Methodbook: Recommending Move Method Refactorings via Relational Topic Models

Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, Andrea De Lucia

Abstract—During software maintenance and evolution the internal structure of the software system undergoes continuous changes. These modifications drift the source code away from its original design, thus deteriorating its quality, including cohesion and coupling of classes. Several refactoring methods have been proposed to overcome this problem. In this paper we propose a novel technique to identify *Move Method* refactoring opportunities and remove the *Feature Envy* bad smell from source code. Our approach, coined as Methodbook, is based on Relational Topic Models (RTM), a probabilistic technique for representing and modeling topics, documents (in our case methods) and known relationships among these. Methodbook uses RTM to analyze both structural and textual information gleaned from software to better support move method refactoring. We evaluated Methodbook in two case studies. The first study has been executed on six software systems to analyze if the move method operations suggested by Methodbook help to improve the design quality of the systems as captured by quality metrics. The second study has been conducted with eighty developers that evaluated the refactoring recommendations produced by Methodbook. The achieved results indicate that Methodbook provides accurate and meaningful recommendations for move method refactoring operations.

Index Terms—Refactoring; Relational Topic Models; Empirical Studies.

1 Introduction

One of the main goals of software development is to tackle the complexity of software. In Object-Oriented (OO) software, classes are the primary decomposition mechanism, which group together data and operations to reduce complexity. Researchers defined coupling and cohesion as properties of software modules, like class and packages. In particular, coupling has been defined as *the degree to which each module relies on each one of the other modules* while cohesion is *the degree to which the elements of a module belong together* [1]. Generally, accepted rules state that modules should have high cohesion and low coupling [1], [2], [3]. In fact, several empirical studies provided evidence that high levels of coupling and/or lack of cohesion are generally associated with lower productivity, greater rework, and more significant design efforts by developers (e.g., [4], [5]). In addition, classes with low cohesion and/or high coupling have been shown to correlate with high defect rates [6].

Software systems inevitably evolve during their life-cycle to meet ever-changing users' needs and adapt to changes in their environment. During evolution, the structural design of the software system changes and the changing forces mostly results in a deterioration of the software structure which also exhibits worse values of cohesion and coupling. Indeed, software evolution is often an unstructured process during which the developers' needs to reduce time to market may lead to design erosion and the introduction of poor design solutions, usually referred to as *code bad smells* [7], [8].

A classic bad small is the *Feature Envy*, arising when a method seems to be more interested in a class other than the one it is implemented in [8]. For example, instances of this smell are present when a method invokes many times methods of another class (i.e., the envied class) or, more in general, when the responsibilities it implements are more similar to those grouped in the envied class than to those of the class it is implemented in. This clearly results in reduced class cohesion and increased coupling between classes. Thus, it is important to identify and remove the *Feature Envy* bad smell whenever instances are found in a software system. In order to remove such a bad smell, a *Move Method* refactoring operation is required, i.e., the method is moved to the envied class. Unfortunately, not all the cases are cut-and-dried. Often a method uses features of several classes, thus the identification of the envied class (as well as the method to be moved) is not always trivial [9].

These considerations highlight the need for auto-

This paper is an extension of "Identifying Method Friendships to Remove the Feature Envy Bad Smell" that appeared in the Proceedings of the 33rd International Conference on Software Engineering (NIER Track)*, Waikiki, Honolulu, Hawaii, pages 820-823, 2011.*

- *G. Bavota, University of Sannio, Benevento, Italy. E-mail: gbavota@unisannio.it.*
- *R. Oliveto, University of Molise, Pesche (IS), Italy. E-mail: rocco.oliveto@unimol.it.*
- *M. Gethers, University of Maryland, Baltimore County, Baltimore, MD 21250, USA. E-mail: mgethers@umbc.edu.*
- *D. Poshyvanyk, The College of William and Mary, Williamsburg, VA 23185, USA. E-mail: denys@cs.wm.edu.*
- *A. De Lucia, University of Salerno, Fisciano (SA), Italy. E-mail: adelucia@unisa.it*

mated support to assist developers in making adequate decisions while analyzing constantly changing structural and textual information in evolving software. In this paper, we present an approach to identify Move Method refactoring opportunities aimed at solving Feature Envy bad smell. The proposed approach, named Methodbook, follows the Facebook[1] metaphor. Facebook is a well-known social networking portal, where users can add people as friends, send messages, and update personal profiles to notify friends about their status. The personal profile plays a crucial role there. In particular, Facebook analyzes users' profiles and suggests new friends or groups of people sharing similar interests.

In our implementation of Methodbook, methods and classes play the same role as people and groups of people, respectively, in Facebook; methods' implementations, that is profiles, contain information about structural (e.g., method calls) and conceptual (i.e., textual) relationships (e.g., similar identifiers and comments) with other methods in the same class and in the other classes. Then, Methodbook uses Relational Topic Model (RTM) [10] to identify "friends" of a method in order to suggest move method refactoring opportunities in software. In particular, given a method, we exploit RTM to suggest as a target class the one containing the highest number of "friends" of the method under analysis. Note that, differently from the approaches existing in the literature [9], [11], [12], Methodbook also exploits textual information present in the source code to capture relationships between methods. This results in the the possibility for Methodbook to identify *Feature Envy* instances that are ignored by the approaches that exist in the literature (e.g., a method having the same amount of structural dependencies with methods in its class and with methods in the envied class, but a much higher textual similarity with methods in the envied class).

In this paper we evaluate the usefulness of Methodbook in two case studies. In the first study we evaluated Methodbook on six software systems through well-established metrics that capture the quality improvement achieved while applying the proposed refactoring operations. In the second study, we evaluated Methodbook's refactoring recommendations with developers' opinions in two case studies, one conducted with ten original developers of two software systems and one with seventy academic and industrial software developers on two open source software systems.

The rest of the paper is organized as follows. Section II discusses the related work, while Methodbook is overviewed in Section III. Section IV reports the first case study where Methodbook has been evaluated via quality metrics, while Section V reports the results of the study with users. Section VI discusses the threats that could affect the validity of our findings while concluding remarks are given in Section VII.

1. http://www.facebook.com/

2 Related Work

Refactoring has been faced in literature from different perspectives. Some authors have shed light on how developers perform refactoring [13], [14], [15] while others have proposed techniques and tools to better integrate refactoring in the software lifecycle [16], [17]. However, most of the work in the field is related to techniques dealing with (semi-) automatic improvement of the design of a software system. In particular, authors have focused their attention on the identification of design problems that may represent refactoring opportunities [18], [19], [20], [21], re-modularization techniques [22], [23], [24], [25], [26], and refactoring approaches [9], [11], [27], [12], [28], [29], [30], [31], [32], [33], [34], [35]. Our approach is mostly related to the latter ones. We briefly discuss approaches in the literature supporting refactoring operations different than move method and then focus the attention on the move method refactoring techniques.

Approaches recommending extract class refactoring solutions have been presented in the literature [28], [29], [30], [35]. Fokaefs *et al.* [28] use a clustering algorithm to perform extract class refactoring. Their approach analyzes structural dependencies among the entities of a class to be refactored, i.e., attributes and methods. Using this information, they compute the *entity set* for each attribute, i.e., the set of methods using it, and for each method, i.e., all the methods that are invoked by a method and all the attributes that are accessed by it. Thus, the Jaccard distance between all the couples of entity sets of the class is computed in order to cluster together cohesive groups of entities that can be extracted as separate classes. A hierarchical clustering algorithm is used to that aim. Note that Methodbook aims at supporting a different refactoring operation than the approach by Fokaefs *et al.* [28] and exploits textual information extracted from source code, in addition to structural information.

Bavota *et al.* [29], [35] proposed two approaches that support extract class refactoring based on graph theory. Both approaches represent a class to be refactored as a weighted graph in which each node represents a method of the class and the weight of an edge that connects two nodes (methods) represents the structural and semantic similarity of the two methods. In both the approaches, the similarity matrix representing the graph is first filtered to remove spurious relations between methods. A MaxFlow-MinCut algorithm is used to split the graph in two subgraphs to obtain two sets of methods that should be placed in the same class [29]. This approach always splits the class to be refactored in two classes. The approach has been extended aiming at splitting a class in more classes [35]: the transitive closure of the incident matrix is computed to identify sets of methods representing the new classes to be extracted. Extract class refactoring operations have been also identified by using game theory [30]. In particular, the extract class refactoring process is modeled as a non-cooperative game between two players, each one in charge to build a new class starting from the methods of the class to be refactored. The results of a preliminary evaluation show that game theory might be a good way to support refactoring operations. The approaches described above are the closest to Methodbook with respect to the exploited information extracted from source code (i.e., both structural and textual information). However, all these approaches [29], [35], [30] aim at solving a different problem (i.e., the decomposition of large classes) than the one tackled by Methodbook.

O'Keeffe *et al.* [32] formulate the refactoring task as a search problem in the space of alternative designs. The alternative designs are generated by applying a set of refactoring operations, e.g., push up field, pull down method, collapse hierarchy while the search from the optimal design is guided by a quality evaluation function based on eleven object-oriented design metrics, i.e., the Chidamber and Kemerer (CK) metrics [36] that reflect refactoring goals. Note that move method refactoring is not supported by this approach. Moreover, in the approach by O'Keeffe *et al.* [32] only structural metrics are used while Methodbook also exploits textual information embedded in the code.

Abadi *et al.* [33] propose the use of fine slicing to support the Extract Method refactoring. The fine slicing is used to extract executable program slices from their surrounding code and encapsulate them in different methods. This refactoring has also been the object of the work by Murphy-Hill and Black [34], that describe a set of characteristics desirable in refactoring tools supporting extract method refactoring. Maruyama *et al.* [27] present a mechanism to improve the reusability of frameworks. In particular, their approach automatically refactors methods in Object-Oriented frameworks by using weighted dependence graphs, whose edges are weighted based on the modification histories of the methods. The assumption is that programmers will reuse and modify code of their frameworks in the future in the same way that they often did in the past. Note that while both the approach by Abadi *et al.* [33] and by Maruyama *et al.* [27] support refactoring at method level (as Methodbook), they (i) do not exploit textual information extracted from source code and (ii) are not aimed at removing Feature Envy bad smells.

To the best of our knowledge, only three approaches exist in literature to automate move method refactoring [9], [11], [12]. Simon *et al.* [11] provide a metric-based visualization tool useful to identify, among others, Move Method refactoring opportunities. In particular, each method is analyzed to verify which are its structural relationships (i.e., method calls and attribute accesses) with the classes of the system. If there is a class having more dependencies with the method as compared to the class the method belongs to, a possible Feature Envy bad smell is identified. Examples of application of the proposed tool showed its potential usefulness in integrated development environments. However, an evaluation of the approach on real software systems has not been performed. Unlike Methodbook, the approach by Simon *et al.* just exploits structural information.

Seng *et al.* [12] use a genetic algorithm to suggest move method refactoring operations. The fitness function used to guide the identification of the refactoring opportunities is defined as a combination of structural metrics able to capture the quality of the system classes. The evaluation performed on an open source software system showed that their approach is able to improve the value of some quality metrics measuring class cohesion and coupling. Moreover, the authors manually inspected the proposed move method refactoring operations finding all of them justifiable.

