally distributed and we were interested in ranking them. Additionally, in order to visualize the relations between the corresponding variables, and potentially mine underlying patterns, we will plot the dataset using scatter plots. Scatter plots are the default mean of visualization for exploring the correlation between two numerical variables [17]. A summary of data analysis techniques is presented in Table 3.

Table 3. Data Analysis Overview

Question	Variables	Statistical Analysis
RQ_1	Cohesion metrics Size Long method (yes / no)	Logistic Regression
RQ_2	Cohesion metrics Size Extract Method Urgency	Spearman Correlation Scatter-plots

5. RESULTS

In this section we present the results that have been obtained from data analysis, organized by research question. Due to space limitations and the public availability of the extracted dataset for data analysis replication, in both sections, we present only results that are statistically significant. Interpretation of the results and implications to researchers and practitioners are provided in Section 6.

5.1 Metrics for predicting the existence of long methods

To assess the ability of metrics to predict whether a method suffers from the long method smell (RQ_1), we present, in Table 4, the results of the corresponding regression analysis. Specifically we present two sets of measures: the first is related to the construction of the prediction model (i.e., *beta values* and *significance*), whereas the second is related to its evaluation (*accuracy*, *precision*, and *recall*). In the table we present only metrics that have a predictive power at a statistically significant level, i.e. lower or equal to 5%.

Table 4. Cohesion Metrics – Long Method (Predictive Power)

Metric	Prediction Model			Predictive Power		
	b0	b1	sig.	Accuracy	Precision	Recall
LOC	-4.491	0.103	0.00	84.8%	80.85%	92.68%
LCOM1	-1.281	0.002	0.00	79.7%	74.47%	89.74%
LCOM2	-0.809	0.002	0.00	70.9%	68.09%	80.00%
LCOM4	-0.536	0.113	0.01	68.4%	76.60%	72.00%
COH	1.392	-10.200	0.03	62.0%	89.36%	62.69%
CC	0.996	-5.362	0.05	68.4%	95.74%	66.18%

The results of Table 4 suggest that in total six metrics are able to predict which methods are in need for refactoring. We observe that *LOC* (i.e., lines of code), and three not normalized cohesion metrics (i.e., *LCOM1, LCOM2*, and *LCOM4*), form a group of measures significant at the 1% level. Finally, we can observe that two normalized cohesion metrics (*COH* and *CC*) are able to predict methods that are in need of extract method refactoring with a precision around 90%. As expected, these two metrics misclassify a larger number of false-negatives compared to the rest of the metrics, leading to a slightly decreased recall rate.

5.2 Metrics for long method prioritization

For answering RQ_2, we summarize the results of the Spearman correlation test in Table 5. Specifically, we present the ability of the examined metrics to rank methods, based on the average number of lines that will be extracted, if a proposed refactoring is applied. We note that the sign of the correlation depends on whether the metric expresses cohesion (e.g., *COH*), or lack of cohesion (e.g., *LCOM1*). Concerning *LOC*, the sign is positive, due to its direct relation to the number of extract method opportunities. From Table 5, we excluded metrics that: (a) were not significantly correlated to the corresponding variable at least at the 5% level, or (b) were correlated with strength lower than 0.2. According to Marg, correlations with $r < 0.2$ present weak or non-existing relations [32].

Table 5. Metrics – Average Lines to be Extracted

Metric	Correl. Coefficient	Sig.
LOC	0.463	0.00
LCOM1	0.472	0.00
LCOM2	0.385	0.00
LSCC	-0.326	0.00
COH	-0.303	0.00
CC	-0.268	0.02
DCD	-0.254	0.02

[7] The cut-off point has been set to 0.5 (default value in SPSS for binary values)

[8] TP: true positive, TN: true negative, FP: false positive, FN: false negative

Metric	Correl. Coefficient	Sig.
SCOM	-0.253	0.02
LCOM5	0.244	0.03

The results of the table suggest that the results are similar to those of RQ_1, in the sense that *LOC*, *LCOM1*, and *LCOM2*, are the top ranked indicators of the urgency for refactoring, followed by *COH* and *CC*. An additional finding from comparing the results of RQ_1 to those of RQ_2 is the fact that *LSCC, DCD, SCOM, and LCOM5* are valid indicators for the urgency for refactoring, but not for the existence of extract method opportunities. On the other hand, *LCOM4* is not able to rank long methods' urgency, despite the fact that it is a statistically significant predictor of their existence.

Finally, to visualize the aforementioned results, we produced scatter-plots for *LOC*, *LCOM1*, *LCOM2*, *LSCC*, and *COH* (top ranked indicators for refactoring urgency), and the average size of extract method opportunities. Based on Fig. 2, we have been able to verify the existence of a trend in our data, represented by a line. For example, concerning *LOC* (see top-left scatter plot), which is expected to have a positive correlation to the average number of extracted lines, we can observe an increasing trend.

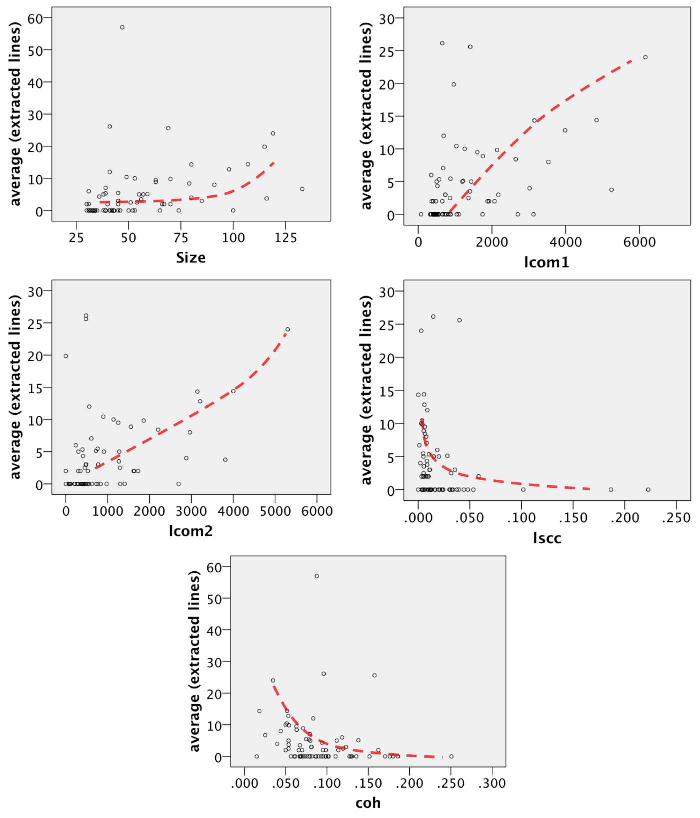

Figure 2. Visualization of Cohesion Metrics – Number of Extract Method Opportunities

6. DISCUSSION

In this section, we discuss the main findings of this case study, from two perspectives: (a) possible explanations for the obtained results (Section 6.1), and (b) implications for practitioners and researchers (Section 6.2).

6.1 Interpretation of results

Based on the results of this study, we argue that cohesion is a quality property that should be used for the identification of extract method opportunities, and subsequently for the mining of long method bad smell instances. This is a rather intuitive result, in the sense that cohesion (i.e., the functional relatedness of source code modules – in this case lines of code) has been already associated in the literature as an indicator of the number of distinct functionalities that the module offers [26], which can be extracted in a new method [18].

Compared to the ***existing approaches*** for long method or extract method identification, our cohesion-based approach presents the highest precision. Specifically, the precision of the cohesion metrics, proposed in this study, ranges from 68% to 96%, whereas related work reports 50% precision of complexity [33] and 38-66% precision of size metrics [13]. The precision of size, based on our results is 81%. We note that the calculation of recall for [13] and [33] was not possible because the relevant data were not provided. Based on our results *CC* and *COH* present the highest precision. However, we note that a safe comparison of the aforementioned findings can only be accomplished by applying all the approaches on a common dataset.

Comparing the ability of ***size and cohesion metrics*** to indicate if a method is in need of extract method refactoring, and therefore if it is long, the results cannot lead to safe conclusions, in the sense that 4 metrics (i.e., the size metric and 3 cohesion metrics) seem to outperform the rest, without large differences among them. However, we need to note that two of the top two cohesion metrics (i.e., *LCOM1* and *LCOM2*) are correlated to size (*LOC*), in the sense that they are open-ended metrics, whose upper limit is calculated as the count of combinations by two for the number of lines of code – the range of values for LCOM1 and LCOM2 is [0, $\binom{LOC}{2}$]. Nevertheless, regarding only precision two normalized cohesion metrics (i.e., *COH* and *CC*) are the optimal predictors. Therefore, if one is interested in capturing as many long methods as possible, one should prefer size or not normalized cohesion metrics, whereas if one is interested to get as fewer false positives as possible, then one should prefer normalized cohesion metrics.