Note that Methodbook, as compared to the approach presented by Seng *et al.* [12], also takes into account textual information embedded in the source code, such as terms present in the comments and identifiers of source code classes. This information can be exploited to measure the lexical similarity between a method and the classes of the system. The conjecture is that the higher the overlap of terms between comments and identifiers of a method m_i and a class C_j, the higher the likelihood that they implement similar responsibilities (and thus the class C_j might be a good candidate as an envied class for the method m_i).

Another approach to automate move method refactoring has been proposed by Tsantalis and Chatzigeorgiou [9]. In particular, for each method of the system, their approach forms a set of candidate target classes where a method should be moved to. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes. It is worth noting that unlike Methodbook, the approach by Tsantalis and Chatzigeorgiou [9] only exploits structural information extracted from the source code to identify the envied class for a method under analysis. The approach presented by Tsantalis and Chatzigeorgiou [9] has been evaluated (i) analyzing its capability to suggest move method refactoring operations that improve design quality (in terms of class cohesion and coupling) of two open source software system, (ii) asking an independent designer to analyze and comment the refactorings proposed for a small application (34 classes) she developed, and (iii) evaluating the efficiency of the proposed algorithm in terms of running time. The achieved results showed that while applying the proposed refactorings (i) it is possible to achieve a decrease of the average class cou-

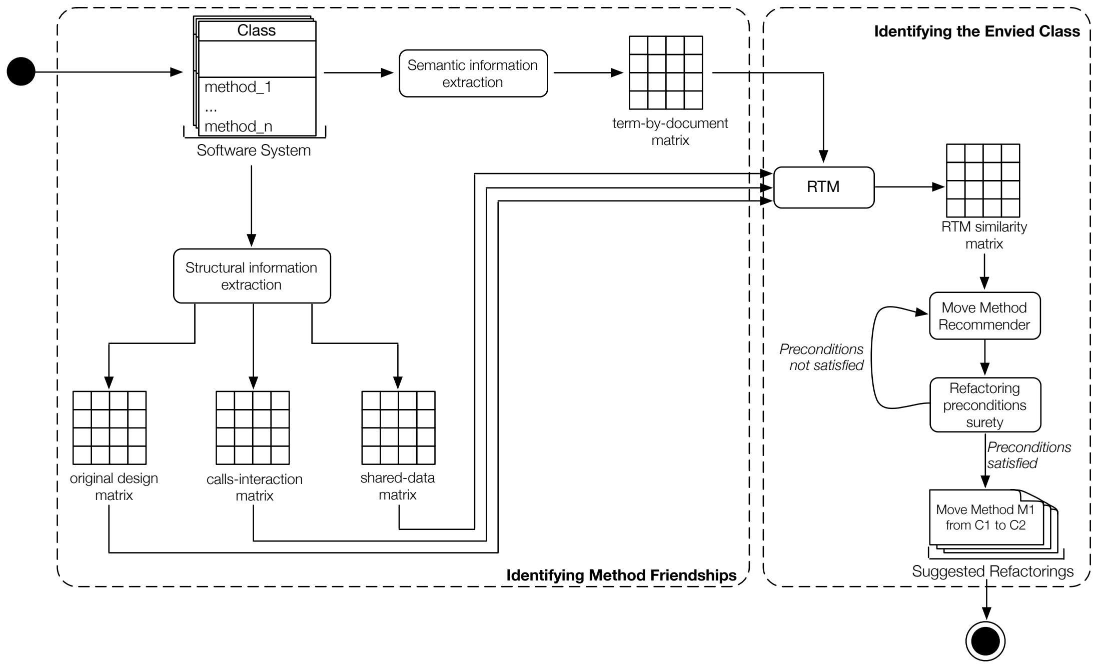


Fig. 1: The process used by Methodbook to identify move method refactoring operations

pling of the systems of about -1.25% and an increase of the average class cohesion of about +3.0%, (ii) the 80% (8 out of 10) of the refactoring operations proposed by the approach made sense from the point of view of the designer involved in the experimentation, and (iii) the time needed by the approach to find refactoring operations went from 7 to 137 seconds, depending on the system's size and on the number of operations identified. The approach has also been implemented as an Eclipse plug-in, coined as JDeodorant[2].

Note that, in the work by Tsantalis and Chatzigeorgiou [9] the authors thoroughly explain why their approach should be preferred to that one proposed by Seng *et al.* [12]. Among the most important weaknesses identified for the approach proposed by Seng *et al.* [12] it is worth to recall the following:

- *it uses genetic algorithms making random choices on mutation and crossover operations*. Thus, the outcome of each execution on the same system may differ. To overcome such an issue the authors propose to run the algorithm several times (ten in the example reported in the paper) and only suggest the move method operations common to all the performed executions. This will clearly negatively affect the efficiency of the proposed technique, especially on large software systems. Moreover, the results reported in the paper are not statistically significant due to the small number of runs.
- *it requires a long calibration procedure*. In particular, a calibration run for each metric exploited in the fitness function is necessary.

Thus, even though the approach by Seng *et al.* [12] might be valid from a theoretical point of view, its practical application would be quite hard. For this reason, in the context of our case studies we compared Methodbook with JDeodorant (see Sections IV and V).

3 Methodbook

In a nutshell, Methodbook works as depicted in Figure 1. The process used by Methodbook to identify move method refactoring operations is composed of two main steps: (i) identification of methods' friendships[3], and (ii) identification of the envied class.

In the first step, textual and structural information is extracted from the source code. The textual information is represented by words in comments and identifiers in source code and is stored in the *term-by-document* matrix. This matrix is used by RTM[4] to derive semantic relationships between methods and define a probability distribution of topics (topic distribution model) among methods. Besides textual information, Methodbook also exploits static analysis to derive (i) structural dependencies among methods (i.e., method calls stored in the *calls-interaction matrix* and shared instance variables stored in the *shared-data matrix*) and (ii) the original design, i.e., which methods are contained in each class of the system, stored in the *original design matrix*. The structural matrices are used to adjust the topic probability distribution taking into account structural relationships between

2. http://jdeodorant.com verified on 08/16/2013

3. The concept of method friendship exploited in the proposed approach is different from the concept of friend classes/methods in C++.

4. The implementation of RTM used in this study was developed by Chang and Blei [10] and can be download at http://cran.r-project.org/web/packages/lda/.

methods, besides textual information. In particular, the *calls-interaction matrix* and the *shared-data matrix* represent the two main forms of interaction among the methods of a system, i.e., calls interaction and shared instance variables, while the aim of the *original design matrix* is to take into account the design decisions made by the developers. Providing RTM with the original design information enables it to suggest move method refactoring operations only if they result in a clear improvement of the overall design quality.

The model derived by RTM is then used to compute "friendships" among methods based on both probabilistic distributions of latent topics and underlying structural dependencies. The friendship relationships among all the pairs of methods of the system are stored in the *RTM similarity matrix* (see Figure 1). In the context of our approach two methods are considered to be friends if they share responsibilities, i.e., they operate on the same data structures or are related to the same features or concepts in the program. Such a definition suggests that methods that are good friends should be in the same class, since *"a class should be a crisp abstraction, handle a few clear responsibilities, or some similar guideline"* [8].

Based on this definition, if the "best" friends of a method m implemented in C_m are in a class C_f, then m shares more responsibilities with the methods of class C_f than with those in C_m. We conjecture that such a scenario implies the presence of a Feature Envy bad smell with the class C_f being an envied class. For this reason, in the second step, Methodbook identifies the envied class as the one containing the highest percentage of best friends of the method m_i (i.e., methods having high similarity with m_i). If the envied class coincides with the original class, Methodbook does not suggest any refactoring operation. Otherwise, Methodbook performs static code analysis to verify if a set of preconditions ensuring the preservation of the system behavior post-refactoring are satisfied for the suggested envied class (e.g., Methodbook ensures that the envied class does not contain a method having the same signature as the method to be moved). If the preconditions are satisfied, the refactoring is suggested, otherwise the second class containing the higher percentage of top friends for the method under analysis becomes the new candidate envied class and thus, is object of the preconditions verification. This process is repeated until (i) an envied class satisfying the refactoring preconditions is identified or (ii) the envied class coincides with the original class (and thus no refactoring is suggested). It is worth noting that Methodbook is fully automated since it can analyze all the system's methods to identify move method refactoring operations. Moreover, Methodbook can also be applied only to a particular method provided by the developer as an input (i.e., a method identified as suffering of the Feature Envy bad smell).

In the next subsections we first provide some details about RTM and then we provide in-depth explanation for the two steps behind the Methodbook's process.

3.1 Relational Topic Model

Relational Topic Model (RTM) [10] is a hierarchical probabilistic model of document attributes and network structure ($i.e.$, links between documents). RTM provides a comprehensive model for analyzing and understanding interconnected networks of documents. Other models for explaining network link structure do exist (see related work by Chang *et al.* [10]), however the main distinction between RTM and other methods of link prediction is RTM's ability to consider both document context and links among the documents. This also distinguishes RTM by other topic modeling techniques, like *Latent Dirichlet Allocation* (LDA) or *Latent Semantic Indexing* (LSI) that only consider textual information from the documents to model.

There are two steps required to generate a model: (i) model the documents in a given corpus as a probabilistic mixture of latent topics and (ii) model the links between document pairs as a binary variable. Established as an extension of LDA, step one is identical to the generative process proposed for LDA. In the context of LDA, each document is represented by a corresponding multinomial distribution over the set of topics T and each topic is represented by a multinomial distribution over the set of words in the vocabulary of the corpus. LDA assumes the following generative process for each document d_i in a corpus D [37]:

1) Choose $N \sim$ Poisson distribution (ξ)
2) Choose $\theta \sim$ Dirichlet distribution (α)
3) For each of the N words w_n:
   a) Choose a topic $t_n \sim$ Multinomial (θ).
   b) Choose a word w_n from $p(w_n|t_n, \beta)$, a multinomial probability conditioned on topic t_n.

The second phase for the generation of the model exploited by RTM is as follows:

For each pair of documents d_i, d_j:
   a) Draw binary link indicator
   $y_{d_i,d_j}|t_i, t_j \sim \psi\,(\eta \cdot |t_i, t_j,)$
   where
   $t_i = \{t_{i,1}, t_{i,2}, \ldots, t_{i,n}\}$

The link probability function ψ_ϵ is defined as:

$$\psi_\epsilon(y = 1) = \mathtt{exp}(\eta^T(\overline{\mathtt{t}}_{d_i} \circ \overline{\mathtt{t}}_{d_j}) + v).$$

where links between documents are modeled by logistic regression. The $\circ$ notation corresponds to the Hadamard product, $\overline{\mathtt{t}}_d = \frac{1}{N_d}\sum_n t_{d,n}$ and `exp()` is an exponential mean function parameterized by coefficients η and intercept v.

One key distinction between establishing link probabilities in RTM and the canonical LDA is the underlying data used. Here, RTM uses topic assignments to make link predictions whereas to compute document similarities we use topic proportions for each document. This difference is discussed in more detail in the original work by Chang *et al.* [10].

Proposed applications of RTM include identifying potential friends within a social network of users, suggesting citations for a given scientific paper, locating web pages relevant to a web page of interest, and analyzing software artifacts to assist with software maintenance tasks and other tasks [26], [38], [39], [40], [41], [42].