Additionally, by ***comparing cohesion metrics***, we can identify four main groups of metrics:

- *LCOM1*, *LCOM2* which are the top ranked indicators for predicting and prioritizing extract method opportunities.
- *COH* and *CC,* which have the highest precision when used for predicting the existence of extract method opportunities, but are ranked lower than the first group, concerning prioritization.
- *LCOM4* which is a useful indicator only concerning identification of long method.
- *LCOM5, LSCC, DCD,* and *SCOM*, which are useful indicators only concerning the urgency for refactoring a long method.
- *LCOM3, TCC, LCC,* and *DCI*, which appear to be unable to indicate either the existence, or the urgency of applying the extract method refactoring.

From the aforementioned findings, we can highlight the following observations:

- The metrics that quantify ***lack of cohesion, instead of cohesion*** (i.e., *LCOM1, LCOM2,* and *LCOM4*), appear to have ***higher predictive power*** concerning extract method opportunities compared to the rest of the metrics. This result is especially interesting since *LCOM1* and *LCOM2* have until now received much criticism in the literature (e.g., [8], [9], [16]). However, the results of this case study suggest that the fact that they are open-ended and that they are to some extent related to method size, provides them with a comparative advantage for the identification and prioritization of methods that are in need of refactoring.

- The metrics that ***involve method invocations*** in their calculation (i.e., *LCOM4, TCC, LCC, DCD,* and *DCI*) appear to provide ***lower ranking power***, than those that omit them. Thus, we assume that the semantic distance between two lines is not related to whether they call the same method, but only to whether they access common variables.
- The metrics *SCOM* and *CC*, that are similar in their calculation, appear to produce similar results in terms of ranking and predictive power[9].
- Although *COH* is calculated as a function of *LCOM5*, it provides better results with respect to every research question, implying that the suggested normalization by Briand et al. (which drops the assumption that each attribute is referenced by at least one method) is more fitting than the original one [9].
- Similarly to O' Cinneide et al. [37], we suggest that the different aspects of cohesion that various metrics quantify, lead to different predictive and ranking power. However, as discussed before, some grouping for closely related metrics (e.g., *LCOM1* and *LCOM2*, and *SCOM* and *CC*) is possible.

6.2 Implications to researchers & practitioners

Implications to practitioners. Based on our results we suggest that software engineers can:

- Include ***method-level cohesion metrics*** in their quality monitoring process, especially in cases that bad smell detection tool support is not available for their programming environments.
- ***Prioritize manual method code inspections***, with respect to refactoring identification, based on the *COH*, *LCOM1* and *LCOM2* metrics, in the sense that they are strongly correlated to the urgency to apply extract method opportunities.

Implications to researchers. Similarly, based on our results we propose that researchers:

- Develop approaches that aim at the identification of extract method opportunities, ***based on method-level cohesion metrics.***
- Explore the potential of ***additional size metrics*** (e.g., number of accessible variables in a method) to indicate the existence and prioritization of extract method opportunities.
- Explore the potentially improved predictive and ranking power of approaches that ***combine size and cohesion metrics*** (e.g. by using multivariate regression models, multi-criteria methods like the analytic hierarchy process (AHP), or Bayesian networks).
- Investigate the possibility of ***identifying thresholds,*** for the six metrics presenting the highest predictive power, that when surpassed a method can be classified as in need for extract method refactoring.
- Investigate if method-level cohesion metrics can be used for the development of ***feature identification algorithms***. The inherent relation between lack of cohesion and the number of functionalities that offers a software module might lead to a promising way for exploring the field of feature extraction.

7. THREATS TO VALIDITY

In this section we will present potential threats to validity for our study following the guidelines proposed by Runeson et al. [39]. According to Runeson et al., there are four types of threats to validity: construct, reliability, external, and internal validity threats. Specifically, *construct validity* concerns the degree to which the study answers explicitly the research questions, alleviating any doubts with regard to the appropriateness of the used methods. *Reliability* concerns the capability of reproducing the study and getting the same results. *External validity* deals with any potential limitations that would prevent or threaten the general application of the proposed method or the derived results. Finally, *internal validity* concerns the investigation of whether a causal conclusion is warranted. In this study internal validity will not be considered, since causal relations are not in the scope of this study.

Construct Validity. Regarding construct validity, the first two threats concern the use of the two automated tools for creating our dataset. JDeodorant is a third-party tool, which was used for counting extract method opportunities detected in the studied methods. The tool is considered as trustworthy due to the extensive validation process and provided evidence [44]. Specifically according to the authors the tool has been evaluated in two different ways: (a) using a well-known open source project for assessing the soundness and usefulness of the identified refactoring opportunities, and also investigating their impact on the slice-based cohesion metrics and the external behavior of the program, and (b) by comparing the results of their tool to those of independent evaluators, on projects developed by themselves. The second tool, which was used for calculating the metrics, is a custom-made tool developed by the authors of this paper for the needs of the study. To mitigate the potential threat of miscalculating the metrics, we verified the tool by manually calculating the metrics for random units of analysis and crosschecked them with those of the tool; we found no inconsistences. Finally, a third potential threat to construct validity is the applicability of the selected metrics to measure method cohesion. However, as described in Section 3, the proposed metrics are produced by transforming well-established class-level cohesion metrics to method-level metrics. By taking into account that we used an exact transformation, without omitting any rules or making further assumptions, we believe that the method-level metrics retain their ability for assessing cohesion.

Reliability. To mitigate threats to reliability, two different researchers conducted the data collection and data analysis process, to double-check the derived results. All raw data can be reproduced by following the process described in Section 4, and by using the mentioned tools. A threat to reliability is that for each project we used only one external evaluator. To mitigate this threat we selected to involve the lead developer of each project.

[9] In terms of predictive power, SCOM presents the following results: precision 93.6%, recall 62.7% (sig: 0.13). The results are not presented in Table 5, since they are not statistically significant (sig > 0.05)

External Validity. With regard to external validity there are three potential threats. First, the study was conducted analyzing only Java projects and thus the results cannot be directly applied to other languages. Second, the subjects of our case study are only research projects, so we cannot generalize the applicability of results to industrial source code. Third, the study focused on 14 specific metrics and the results cannot be generalized to other metrics quantifying the same or different quality properties.

8. CONCLUSIONS

Extract method refactoring aims at alleviating the negative impact of one of the most common and persistent code bad smells, i.e., the *long method*. Although, tools and approaches on the identification of long methods and extract method opportunities exist in the literature, their applicability is limited, and none of them deal with the prioritization of methods in need for refactoring; whereas *existing metric-based approaches* present a relatively *low accuracy*. Therefore, in this study, we explored the potential ability of ***cohesion*** and ***size metrics*** to: (a) ***predict*** which methods are long and which are not, and (b) ***rank*** methods, with respect to their urgency of applying ***extract method refactorings***.

The results of the study suggested that the used size metric (i.e., *LOC*) and three cohesion metrics (i.e., *LCOM1*, *LCOM2*, and *COH*) *are the most prominent metrics with respect to their predictive and ranking power*. The study results confirms the intuitive connection among cohesion and the long method bad smell, unveils a significant dimension on the possible use of the *LCOM1* and *LCOM2* metrics that have until now received criticism in the literature, and point out several implications for researchers and practitioners.

ACKNOWLEDGMENT

This research has been partially funded by the ITEA2 project 11013 PROMES.

REFERENCES

[1] 1061-1998: IEEE Standard for a Software Quality Metrics Methodology, *IEEE Standards*, IEEE Computer Society, 31 December 1998 (re-affirmed 9 December 2009).

[2] Al Dallal, J. 2011. Measuring the Discriminative Power of Object-Oriented Class Cohesion Metrics. *Transactions on Software Engineering*. IEEE Computer Society, 37, 6 (November/December 2011), 788 – 804.

[3] Al Dallal J., and Briand L. 2012. A Precise method-method interaction-based cohesion metric for object-oriented classes. *Transactions on Software Engineering and Methodology*. ACM Press, 23, 2 (March 2012), art. 8.

[4] Badri L., and Badri M. 2004. A Proposal of a new class cohesion criterion: an empirical study. *Journal of Object Technology*. 3, 4 (April 2004), 145-159.

[5] Bansiya J., and Davis C. 2002. A hierarchical model for object-oriented design quality assessment. *Transaction on Software Engineering*. IEEE Computer Society, 28, 1 (January 2002), 4–17.

[6] Basili V., Caldiera G., and Rombach D. 1994. The Goal Question Metric Approach. *Encyclopedia of Software Engineering*. John Wiley & Sons, 528-532.

[7] Bieman J. M., and Kang B. 1995. Cohesion and reuse in an object-oriented system. *Proceedings of the 1st Symposium on Software Reusability* (Seattle, USA, 29 – 30 April 1995). SSR'95. ACM Press, 259-262.