3.1.1 RTM Configuration used in Methodbook

In Methodbook we configured the RTM parameters as done in the work by Gethers and Poshyvanyk [38]. Our choice is due to the fact that also Gethers and Poshyvanyk [38] have applied RTM to text extracted from source code, and in particular to measure coupling between classes. The following setting was used:

- $|T|$ = 75. This is the number of topics that the latent model should extract from the data.
- α = 0.1. This parameter influences the topic distributions per document.
- β = 1.0. This parameter affects the terms distribution per topic.
- η = 1.0. RTM parameter used in the link probability function.

While in this paper we set all RTM parameters based on our experience and prior work, it is also possible to devise near optimal values of these parameters using recently proposed approaches [43].

Note that the above reported configuration has been used in all the empirical studies that we conducted to evaluate Methodbook, as reported in Sections IV and V. In other words no project specific configurations have been used.

3.2 Identifying Method Friendships

The *method friendships* are identified using RTM through the analysis of structural and textual relationships among methods as well as the original structure of the classes (see Figure 1).

As the very first step, methods are analyzed to extract words contained in comments, identifiers, and string literals. Methodbook takes into account (i) all types of comments (i.e., Javadoc and inline comments), (ii) all types of identifiers (i.e., name of variables declared as well as used, name of parameters, name of the method declaration as well as of the invoked methods), and (iii) all literal strings present in a method. In order to extract words from compound identifiers and comments, advanced algorithms for splitting identifiers are employed [44]. Then, a stop word list is used to cut-off all common English words[5] as well as the Java keywords (e.g., String, int, public). Finally, all terms are converted to lowercase. The extracted information is stored in a $m \times n$ matrix (called *term-by-document matrix*), where m is the number of terms occurring in all the methods, and n is the number of methods in the system (see Figure 1). A generic entry $w_{i,j}$ of this matrix denotes a measure of the weight (i.e., relevance) of the i^{th} term in the j^{th} document. In order to weight the relevance of a term in a document we employ the *tf-idf* weighting schema [45]:

$$w_{i,j} = tf_{i,j} \cdot idf_i$$

where $tf_{i,j}$ and idf_i are the term frequency and the inverse document frequency of the term i, respectively. The term frequency is computed as:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

where $n_{i,j}$ represents the occurrences of term i in the document j. The inverse document frequency is computed as:

$$idf_i = log\left(\frac{n}{doc_i}\right)$$

where doc_i is the number of documents where the term i appears. The *term-by-document matrix* weighted with the *tf-idf* schema represents a common model for representing conceptual information, that has been previously used to support different software maintenance tasks (e.g., [38]).

A light-weight static analysis[6] is also applied to the software system to detect (i) structural dependencies between methods (i.e., method calls and shared instance variables) and (ii) the original system design. The latter is a simple boolean $n \times n$ matrix (called *original design matrix*), where n is the number of methods composing the software system. A generic entry $o_{i,j}$ of this matrix is equal to 1 if the method m_i and the method m_j are grouped in the same class in the original design, otherwise it is 0. Concerning the structural dependencies among the methods of the system, Methodbook exploits two structural measures, namely Structural Similarity between Methods (SSM) [46] and Call-based Dependence between Methods (CDM) [29], previously used to compute similarities between methods for identifying Extract Class refactoring opportunities [29], [31], [35]. These measures do not correlate and capture two orthogonal aspects of method relationships [29].

SSM captures attribute references in methods and it is used to build the *shared-data matrix*. Let I_i be the set of instance variables referenced by method m_i. The SSM of m_i and m_j is calculated as the ratio between the number of referenced instance variables shared by methods m_i and m_j and the total number of instance

5. http://www.textfixer.com/resources/common-english-words.txt

6. The static analysis is performed using the Eclipse AST parser.

variables referenced by the two methods:

$$SSM(m_i, m_j) = \begin{cases} \frac{|I_i \cap I_j|}{|I_i \cup I_j|} & \text{if } |I_i \cup I_j| \neq 0; \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the higher the number of instance variables the two methods share, the higher the similarity between the two methods.

CDM [29] takes into account the calls performed by the methods and it is used to build the *calls-interaction matrix*. Let $calls(m_i, m_j)$ be the number of calls performed by method m_i to m_j and $calls_{in}(m_j)$ be the total number of incoming calls to m_j. $CDM_{i \to j}$ is defined as:

$$CDM_{i \to j} = \begin{cases} \frac{calls(m_i, m_j)}{calls_{in}(m_j)} & \text{if } calls_{in}(m_j) \neq 0; \\ 0 & \text{otherwise.} \end{cases}$$

$CDM_{i \to j}$ values are in $[0, 1]$. If $CDM_{i \to j} = 1$, then m_j is only called by m_i. Otherwise, if $CDM_{i \to j} = 0$, then m_i never calls m_j. To ensure that CDM represents a commutative measure, the overall CDM of m_i and m_j is:

$$CDM(m_i, m_j) = \max \{CDM_{i \to j}, CDM_{j \to i}\}$$

The set of friendships derived by analyzing structural similarity between methods are supplied as existing links to RTM. RTM models each method represented in the *term-by-document matrix* as random mixtures over latent topics, where each topic is characterized by a probabilistic distribution over words and is represented by a set of words mostly relevant for explaining the topic [10]. As explained in Section III-A, RTM is able to adjust the probability distribution of each topic taking into account explicit relationships between documents. In Methodbook, explicit relationships between documents (methods) are modeled through (i) the structural dependencies existing among the methods, and (ii) the original design.

The enriched topic distribution model (based on both textual and structural information) obtained by RTM is used to compute similarities among all the methods of the system. Such similarities are stored in a $n \times n$ matrix (where n is the number of methods in the system), namely *RTM similarity matrix*, that is employed to identify move method refactoring operations (see Figure 1).

3.3 Identifying the Envied Class

Once the *RTM similarity matrix* has been computed, the information stored in it is used to determine the degree of similarity among methods in the system and rank friendships among these methods. A cut point is then used to identify the μ best friends of (the methods having the highest similarity with) the method under analysis. Once the "best" friends of a given method are identified, Methodbook analyzes the classes where these methods are implemented aiming at identifying the envied class. Having this information, the first possible way to identify the envied class is to simply find the class containing the highest number of identified friend methods. However, in this way the approach will not take into account the class size. In other words, if for a method under analysis m_k, a class C_i composed of 50 methods contains 4 best friends of m_k, while a class C_j composed of 4 methods contains 3 best friends of m_k, the approach will identify as envied class C_i totally ignoring the fact that only the 8% (4/50) of the methods in this class are friends of m_k while 75% (3/4) of methods of class C_j are friends of m_k. To avoid this issue, the envied class is identified as the one containing the highest percentage of best friends of m_k among its methods (in the previous example, C_j with 75%). Note that if two or more classes contain identical percentage of friend methods, the envied class is the class that contains the highest ranked best friend methods.

When the envied class has been identified, Methodbook verifies that a set of refactoring preconditions is satisfied when moving the method from its original class to the envied class. We use the same set of move method refactoring preconditions defined by Tsantalis and Chatzigeorgiou [9] to ensure that the program behavior does not change after the application of the suggested refactoring. These preconditions are classified in three different categories [9]: (i) compiling preconditions, e.g., the envied class does not contain a method having the same signature as the moved method, (ii) behavior-preservation preconditions, e.g., the envied class should not inherit a method having the same signature as the moved method, and (iii) quality preconditions, e.g., the method to be moved should not contain assignments of a source class field. A complete explanation of verified preconditions is provided by Tsantalis and Chatzigeorgiou [9]. If an identified refactoring opportunity satisfies all the preconditions, the move method operation is suggested by Methodbook. Otherwise, the second class containing the highest percentage of top friends for the method under analysis becomes the new candidate envied class and thus is object of the precondition verification. This process is performed until (i) an envied class satisfying the preconditions is identified or (ii) the original class becomes a candidate envied class, leading Methodbook to not suggest any refactoring operation for the analyzed method.

It is worth noting how there are cases where the identification of an envied class is trivial, i.e., there is a class containing a sensibly higher percentage of friend methods than the other classes. However, there might also be cases where the envied class is difficult to identify, i.e., there are two or more classes that contain a comparable percentage of friend methods. To provide further support to software engineers, the suggestion of envied class is supplemented with a

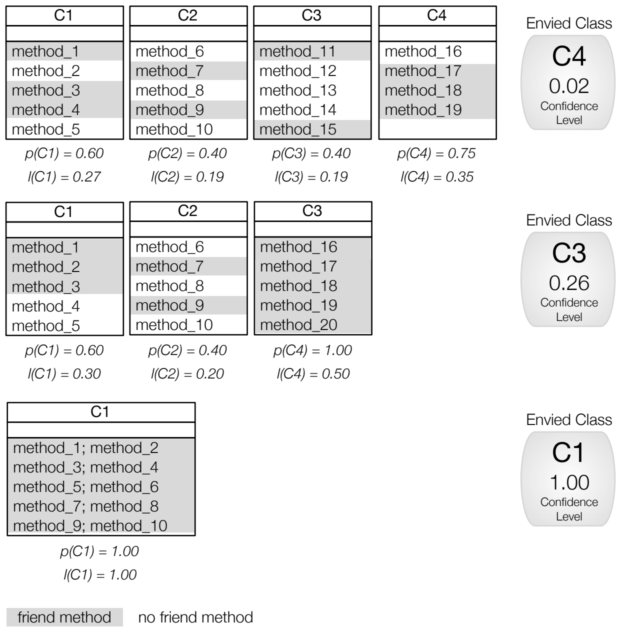


Fig. 2: Three examples of envied class identification with different confidence levels.

confidence level that indicates the reliability of the proposed refactoring. The confidence level uses the concept of information entropy, which measures the amount of uncertainty of a discrete random variable [47]. In particular, we consider the suggestion of an envied class as a random variable, where the probability of its states is given by the distribution of the friend methods over the system classes. We compute the confidence level as the entropy of the suggestion of the envied class. That is, the more scattered the friend methods among the classes, the higher the entropy of the suggestion of the envied class, i.e., the higher the difficulty to identify the envied class. On the contrary, if nearly all the friend methods are implemented in a single class, the entropy of the suggestion is low.

The confidence level is computed as follows:

$$\text{Confidence level}_m = 1 - \sum_{c \in C_m} l(c) \cdot log_{|C_m|} \frac{1}{l(c)}$$

where C_m is the set of classes containing the identified method friends for the method to be moved m, while $l(c)$ represents the likelihood that the envied class is c. For a given class c_i, it is computed as:

$$l(c_i) = \frac{p(c_i)}{\sum_{c \in C_m} p(c)}$$

where $p(c)$ is the percentage of methods of the class c that are friends of m. The defined confidence level has a value in the interval [0, 1]. The higher the value, the higher the goodness of the provided recommendation.

Figure 2 shows three examples of identifying envied class with different confidence levels. In these scenarios the number of best friends identified is ten. In the first case, the friend methods are scattered across several classes. In this case the envied class is C_4 with a very low confidence level, i.e., 0.02. The situation is different in the second example, where, even if three classes contain best friends of the method under analysis, the class C_3 contains a higher percentage of friend methods than the other classes. In this case the confidence level is higher (0.26) indicating a better recommendation reliability as compared to the prior scenario. Finally, the last scenario is the best possible: all the ten best friend methods are concentrated in a single class, i.e., C_1. This will result in a recommendation with the maximum confidence level (1.0).

4 Evaluation based on Metrics

One widely accepted rule to increase the maintainability of software systems is to pursue low coupling and high cohesion [1], [2], [3], where coupling measure the degree to which each program module relies on each one of the other modules, while cohesion is the degree to which the elements of a module belong together [1].

The *goal* of this case study is to (i) evaluate the impact on cohesion and coupling metrics of the refactoring operations recommended by Methodbook and (ii) assess the confidence level as indicator of the quality of the recommended move method operations. Good move method recommendations should help to improve class cohesion while reducing coupling between classes. Also, we need to verify if high values of the confidence level are associated with high quality of refactoring recommendations. Thus, the following research questions were formulated:

- **RQ**$_1$: *What is the impact of the Methodbook's refactoring recommendations on cohesion and coupling?*
- **RQ**$_2$: *Is the confidence level a good indicator for the goodness of Methodbook's recommendations?*

The experimentation was carried out on two open source software systems, namely jEdit[7] and JFreeChart[8], on three industrial projects, namely AgilePlanner[9], eXVantage[10], and GESA[11], and on a software system, SMOS, developed by a team of Master's students at the University of Salerno (Italy) during their industrial internship. jEdit is a programmer's text editor supporting hundreds of programming languages. JFreeChart is a chart library supporting the creation of many kinds of charts in Java applications. AgilePlanner is an industrial tool that supports agile teams in project planning, while eXVantage is a product line of eXtreme Visual-Aid Novel Testing and Generation tools which focuses on providing code coverage information to software developers and testers. GESA automates the most important

7. http://www.jedit.org/ verified on 08/16/2013
8. http://www.jfree.org/jfreechart/ verified on 08/16/2013
9. http://ase.cpsc.ucalgary.ca/ verified on 08/16/2013
10. http://www.avaya.com/usa/avaya-labs/ verified on 08/16/2013
11. http://www.distat.unimol.it/gesa/ verified on 08/16/2013

TABLE 1: Software systems used in the case study

System	KLOC	Classes	Methods	Connectivity$_{avg}$	C3$_{avg}$	MPC$_{avg}$	CCBC$_{avg}$	MTerms$_{avg}$	UniqueMTerms$_{avg}$
AgilePlanner 2.5.0	24	299	2,731	0.195	0.220	3.460	0.070	22	5
eXVantage 2.01	36	352	2,172	0.240	0.283	2.707	0.077	21	6
GESA 2.2	46	295	1,643	0.144	0.082	10.955	0.349	66	7
jEdit 3.0	72	425	1,864	0.228	0.155	4.275	0.038	25	10
JFreeChart 0.9.6	107	436	1,847	0.143	0.185	0.836	0.077	24	6
SMOS 1.0	23	121	599	0.197	0.082	5.984	0.389	51	7
Total	308	1,938	10,856	-	-	-	-		

activities in the management of university courses, like timetable creation and classroom allocation. It is operational since 2007 at the University of Molise (Italy). Finally, SMOS is a software developed for high schools, which offers a set of features aimed at simplifying the communication and interaction between the school and the parents of the students.

Table I reports the size, in terms of KLOC, number of classes, and number of methods, as well as the versions of the object systems. The table also reports the average verbosity of methods for each object systems in terms of (i) average number of terms in the methods (column MTerms$_{avg}$), and (ii) average number of unique, i.e., different, terms in the methods (column UniqueMTerms$_{avg}$). This information is important to check the availability of the textual information exploited by Methodbook in the object systems. Note that we excluded those terms appearing in the stop word list (since they are ignored by our approach) from the total count of terms.

Table I also shows the average value for four metrics aimed at measuring class cohesion and coupling at structural (i.e., Connectivity [48] and Message Passing Coupling (MPC) [49]) and semantic[12] level (i.e., Conceptual Cohesion of Classes (C3) [6] and Conceptual Coupling Between Classes (CCBC) [50]) computed considering all the classes of the object systems. Connectivity is a structural metric to measure class cohesion and it is computed as the number of method pairs in a class sharing an instance variable or having a method call among them divided by the total number of method pairs in the class. We did not consider constructors and accessor methods (i.e., getter and setter) as methods since, as highlighted by Briand *et al.* [48], they can artificially increase the class cohesion. C3 is a conceptual cohesion metric, complementary to structural cohesion, which exploits LSI (Latent Semantic Indexing) [51] to compute the overlap of textual information in a class expressed in terms of textual similarity among methods. Higher values of C3 indicate higher class cohesion. The MPC is a structural coupling metric based on method-method interaction. MPC measures the number of method calls defined in methods of a class to methods in other classes, and, therefore, the dependency of local methods to methods implemented by other classes. Higher MPC values indicate higher coupling. Finally, CCBC is another coupling metric based on the textual information captured in the code by comments and identifiers. Two classes are conceptually related if the terms present in their comments and identifiers are similar.

12. Note that “semantic metrics” usually indicate in the literature metrics exploiting textual information from source code.

We evaluate the impact that the move method operations recommended by Methodbook have on these four quality metrics. Note that these four metrics do not directly measure the design quality of a system. However, they have been shown to measure desirable quality aspects of a software system. In particular:

1) Classes with low cohesion have been shown to correlate with high defect rates [6]. The C3 is the only semantic cohesion metric available in literature, while we choose the Connectivity metric on the structural side since on the contrary of other structural cohesion metrics, e.g., Lack of Cohesion of Methods (LCOM) [36], considers two methods to be cohesive not only if they share an instance variable, but also if they have a call among them.
2) MPC has been shown to directly correlate with maintenance effort [49]. Thus, higher MPC values (higher coupling) indicate higher effort in maintaining a software system.
3) CCBC has been used to support change impact analysis. In other words, two classes exhibiting high CCBC are likely to be changed together during a modification activity performed in a system. Consequently, having classes with high CCBC between them grouped together in the same software module could reduce the effort needed by a developer to localize the change. This clearly results in more manageable maintenance activities.

Thus, if we are able to increase the average class cohesion and/or reduce the average class coupling of the object systems while applying move method operations suggested by Methodbook, this represents a first indication of the goodnesses of the Methodbook’s recommendations.

4.1 Planning

To respond to our research questions we used Methodbook to suggest an envied class for all the methods in the studied software systems (for a total of 10,856 methods)[13]. The set of methods for which Methodbook did not identify the original class as envied class represents the move method refactoring operations suggested by our approach. To respond to our first research question (**RQ**$_1$), we applied them incrementally starting from those having the higher confidence level (see Section III). After performing each refactoring operation we measured the value for the four quality metrics presented above, i.e., Connectivity, C3, MPC, and CCBC. In this way we were able to observe the impact of the refactoring operations on the object systems in terms of class cohesion and coupling. To better evaluate the goodnesses of the refactorings suggested by Methodbook, we also executed the approach presented by Tsantalis and Chatzigeorgiou [9] (using the JDeodorant Eclipse plug-in) on the same six object systems in order to obtain the move method refactoring operations suggested by the competitive approach. In this way, we were able to compare the cohesion and coupling trends obtained using Methodbook with those obtained using JDeodorant [9]. Note that the two structural quality metrics used in our evaluation (i.e., Connectivity and MPC), and two of the adopted object systems (i.e., jEdit and JFreeChart) were also used in the original JDeodorant evaluation [9].

Concerning our second research question (**RQ**$_2$), the order in which these refactoring operations are applied allows to analyze a possible correlation between the confidence level and the goodnesses of the suggested refactoring operations. If the confidence level is a good indicator for the goodness of Methodbook's recommendations, we expect to observe higher increase in average class cohesion and a higher decrease in average class coupling for higher confidence levels of a refactoring operation (and *vice versa*).

In the following section we report the results while setting the number of top friends considered by Methodbook (i.e., the μ parameter, see Section III) to ten. Our choice is not random since we tried several different values for this parameter (1, 3, 5, 7, and 10) and, after manually analyzing the suggestions proposed by Methodbook[14], we believe that the best refactoring recommendations are usually obtained using $\mu = 10$. The main advantage we observed in setting $\mu = 10$, was the higher reliability of the confidence level as indicator of refactoring suggestions goodness. In fact, the higher the number of "best" friend methods evaluated, the higher the possible values of entropy (and thus, of the confidence level) that we can measure. While this could seem like just a small a detail, during the assessment we found that a high confidence level corresponds to far more reliable refactoring recommendations when considering ten friends than when considering a lower number of friends. On the other side, a higher value for μ was simply not practical, given that we want to consider only the "best" (i.e., top) friends of a method under analysis.

TABLE 2: Number of move method refactorings suggested by Methodbook and JDeodorant

System	#Methods	Methodbook	JDeodorant	Methodbook ∩ JDeodorant
AgilePlanner	2,731	27	69	1
eXVantage	2,172	95	80	25
GESA	1,643	30	165	0
jEdit	1,864	8	18	7
JFreeChart	1,847	10	18	5
SMOS	599	27	69	0
Total	10,856	198	419	38

4.2 Analysis of the Results

Before answering our research questions, it is important to quantitatively discuss the suggestions generated by Methodbook and JDeodorant[15], in order to verify if (i) both the approaches generate move method suggestions on all object systems, and (ii) what is the overlap in terms of generated suggestions (i.e., to what extent the two techniques suggest the same move method refactorings).

Table II reports the number of suggestions generated by Methodbook and JDeodorant on each of the object systems. Also, column "Methodbook ∩ JDeodorant" shows the number of equal suggestions generated by both techniques (i.e., the same method is moved to the same envied class). Note that there were no cases when Methodbook and JDeodorant suggested to move the same method to two different envied classes. The main observation from data reported in Table II are:

- *JDeodorant generally generates more suggestions than Methodbook*. This holds for all object systems but eXVantage. This result may seem unintuitive, given the fact that Methodbook exploits more information than JDeodorant when generating the refactoring suggestions (i.e., Methodbook also exploits textual information from source code). However, this does not ensure that it is able to identify a higher number of recommendations. In fact, it could happen that while analyzing a method m_i from a class C_i, by only taking into account structural information, the majority of "best friends" of m_i are implemented in a class C_j, while by also taking into account textual information it turns out that the majority of "best friends" of m_i are implemented in C_i, thus leading to an empty refactoring suggestion.
- *the number of methods moved by the two approaches just represent a small percentage of the methods in the systems*. In total, Methodbook suggests to move 2% of the methods (i.e., 198 out of 10,856) and JDeodorant 4% (i.e., 419 out of 10,856).

13. We applied Methodbook in isolation on each object system.
14. The training of Methodbook has been performed on a system not used in its empirical evaluation (i.e., Apache Xerces).
15. The move method recommendations are available online [52].