[8] Bonja C., and Kidanmariam E. 2006. Metrics for class cohesion and similarity between methods. *Proceedings of the 44th Annual Southeast Regional Conference* (Melbourne, Florida, 10-12 March 2006). ACMSE'06. ACM Press, 91-95.

[9] Briand L. C., Daly J., and Wuest J. 1998. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering* (March 1998). Springer, 3, 1, 65-117.

[10] Chatzigeorgiou A., and Manakos A. 2014. Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering* (March 2014). Springer, 10, 1, 3-18.

[11] Chidamber S. R., and Kemerer C. F. 1991. Towards a metrics suite for object oriented design. *Proceedings of the 6th Conference on Object-Oriented Programming Systems Languages, and Applications* (Phoenix, Arizona, 6-11 October, 1991). OOPSLA'91. ACM Press, 197-211.

[12] Chidamber S. R., and Kemerer C. F. 1994. A metrics suite for object oriented design. *Transactions on Software Engineering* (June 1994). IEEE Computer Society, 20, 6, 476-493.

[13] Demeyer S., Ducasse S., and Nierstrasz O. 2000. Finding refactorings via change metrics. *Proceedings of the 15th Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, USA, 15-19 October 2000). OOPSLA'00. ACM Press, 166-177.

[14] Dexun J., Peijun M., Xiaohong S., and Tiantian W. 2012. Detecting Bad Smells with Weight Based Distance Metrics Theory. *Proceedings of the 2nd International Conference on Instrumentation, Measurement, Computer, Communication and Control* (8-10 December 2012). IMCCC'12. IEEE Computer Society, 299-304.

[15] Fenton N. E., and Pfleeger S. L. 1998. Software Metrics: A Rigorous and Practical Approach (2nd ed.). *PWS Pub. Co*.

[16] Fernández L., and Peña R. 2006. A sensitive metric of class cohesion. *International Journal of Information Theories and Applications* (January 2006). IJ ITA, 13, 1, 82-91.

[17] Field A. 2013. Discovering Statistics using IBM SPSS Statistics. *SAGE Publications Ltd*.

[18] Fowler M., Beck K., Brant J., Opdyke W., and Roberts D. 1999. Refactoring: Improving the Design of Existing Code (1st ed.). *Addison-Wesley Professional*.

[19] Henderson-Sellers B. 1996. Object-Oriented Metrics Measures of Complexity. *Prentice-Hall*.

[20] Hitz M., and Montazeri B. 1995. Measuring coupling and cohesion in object oriented systems. *Proceedings of the International Symposium on Applied Corporate Computing* (Monterrey, Mexico, 25-27 October 1995). ISACC'95. 25-27.

[21] Joshi P., and Joshi R. K. 2009. Concept Analysis for Class Cohesion. *Proceedings of the 13th European Conference on Software Maintenance and Reengineering* (Kaiserslautern, Germany, 24-27 March 2009). CSMR'09. IEEE Computer Society, 237-240.

[22] Kataoka Y., Imai T., Andou H., and Fukaya T. 2002. A quantitative evaluation of maintainability enhancement by refactoring. *Proceedings of the 18th International Conference on Software Maintenance* (Montréal, Canada, 3-6 October

2002). ICSM'02. IEEE Computer Society, 576-585.

[23] Khomh F., Di Penta M., and Guéhéneuc Y.-G. 2009. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. *Proceedings of the 16th Working Conference on Reverse Engineering* (Lille, France,13-16 October 2009). WCRE'09. IEEE Computer Society, 75-84.

[24] Khomh F., Vaucher S., and Guéhéneuc Y.-G. 2009. A Bayesian Approach for the Detection of Code and Design Smells, *Proceedings of the 9th International Conference on Quality Software* (Jeju, South Korea, 24-25 August 2009). QSIC'09. IEEE Computer Society, 305-314.

[25] King G., and Zeng L. 2001. Logistic Regression in Rare Events Data Political Analysis, *Oxford Journal*, 9, 2 *(*March 2001), 137-163.

[26] Laird L. M., and Brennan M. C. 2006. Software Measurement and Estimation: A Practical Approach. *Wiley-IEEE Computer Society Press*.

[27] Lanza M., and Marinescu R. 2010. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. *Springer*.

[28] Li W., and Henry S. M. 1993. Maintenance metrics for the object oriented paradigm. *Proceedings of the the 1st International Symposium on Software Metrics* (Baltimore, MD, 21-22 May 1993). METRICS'93. IEEE Computer Society, 52-60.