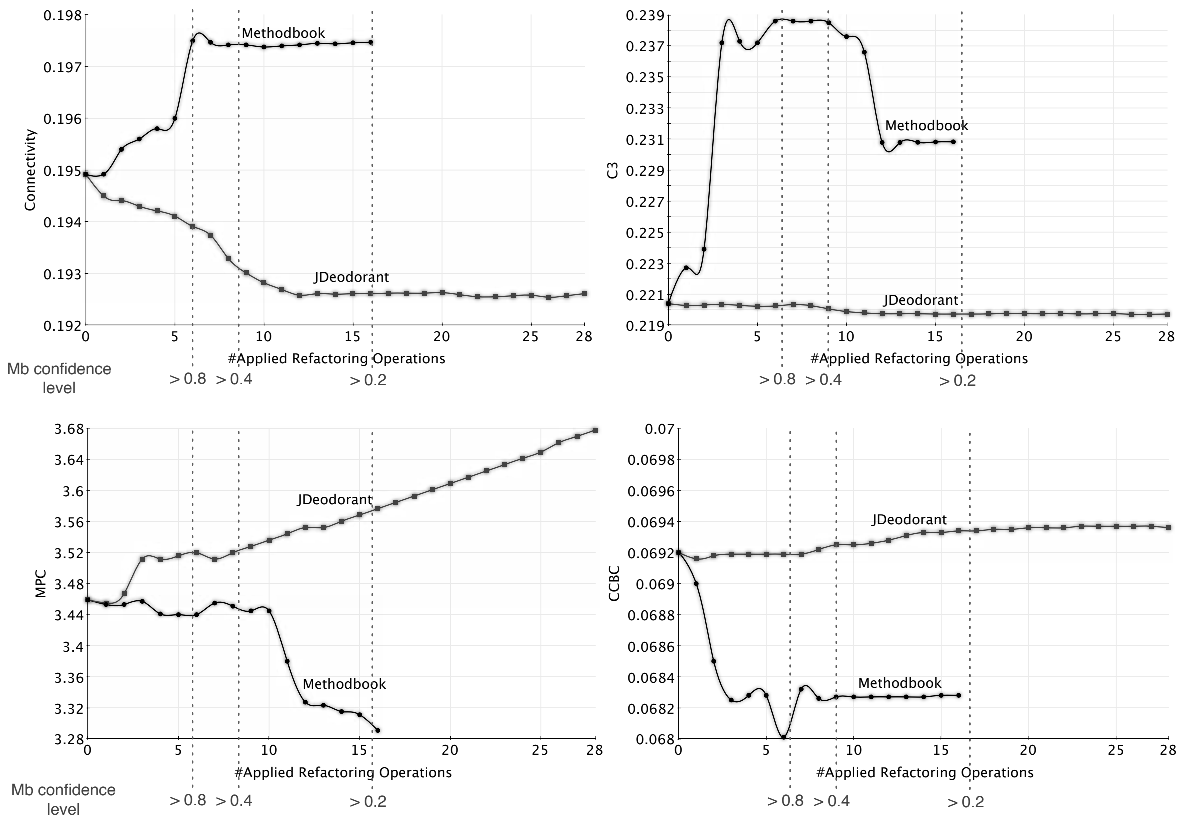


Fig. 3: Evolution of the four quality metrics on AgilePlanner by applying the refactoring operations suggested by Methodbook (16) and JDeodorant (28)

- *there is generally a small overlap between the suggestions generated by Methodbook and those generated by JDeodorant*. The only exceptions are eXVantage, where there is an overlap of 25 suggestions, jEdit, where 7 out of the 8 suggestions generated by Methodbook are also generated by JDeodorant, and JFreeChart, with 5 out of the 10 Methodbook's recommendations also provided by JDeodorant. This result highlights that while generally identifying a lower number of suggestions, Methodbook retrieves *Feature Envy* smells that are ignored by JDeodorant. Since JDeodorant only exploits structural information, those additional suggestions are likely due to the textual information exploited by Methodbook (e.g., a method having the same amount of structural dependencies with methods in its class and with methods in the envied class, but a much higher textual similarity with methods in the envied class).

Figures 3 and 4 show the evolution of the four employed metrics by applying the refactoring operations suggested by Methodbook and JDeodorant on AgilePlanner and SMOS, respectively. The graphs for the other four systems, i.e., eXVantage, jEdit, JFreeChart, and GESA, can be found in Appendix A. As explained before, the suggestions by Methodbook are applied in a decreasing order of confidence level. The dotted lines in the graphs show different confidence level thresholds to get an idea of which refactoring operations improve more the metrics. As an example, in Figure 3 the first six Methodbook's suggestions have a confidence level higher than 0.8, the 7th and the 8th have a confidence level lower than 0.8 and higher than 0.4, from 9th to 17th lower than 0.4 and higher than 0.2, from 18th to 28th lower than 0.2.

The results achieved on AgilePlanner (Figure 3) show that Methodbook is able to improve the four cohesion and coupling metrics. In particular, the Connectivity cohesion metric shows a strong increase during the application of the first 6 suggestions by Methodbook (i.e., those having a confidence level higher than 0.8). Applying these 6 suggestions also results in a very high increase of the semantic cohesion (C3 metric) together with a decrease of the structural and semantic coupling (MPC and CCBC metric, respectively). However, when applying Methodbook's recommendations having a confidence level lower than 0.8 the achieved results are quite different. In fact, there is no a clear improvement of the cohesion and/or coupling of the system classes. On the contrary, more often than not, the application of these move method operations results in deteriorating the cohesion/coupling of the classes with respect to the levels reached after the application of the recommendations having high confidence level.

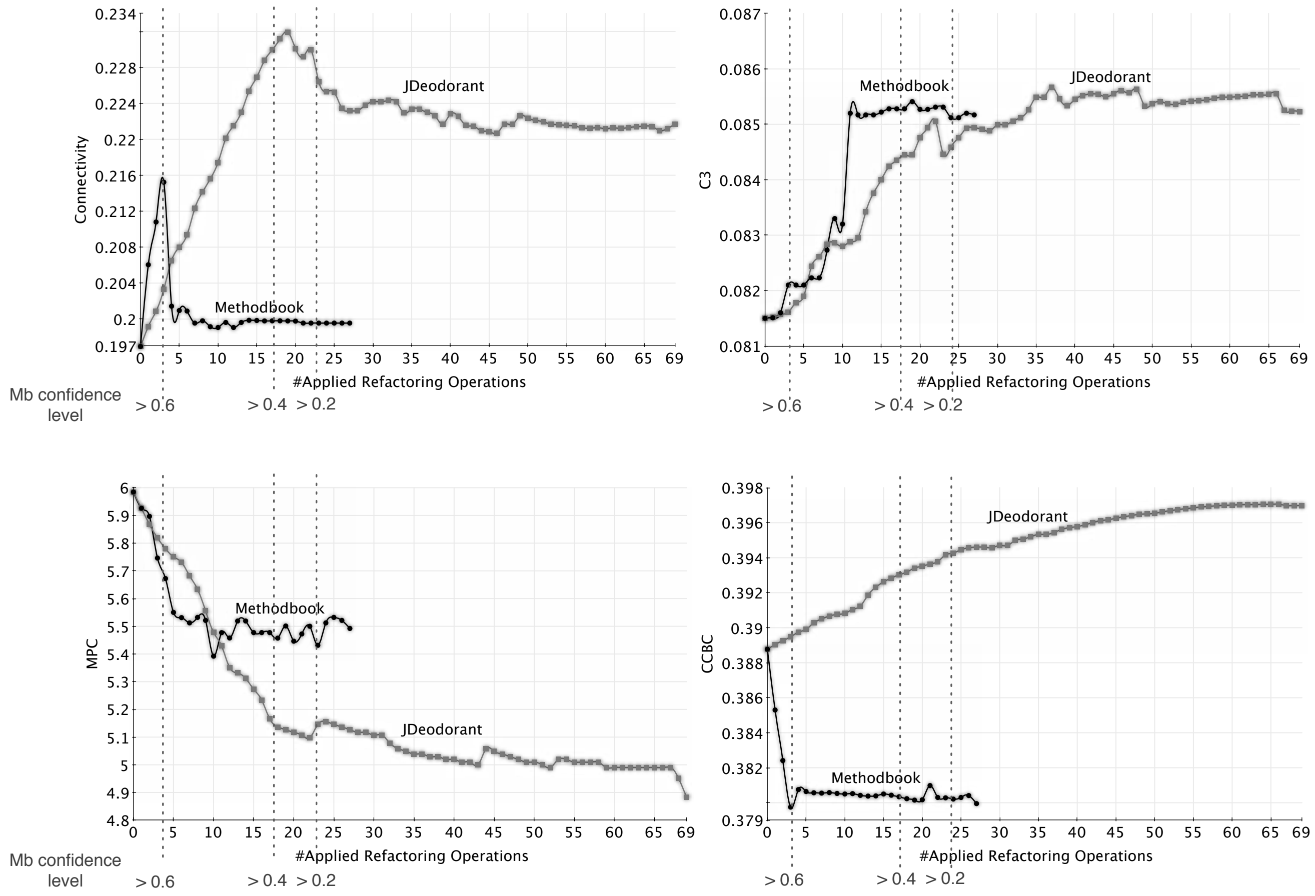


Fig. 4: Evolution of the four quality metrics on SMOS by applying the refactoring operations suggested by Methodbook (27) and JDeodorant (69).

On the same system, the application of the move method operations suggested by JDeodorant results in a decrease of the average structural and semantic class cohesion together with an increase of the average structural and semantic class coupling (see Figure 3). Note that on this system, only one of the move method refactoring suggested by JDeodorant is also suggested by Methodbook (see Table II).

The situation is totally different on SMOS (see Figure 4). On this system JDeodorant is able to achieve very good performances for the two structural metrics, i.e., Connectivity and MPC, while on the semantic side it is able to improve only the cohesion (the semantic coupling CCBC increases). This result is likely due to the fact that JDeodorant does not take into account textual information during the generation of the move method recommendations. As for Methodbook, it suggests a lower number of operations as compared to JDeodorant (27 *vs* 69) on SMOS, none in overlap with JDeodorant. Moreover, these suggestions have a confidence level not so high (the highest recommendation rate is 0.771 and only 3 recommendations have a confidence level higher than 0.6 - see Figure 4). Concerning the structural metrics, Methodbook's recommendations having confidence level higher than 0.6 are able to improve all the quality metrics, while, as observed on AgilePlanner, low confidence level operations are not always able to improve the quality metrics. As expected, on the semantic side Methodbook performs better than JDeodorant, ensuring increase of semantic cohesion and decrease of semantic coupling.

As for the other software systems, on eXVantage the trend is almost the same as on AgilePlanner (see Figure 9). In fact, on this system (i) Methodbook's suggestions having high confidence level (i.e., higher than 0.6) increase the structural and semantic cohesion while decrease the structural and semantic coupling, and (ii) JDeodorant is not able to achieve comparable results. Concerning GESA, the results are almost in-line with SMOS: JDeodorant is able to improve structural cohesion and coupling metrics while Methodbook is able to improve both the structural metrics (but less than JDeodorant) as well as the semantic ones (see Figure 8). Note that, as for SMOS, also on GESA the suggestions by Methodbook generally have a low confidence level. jEdit is interesting since here seven out of the eight Methodbook's suggestions are also generated by JDeodorant. The only one not captured by JDeodorant has confidence level equals 1 and it is the first applied in Figure 10, showing good performances in improving all four metrics. Overall, Methodbook is able to achieve better performances on the semantic side (still improving the structural metrics), while JDeodorant performs slightly better on the structural one. Finally, on JFreeChart (see Figure 11), Methodbook performs better than JDeodorant for all metrics but Connectivity, where still it is able to achieve a good improvement. Also on these two systems, Methodbook's suggestions having a higher confidence level generally result in stronger improvements in terms of quality metrics.

Thus, while Methodbook always outperforms JDeodorant on the semantic side (expected result), it is interesting to note as on the structural side it is able to performs better than JDeodorant only on three systems, while on the remaining three JDeodorant stands out. We inspected different system characteristics trying to understand which were the reasons behind these inconsistencies of performances. The data reported in Table I provided a possible answer. In particular, the information about the verbosity of the methods present in the object systems showed that the three systems on which JDeodorant outperforms Methodbook on the structural side (i.e., GESA, SMOS, and jEdit) are those containing the more verbose methods. Indeed, these systems have (i) a higher average number of terms per method (column MTerms$_{avg}$) and (ii) a higher average number of unique, i.e., different, terms per method (column UniqueMTerms$_{avg}$) as compared to the other three systems. This result might seem counterintuitive, since one could expect that the more the textual information present in the source code the better the Methodbook's suggestions. However, on these three systems the Methodbook's suggestions are still good, but more focused on improving the semantic metrics rather than the structural ones (that are still improved). This is likely due to the large quantity of textual information present in these systems. Note that all these three systems will be part of our studies conducted with developers (Section V), where the quality of the Methodbook's suggestions (as well as of the JDeodorant ones) will be judged from a developer's point of view.

In summary, our results highlight the importance of the confidence level as indicator of goodnesses of the Methobook's suggestions (**RQ**$_2$). When the confidence level is high (generally higher than 0.6) the Methodbook's recommendations are able to improve the values of class cohesion and coupling (**RQ**$_1$). Moreover, on three out of six object systems (AgilePlanner, eXVantage, and JFreeChart) Methodbook performs better than JDeodorant while on the remaining three systems (GESA, SMOS, and JEdit) Methodbook achieves a better improvement of the metrics only on the semantic side.

5 Evaluation with Developers

In our previous case study (Section IV) we evaluated the Methodbook's recommendations by measuring the difference between pre- and post-refactoring in terms of class cohesion and coupling. However, the refactoring operations should not only improve the values of some quality metrics but should also be meaningful from a developer's point of view. Thus, the following research question was formulated in this new experimentation:

- **RQ**$_3$: *Are the refactoring recommendations produced by Methodbook meaningful from a developer's point of view? How do they compare with those generated by JDeodorant?*

In order to answer this research question, we performed two studies involving software developers in the evaluation of refactoring operations proposed by Methodbook and JDeodorant. The first study was conducted on jEdit and JFreeChart involving 56 and 70 developers, respectively. Since they have not participated in the development of jEdit and JFreeChart we refer to them as "external developers". The second study was conducted on GESA and SMOS with the original developers of the systems (5 developers for each system).

It was necessary to perform both these studies to evaluate Methodbook from all possible perspectives. Indeed, the study with external developers is not enough since they do not have deep knowledge of the design of the software system. Thus, they may be not aware of some of the design choices that could appear wrong, but that are the results of a conscious choice. This is the reason why we also performed a user study with original developers. However, this study alone is also not enough. Even if the original developers have deep knowledge of all the design choices that led to the original design, they could be the "fathers" of some bad design choices and consequently could not recognize a good move method suggestion as meaningful. This threat is mitigated by the study conducted with the external developers. Thus, the two experiments are complementary and allow us to investigate the meaningfulness of the suggestions performed by Methodbook and JDeodorant from different points of view.

5.1 Evaluation with External Developers

In this section we report the design of the study and the results achieved in our first evaluation conducted with external developers.

5.1.1 Planning

To recruit participants for our study we invited 105 people all around the world among students, academics, and industrial developers. We asked each of them to complete two questionnaires aimed at

TABLE 3: External Developers Involved

System	#Participants	Bachelors Students	Masters Students	PhD Students	Faculty	Industrial Developers
jEdit	56	6	12	14	1	23
JFreeChart	70	7	18	15	1	29

evaluating all refactoring suggestions generated by Methodbook and JDeodorant on jEdit and JFreeChart, respectively. Of them, 56 completed the questionnaire for both systems, while 14 only for JFreeChart. Thus, of the 105 people invited, 56 evaluated the refactoring suggestions on jEdit and 70 on JFreeChart.

Table III reports the number of external developers involved in our study by classifying them on the basis of their background. The vast majority of our participants are industrial developers (41% on both jEdit and JFreeChart), while we just had one faculty and few Bachelors students (almost 10% of participants for both systems). These latter, represent the less experienced participants involved in our study. However, they were third year Bachelors students that in the context of the Software Engineering course had participated in software projects, where they practiced software development and documentation production. While all the external developers involved in our study had good knowledge of object oriented design principles, they had little prior knowledge of the object systems. However, this is wanted by design (as explained before) as we also executed another experiment with original developers.

Each participant received via e-mail (i) the source code of the two object systems, and (ii) the links to two separate questionnaires (one for jEdit and one for JFreeChart) implemented as web-applications. Each question was related to one of the refactoring operations generated by Methodbook/JDeodorant. For each operation, participants had to answer to the question

Would you apply the proposed refactoring?

assigning a score on a three point Likert scale: 1 (no), 2 (maybe), and 3 (yes). Also, participants could add an optional comment explaining the rational behind each score. We gave participants three weeks to complete both questionnaires. Note that the participants were not aware of the experimented techniques, i.e., Methodbook and JDeodorant, nor of the fact that the move method refactoring suggestions were automatically generated. The web-applications hosting the questionnaire was also used to measure the time spent by each participant in answering each question. The latter information was collected to remove participants that randomly evaluated the refactoring operations from the analysis of the results. Our aim was to remove all participants that answered to at least one question in less than 10 seconds, since it is very unlikely that in such little time a developer is able to analyze the moved method, its original class and the suggested envied class to make a decision on the quality of the refactoring. Based on this criterion, none of the participants involved in our evaluation was removed from the analysis of the results.

The jEdit questionnaire included 19 questions and was composed as follows:

- eleven questions were related to move method refactorings generated only by JDeodorant. We refer to these refactoring suggestions as the *onlyJDeodorant* group (OJ);
- seven questions were related to move method refactorings generated by both JDeodorant and Methodbook (i.e., both techniques suggested to move the same method in the same envied class). We refer to these refactoring suggestions as the *both* group (B);
- one question was related to a move method refactoring generated only by Methodbook. We refer to these refactoring suggestions as the *onlyMethodbook* group (OM).

Concerning the JFreeChart questionnaire, it was composed of 23 questions, of which 13 fell in the OJ group, 5 in the B group, and 5 in the OM group. Note that both questionnaires include the evaluation of all refactoring recommendations generated by both Methodbook and JDeodorant on the object systems (see Table II).

We analyzed the answers provided by the participants through descriptive statistics and statistical tests. As for the statistical tests, for each of the three groups of methods (i.e., OJ, B, and OM), we collected the scores assigned by participants. Then, considering two particular groups, e.g., OM *vs.* OJ, we used the Wilcoxon test [53] to analyze the statistical significance of scores assigned by participants to the move method refactorigs in the two groups. However, since we performed multiple tests, we adjusted our p-values using the Holm's correction procedure [54]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on. The results were intended as statistically significant at $\alpha = 0.05$.

We also estimated the magnitude of the difference between the scores in the different groups. We used Cliff's Delta (or d), a non-parametric effect size measure [55] for ordinal data. The effect size is small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$ [55]. Note that both the Wilcoxon test and the Cliff's Delta are suited when working with unpaired data like the groups of scores used in our tests, i.e., the number of scores in each group is different due to the different number of refactoring operations in each of them.

TABLE 4: Participants' answers to the question "*Would you apply the proposed refactoring?*"

Group	System	*no*	*maybe*	*yes*
	jEdit (11 suggestions)	44%	26%	31%
OJ	JFreeChart (13 suggestions)	48%	22%	30%
	Overall (24 suggestions)	**46%**	**23%**	**30%**
	jEdit (7 suggestions)	30%	17%	53%
B	JFreeChart (5 suggestions)	33%	27%	40%
	Overall (12 suggestions)	**31%**	**22%**	**47%**
	jEdit (1 suggestion)	2%	11%	87%
OM	JFreeChart (5 suggestions)	30%	19%	51%
	Overall (6 suggestions)	**26%**	**18%**	**56%**

5.1.2 Analysis of the Results

Table IV reports the answers provided by the participants to the question "*Would you apply the proposed refactoring?*". The data are presented by group of methods, i.e., OM, OJ, and B. In addition, Table V shows the results of the Wilcoxon test and the effect size d, while Table VI classifies the refactoring suggestions in each group on the basis of the most frequent score they received from participants. Note that, given the three possible scores that can be assigned to a refactoring operation (i.e., no, maybe, and yes), the most frequent score must be chosen by at least 34% of subjects (e.g., 33% yes, 33% no, and 34% *maybe*). On JFreeChart, where each refactoring recommendation has been evaluated by 70 subjects, the most frequent score has been chosen, on average, by 61% of subjects (median 62%), while on JEdit, where 56 subjects evaluated the refactoring recommendations, the most frequent score has been chosen, on average, by 68% of subjects (median 70%). Overall, the maximum agreement reached by subjects is 91% achieved on two recommendations generated on JEdit, while the minimum is represented by 43% achieved on one recommendation generated by JFreeChart.

Going to the results, on the jEdit system only one suggestion fell in the OM group (i.e., only Methodbook suggests to move the method from its original class to a new envied class). This suggestion had confidence level of one (the maximum possible) and has been highly appreciated by participants, with 87% of them (49 out of 56) answering *yes*, 11% (6 out of 56) *maybe*, and 2% (1 out of 56) *no*. This suggestion was the move of the method `getStyleString`, shown in Figure 5, from its class `GUIUtilities` to the envied class `SyntaxStyle`. Among the comments left by the participants about this move method refactoring, an industrial developer wrote:

I would definitely move the method to the SyntaxStyle class and rename it in toString

Indeed, the method `getStyleString`, as reported in its comment (see Figure 5), is in charge of converting a `SyntaxStyle` object into a String. Thus, the choice of moving it in the envied class seems to be appropriate.

On the same system, seven suggestions were part of the B group (i.e., both the approaches suggest to move a method from its original class to the same envied class). Among these, two suggestions had the maximum confidence level (i.e., one), three a confidence level between 0.5 and 0.7, and the remaining two a very low confidence level (i.e., 0.143 and 0.105). Overall, the percentage of *yes* received by participants for these seven recommendations is 53%, together with a 17% of *maybe*, and a 30% of *no*. By looking at the data in Table VI, five of the seven suggestions belonging to the B group were mostly positively evaluated by the participants, while two were mostly rejected. These latter are those for which Methodbook assigned a very low confidence level and thus, it is somewhat an expected result.

Going to the OJ group for JEdit (i.e., only JDeodorant suggests to move the method from its original class to a new envied class), the eleven suggestions falling in this group were evaluated by participants with 31% of *yes*, 26% of *maybe*, and 44% of *no*. The high percentage of *no* answers is due to five JDeodorant suggestions generally discarded by participants, like for example moving the method `invokeDeclaredMethod` from its class `BSHMethodDeclaration` to the class `BSHFormalParameters`, evaluated with 51 *no*, 4 *maybe*, and 1 *yes*. However, it is also worth noting as

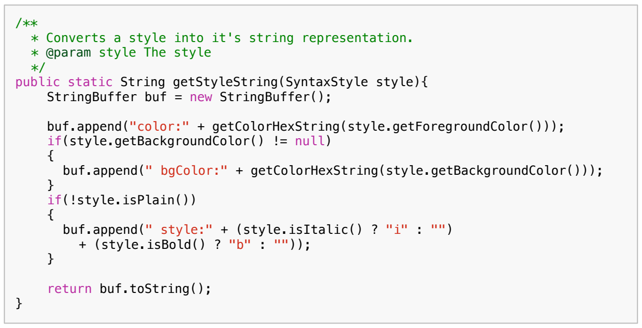
TABLE 5: Scores assigned by external developers: Wilcoxon test (adjusted p-value with Holm's correction) and Cliff's effect size (d). The group achieving the better scores in each comparison is highlight in **bold** face.