[29] Lippert M., and Roock S. 2006. Refactoring in Large Software Projects, 1st edition. *Wiley & Sons*.

[30] Mäntylä M. V., Vanhanen J., and Lassenius C. 2004. Bad smells - humans as code critics. *Proceedings of the 20th International Conference on Software Maintenance* (Chicago, USA , 11-14 Sept. 2004). ICSM'04. IEEE Computer Society, 399-408.

[31] De Marco T. 1979. Structured Analysis and System Specification. *Yourdon Press Computing Series*.

[32] Marg L., Luri L. C., O'Curran E. and Mallett A. 2014. Rating Evaluation Methods through Correlation. *Proceedings of the 1st Workshop on Automatic and Manual Metrics for Operational Translation Evaluation* (Reykjavik, Iceland, 26 May 2014). MTE'14.

[33] Marinescu R. 2004. Detection strategies: metrics-based rules for detecting design flaws. *Proceedings of the 20th International Conference on Software Maintenance* (Chicago, USA, 11-14 September 2004). ICSM'04. IEEE Computer Society, 350-359.

[34] Meananeatra P., Rongviriyapanish S., and Apiwattanapong T. 2011. Using software metrics to select refactoring for long method bad smell. *Proceedings of the 8th International Conference on Electrical Engineering, Electronics, Computer, Telecommunications and Information Technology* (Khon Kaen, Thailand, 17-19 May 2011). ECTI-CON'11. IEEE Computer Society, 492-495.

[35] Mihancea P. F., and Marinescu R. 2005. Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems. *Proceedings of the 9th European Conference on Software Maintenance and Reengineering* (Manchester, UK, 21-23 March 2005). CSMR'05. IEEE Computer Society, 92-101.

[36] Moha N., Guéhéneuc Y.- G., Duchien L., and Le Meur A. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *Transactions on Software Engineering* (January/February 2010), IEEE Computer Society, 36, 1, 20-36.

[37] Ó Cinnéide M., Tratt L., Harman M., Counsell S., and Moghadam I. H. 2012. Experimental assessment of software metrics using automated refactoring. *Proceedings of the 6th International symposium on Empirical software engineering and measurement* (Lund, Sweden, 19 -20 September). ESEM '12. ACM/IEEE.

[38] Palomba F., Bavota G., Di Penta M., Oliveto R., De Lucia A., and Poshyvanyk D. 2013. Detecting bad smells in source code using change history information. *Proceedings of the 28th International Conference on Automated Software Engineering* (11-15 November 2013). ASE'13. IEEE Computer Society, 268-278.

[39] Runeson P., Höst M., Rainer A., and Regnell B. 2012. Case Study Research in Software Engineering: Guidelines and Examples. *John Wiley and Sons*, Inc.

[40] Salehie M., Shimin L., and Tahvildari L. 2006. A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. *Proceedings of the 14th International Conference on Program Comprehension* (Athens, Greece, 14-16 June 2006). ICPC'06. IEEE Computer Society, 159-168.

[41] Silva D., Terra R., Valente M. T. 2014. Recommending automated extract method refactorings. *Proceedings of the 22nd International Conference on Program Comprehension.* ICPC 2014. ACM Press,146-156.

[42] Simon F., Steinbruckner F., and Lewerentz C. 2001. Metrics based refactoring. *Proceedings of the 5th European Conference on Software Maintenance and Reengineering* (Lisbon, Portugal, 14-16 March 2001). CSMR'01. IEEE Computer Society, 30, 38.

[43] Sjoberg D. I. K., Yamashita A., Anda B. C. D., Mockus A., Dyba T. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *Transactions on Software Engineering*. IEEE Computer Society, 39, 8 (August 2013), 1144-1156.

[44] Tsantalis N., Chatzigeorgiou A. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84, 10 (October 2011), 1757-1782.

[45] Yoshida N., Masataka K., and Hajimu I. 2012. A cohesion metric approach to dividing source code into functional segments to improve maintainability. *Proceedings of the 16th European Conference on Software Maintenance and Reengineering* (Szeged, Hungary, 27-30 March 2012). CSMR'12. IEEE Computer Society, 365-370.

[46] Zhao L., and Hayes J. 2006. Predicting classes in need of refactoring: an application of static metrics. *Proceedings of the 1st Workshop on Predictive Models of Software Engineering* (Philadelphia, USA, 23 September 2006). PROMISE'06.