Test	jEdit	
	p-value	d
OM *vs* OJ	<0.0001	-0.61 (Large)
OM *vs* B	<0.0001	-0.37 (Medium)
B *vs* OJ	<0.0001	-0.22 (Small)

Test	JFreeChart	
	p-value	d
OM *vs* OJ	<0.0001	-0.24 (Small)
OM *vs* B	0.0066	-0.10 (Small)
B *vs* OJ	<0.0001	-0.15 (Small)

TABLE 6: Refactoring recommendations as evaluated by the majority of participants

Group	System	*no*	*maybe*	*yes*
	jEdit (11 suggestions)	5	2	4
OJ	JFreeChart (13 suggestions)	8	1	4
	Overall (24 suggestions)	**13**	**3**	**8**
	jEdit (7 suggestions)	2	0	5
B	JFreeChart (5 suggestions)	2	1	2
	Overall (12 suggestions)	**4**	**1**	**7**
	jEdit (1 suggestion)	0	0	1
OM	JFreeChart (5 suggestions)	1	0	4
	Overall (6 suggestions)	**1**	**0**	**5**



```
/**
  * Converts a style into it's string representation.
  * @param style The style
  */
public static String getStyleString(SyntaxStyle style){
    StringBuffer buf = new StringBuffer();

    buf.append("color:" + getColorHexString(style.getForegroundColor()));
    if(style.getBackgroundColor() != null)
    {
      buf.append(" bgColor:" + getColorHexString(style.getBackgroundColor()));
    }
    if(!style.isPlain())
    {
      buf.append(" style:" + (style.isItalic() ? "i" : "")
        + (style.isBold() ? "b" : ""));
    }

    return buf.toString();
}
```

Fig. 5: The method `getStyleString` was moved by Methodbook from its class `GUIUtilities` to the envied class `SyntaxStyle`

JDeodorant is able to identify four suggestions mostly appreciated by participants that Methodbook is not able to capture. In other words, it seems that JDeodorant is less conservative than Methodbook, producing more false positives but also good suggestions missed by Methodbook.

As for the JFreeChart system, five suggestions were part of the OM group. Overall they received 51% of *yes*, 19% of *maybe*, and 30% of *no*. However, by looking at the data at a finer granularity level, it turns out how four out of these five refactorings four mostly received *yes* as answer by participants, while only one was rejected with a majority of *no* answers (see Table VI). The latter is the suggestion to move the method `drawRangeMarker` from its class `VerticalBarRenderer3D` to the envied class `Marker`, generated by Methodbook with a confidence level of 0.567 (the lowest among the five refactorings belonging to the OM group). Thirty out of the 70 participants answered that this refactoring should not be performed, 23 answered *maybe*, and 17 *yes*.

Among the negative evaluations we found an interesting explanation:

while this refactoring could make sense,
`VerticalBarRender3D` is a rendering class,
so it is supposed to draw a Marker as well as
other types of objects

All other four suggestions were generally appreciated by the participants with a percentage of *yes* going from 56% up to 65%, while the *no* answers were limited between 20% and 30%. Note that all these four refactorings had a confidence level higher than 0.85.

Five move method recommendations belonging to the B group have been rewarded with 40% of *yes*, 27% of *maybe*, and 33% of *no*. Data reported in Table VI, show that two of these five refactorings were generally appreciated by participants, two were generally rejected, while one has mostly collected *maybe* answers. The two recommendations rejected by developers, both suggesting to move a method from class `ContourPlot` to class `Marker`, had the lowest confidence levels, in one case (0.602) very close to the threshold (0.6) identified in the metric based evaluation and in the other case (0.208) much lower. On the other side, the remaining three suggestions had a high confidence level (1, 0.897, and 0.897, respectively).

The 13 suggestions in the OJ group received a 30% of *yes*, 22% of *maybe*, and 48% of *no*. Also on this system, by looking the results in Table VI, it seems confirmed that JDeodorant generally produces a higher number of false positives than Methodbook, with eight suggestions generally rejected by developers. However, as said before the less conservative nature of JDeodorant allows to identify four "correct" refactoring operations missed by Methodbook.

Finally, the results of the Wilcoxon test reported in Table V show how on both systems:

1) refactorings in the OM group achieve statistically significant higher score than refactorings in the OJ group (large effect size on jEdit and small on JFreeChart);
2) refactorings in the OM group achieve statistically significant higher score than refactorings in the B group (medium effect size on jEdit and small on JFreeChart);
3) refactorings in the B group achieve statistically significant higher score than refactorings in the OJ group (small effect size on both systems).

Summarizing, the results of this study highlight that:

- *when Methodbook suggests a refactoring operation with a high confidence level (higher than 0.6), developers generally appreciate the recommendation*. This result holds on both systems for refactorings present in the OM as well as in the B group. In fact, the only refactoring operation rejected in the OM group had a confidence level of 0.567 and three of the four rejected in the B group had a very low confidence level (lower than 0.210). The only exception is one of the rejected refactoring recommendations in the B group that, however, had a confidence level just slightly higher than 0.6 (i.e., 0.602).
- *The suggestions in the OM group achieved statistically significant higher scores than those present in the OJ and in the B group*. The latter result could seem surprising, since also the suggestions in the B group are generated by Methodbook (and confirmed by JDeodorant). However, while the average confidence level of the refactorings in the OM group is 0.90, this value drops down to 0.64 for refactorings in the B group, explaining the observed difference of scores in favor of the OM group.
- *JDeodorant seems to be less conservative than Methodbook in suggesting refactoring operations*. This is demonstrated by the quite higher number of recommendations generated on both systems (36 against the 18 of Methodbook). This allows

TABLE 7: Number of refactoring operations suggested by Methodbook and JDeodorant on the two object systems

System	JDeodorant	Methodbook
GESA 2.2	165	30
SMOS 1.0	69	27
Total	234	57

JDeodorant to identify a total of 8 good suggestions missed by Methodbook that, on its side, is able to identify only 5 good suggestions missed by JDeodorant. However, the challenge for a refactoring recommendation system is not only to generate good refactoring solutions, but also to avoid surrounding them with noisy and useless suggestions that a developer would just discard. On this front, Methodbook performed better than JDeodorant, with only one refactoring operation discarded in the OM group, against the 13 in the OJ group.

The performed analysis allow us to positively answer to our research question (**RQ**$_3$): Methobook is indeed able to identify meaningful refactoring operations from a functional point of view. However, this is true under a precise condition: the confidence level must be high.

5.2 Evaluation with Original Developers

In this section we report the design and the results achieved in the evaluation conducted with original developers.

5.2.1 Planning

In this study we executed both Methodbook and JDeodorant on the two object systems, i.e., GESA and SMOS. Then, in order to answer our research question (**RQ**$_3$), we asked the original developers of these two systems to evaluate all the refactoring operations suggested by the two techniques. Note that for GESA we were able to involve the entire team (composed of five people) that developed the system while for SMOS we involved five out of the seven developers. All five members of the SMOS project and three of the five members of the GESA project taking part to this experimentation work in industry, while the remaining two members of the GESA project are a Ph.D. student and a Masters' student.

The participants evaluated all the refactoring operations suggested by the two approaches[16] through a questionnaire where, for each operation, they had to answer to the question

Would you apply the proposed refactoring?

assigning a score on a five point Likert scale: 1 (definitely not), 2 (no), 3 (maybe), 4 (yes), and 5 (absolutely yes). Note that the chosen Likert scale is different from the experiment with the external developers by choice. In fact, for external developers without appropriate system domain knowledge, the difference between a *definitely not* and a *no* or between a *yes* and an *absolutely yes* would have been too difficult to define. Thus, we preferred a simpler three points Likert scale. In this study with original developers, their experience on the two systems should allow a finer evaluation of which refactoring suggestions are valuable to apply in the systems.

The number of refactoring operations suggested by the two approaches (and thus, evaluated by the participants) is reported in Table VII. As we can see the number is rather high for both the systems, and thus we gave ten days to the participants to evaluate all the refactoring operations. Note that on both systems there was no overlap between Methodbook's and JDeodorant's recommendations. Each participant filled-in two questionnaires, i.e., one with Methodbook's suggestions and one with JDeodorant's suggestions, independently. After that, all the participants involved in the development of each system performed a review meeting to discuss their scores and reach a consensus. More specifically, in the meeting of the SMOS team there was full agreement by developers on the score to assign for 59 out of the 69 JDeodorant's suggestions (86%) and on 24 out of the 27 Methodbook's suggestions (89%). While in the GESA meeting participants fully agreed for 161 out of the 169 JDeodorant's suggestions (95%) and for 28 out of the 30 Methodbook's suggestions (93%). In the few remaining cases the score to assign was decided by the majority of developers in each system. At the end of the meeting the participants provided only one filled-in questionnaire reporting their comprehensive evaluation.

The goal of the process described above was to maximize the benefits derived by involving the original developers in the evaluation of the refactoring suggestions. In fact, each of the involved participants just developed a specific portion of the system and thus her/his experience was particularly suited to evaluate refactoring suggestions on that specific part of the system. By performing a review meeting, the participants having better knowledge of particular code components (e.g., classes) could explain the reasons why a refactoring operation was meaningful or not to the other participants. Thus, since our goal in this study was to gather as much qualitative feedback as possible, we preferred to have less data (i.e., aggregated scores instead of individual scores) but derived by a full knowledge of the system. Also, we asked the developers to comment on some particular cases.

5.2.2 Analysis of the Results

Table VIII summarizes the answers of the participants to the question *"Would you apply the proposed refac-*

16. The recommendations generated by both the approaches are available online [52].

TABLE 8: Participants' answers to the question *"Would you apply the proposed refactoring?"*

		definitely not	*no*	*maybe*	*yes*	*absolutely yes*
JDeodorant	GESA (165 suggestions)	24%	7%	57%	11%	1%
	SMOS (69 suggestions)	6%	12%	59%	22%	1%
	Overall (234 suggestions)	**19%**	**8%**	**58%**	**14%**	**1%**
Methodbook	GESA (30 suggestions)	12%	30%	14%	37%	7%
	SMOS (27 suggestions)	11%	15%	37%	30%	7%
	Overall (57 suggestions)	**11%**	**22%**	**26%**	**34%**	**7%**

toring?". Concerning the GESA software system the developers gave a positive answers to 12% of the operations suggested by JDeodorant (11% *yes* + 1% *absolutely yes*) against a 44% achieved by Methodbook (37% *yes* + 7% *absolutely yes*).

On the "negative answers side" 31% of JDeodorant's suggestions (24% *definitely not* + 7% *no*) were discarded by participants against 42% of Methodbook's suggestions (12% *definitely not* + 30% *no*). Finally, there is a huge percentage (57%) of JDeodorant's refactorings marked with *maybe* by the developers against a 14% achieved by Methodbook.

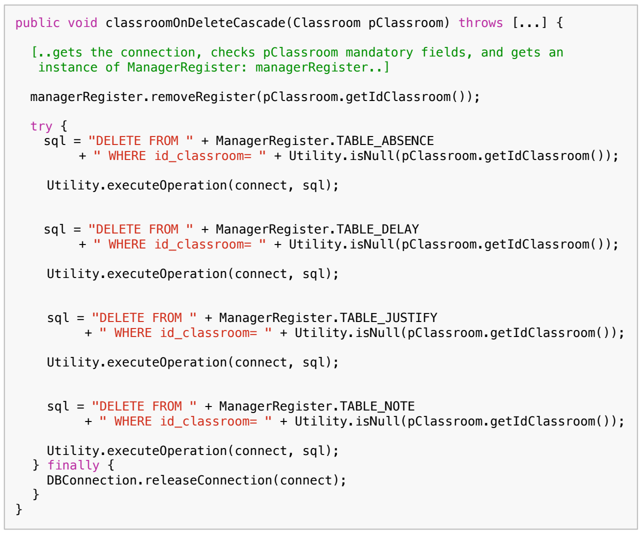
Thus, despite the average low confidence level of Methodbook's suggestions on GESA (0.31 with only one suggestion having confidence level higher than 0.6), the original developers accepted (through a *yes* or a *absolutely yes* answer) a much higher percentage of Methodbook's suggestions than of JDeodorant's ones. On the other side, Methodbook also received a higher percentage of "rejected operations" that, however, had a very low confidence level (0.14 on average) as compared to the "accepted operations" (0.53 on average). This confirms the goodnesses of the confidence level as indicator of the quality of the Methodbook's recommendations.

In order to get a deeper view of the achieved results we asked GESA developers to comment on some of their decisions. One of the refactoring operations suggested by Methodbook that the developers would *absolutely* apply is moving the method `executeOperation(Connection pConnect, String pSql)` from its class `Utility` to the envied class `ControlConnection`. Thus, we asked them to comment on the rationale behind these refactoring operations. The developers explained that the method `executeOperation` is in charge to execute a given query (the parameter `pSql`) in the database by using an existing connection to it (the parameter `pConnect`). The class containing this method, i.e., `Utility`, groups together miscellaneous services that (i) can be useful for different classes in the system, e.g., convert a date in SQL format, and (ii) have not a clear collocation in other classes of the system. However, GESA also contains a class implementing all the operations needed to exchange data with the database, that is the class `ControlConnection`. Thus, the developers felt that the envied class identified by Methodbook was a better place to implement the method `executeOperation`.

An example of Methodbook's suggestion that the participants would *not* apply is the move of the method `daysBetween(Date pDate1, Date pDate2)` from the class `Utility` to the class `ServletExportTimetableStampToPdf`. In fact, `daysBetween` represents a clear example of the kind of methods that should be placed in the `Utility` class (it computes the number of days between two given dates and such method could be used in the future by other classes). The wrong suggestion of Methodbook was the result of the high number of calls that the `ServletExportTimetableStampToPdf` class performs to this method. This is a clear example of move method refactoring that improve the software system from the point of view of metrics, but that is not meaningful from the point of view of developers.

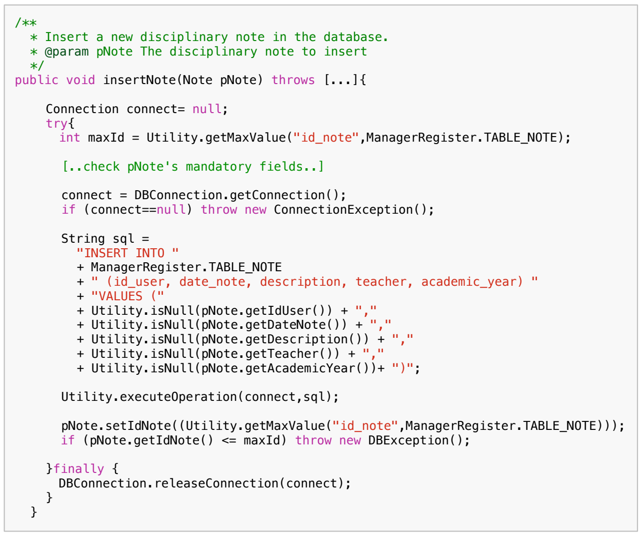
Concerning the SMOS software system, the developers accepted 23% of the operations suggested by JDeodorant (22% *yes* + 1% *absolutely yes*) against 37% achieved by Methodbook (30% *yes* + 7% *absolutely yes*). As for the "rejected" refactoring operations, 18% of the JDeodorant's suggestions (6% *definetly not* + 12% *no*) and 26% of the Methodbook's suggestions (11% *definetly not* + 15% *no*) were classified as bad. Finally, also in this case a high percentage of JDeodorant's suggestions were classified with *maybe* (59%) against 37% of Methodbook. Note that also on this system the average confidence level of the SMOS suggestions is quite low (0.42) with only 3 move method operations recommended with a confidence level higher than 0.6. However, also on SMOS the confidence level provides a good indication of the Methodbook's suggestion quality. In fact, the "rejected suggestions" provided by Methodbook have an average confidence level of 0.18, against 0.60 of the "accepted suggestions".

Like for the study with GESA developers, we asked SMOS developers to comment on some of their decisions. Figure 6 shows the method `classroomOnDeleteCascade` moved by Methodbook from the class `ManagerClassroom` to the class `ManagerRegister`. All SMOS developers agreed that this refactoring should be *absolutely* applied. Indeed, this method is invoked when a classroom from the system is deleted in order to remove all the related information from the database. As we can see, the information related to a classroom are mostly con-



```
public void classroomOnDeleteCascade(Classroom pClassroom) throws [...] {

  [..gets the connection, checks pClassroom mandatory fields, and gets an
   instance of ManagerRegister: managerRegister..]

  managerRegister.removeRegister(pClassroom.getIdClassroom());

  try {
    sql = "DELETE FROM " + ManagerRegister.TABLE_ABSENCE
         + " WHERE id_classroom= " + Utility.isNull(pClassroom.getIdClassroom());

    Utility.executeOperation(connect, sql);


    sql = "DELETE FROM " + ManagerRegister.TABLE_DELAY
         + " WHERE id_classroom= " + Utility.isNull(pClassroom.getIdClassroom());

    Utility.executeOperation(connect, sql);


    sql = "DELETE FROM " + ManagerRegister.TABLE_JUSTIFY
         + " WHERE id_classroom= " + Utility.isNull(pClassroom.getIdClassroom());

    Utility.executeOperation(connect, sql);


    sql = "DELETE FROM " + ManagerRegister.TABLE_NOTE
         + " WHERE id_classroom= " + Utility.isNull(pClassroom.getIdClassroom());

    Utility.executeOperation(connect, sql);
  } finally {
    DBConnection.releaseConnection(connect);
  }
}
```

Fig. 6: The method `classroomOnDeleteCascade` was moved by Methodbook from its class `ManagerClassroom` to the envied class `ManagerRegister`



```
/**
  * Insert a new disciplinary note in the database.
  * @param pNote The disciplinary note to insert
  */
public void insertNote(Note pNote) throws [...]{

    Connection connect= null;
    try{
      int maxId = Utility.getMaxValue("id_note",ManagerRegister.TABLE_NOTE);

      [..check pNote's mandatory fields..]

      connect = DBConnection.getConnection();
      if (connect==null) throw new ConnectionException();

      String sql =
        "INSERT INTO "
        + ManagerRegister.TABLE_NOTE
        + " (id_user, date_note, description, teacher, academic_year) "
        + "VALUES ("
        + Utility.isNull(pNote.getIdUser()) + ","
        + Utility.isNull(pNote.getDateNote()) + ","
        + Utility.isNull(pNote.getDescription()) + ","
        + Utility.isNull(pNote.getTeacher()) + ","
        + Utility.isNull(pNote.getAcademicYear())+ ")";

      Utility.executeOperation(connect,sql);

      pNote.setIdNote((Utility.getMaxValue("id_note",ManagerRegister.TABLE_NOTE)));
      if (pNote.getIdNote() <= maxId) throw new DBException();

    }finally {
      DBConnection.releaseConnection(connect);
    }
  }
```

Fig. 7: The method `insertNote` was moved by JDeodorant from its class `ManagerRegister` to the envied class `Note`

cerned with the class register, e.g., students' absences. Thus, the method `classroomOnDeleteCascade` invokes several times the class `ManagerRegister` that, for this reason, is identified by Methodbook as envied class. Note that, besides several queries executed on the database to delete the class register's information, `classroomOnDeleteCascade` also invokes the method `removeRegister` of the class `ManagerRegister` (first instruction in Figure 6). The latter updates a Vector of Integer stored in `ManagerRegister` and aimed at keeping track of all the classes for which a register exists. This is an optimization done to avoid querying the database each time the list of registers must be shown to the SMOS's users. In particular, the `removeRegister` method removes the id of the deleted classroom from that vector.

There are also three move method refactoring operations suggested by Methodbook with a confidence level lower than 0.2 that were rejected by the developers. For these operations, the developers did not find any explanation, confirming that when the confidence level is too low, the Methodbook's suggestions are generally not recommended to be applied.

Finally, we also asked both GESA and SMOS developers to comment on the high number of JDeodorant move method suggestions answered with a *maybe* (135 out of 234). The explanation was quite simple. In both GESA and SMOS there is a clear separation between the entity objects of the systems (e.g., user, classroom), that are implemented through specific java bean classes (e.g., `User`, `Classroom`), and the control classes managing that objects (e.g., `ManagerUser`, `ManagerClassroom`). JDeodorant suggests to move several of the methods present in each control class to the corresponding entity object. While these move methods will result in improving quality metrics, they are only considered as an alternative to the original design by the developers that generally prefer their choice of separating entity and control objects in the systems. This set of move method operations suggested by JDeodorant also explains (i) the very high number of suggestions on these two systems and (ii) the very good performances achieved by it on GESA and SMOS in the software metrics evaluation reported in Section IV. For example, one of the JDeodorant's suggestions falling in these cases is the move of the method `insertNote` of the SMOS system, reported in Figure 7, from its class `ManagerRegister` to the `Note` entity object. This method is in charge of storing—into the database—a new disciplinary note embedded in the `Note` object. JDeodorant identifies this method as a Feature Envy bad smell mainly for the fact that `insertNote` updates the state of the object `Note` when invoking its setter method `setIdNote` (see bottom part of Figure 7). However, this is just done to check if the performed operation (i.e., the insertion of the disciplinary note) was successfully executed. In fact, in the last `if` statement, it is checked if the id of the new note stored in the database (the auto increment field `id_note` of table `ManagerRegister.TABLE_NOTE`) is higher than the maximum id present in the table before the addition of the new note (stored in the variable `maxId` in the first instruction of the `try` block). This example reflects exactly what a bad smell is about: it is a symptom in the code that may (or may not) indicate a design problem. In this case, from the original developers' point of view, it does not indicate a design problem and thus, even recognizing as reasonable the JDeodorant's suggestion (*maybe* score), they prefer to keep the current methods' organization.

In conclusion, Methodbook's suggestions were generally preferred to those by JDeodorant (41% of accepted suggestions for Methodbook, 15% for JDeodorant) on both systems. In particular, participants generally appreciated Methodbook's suggestions having a confidence level higher than 0.5. Despite this, also in this study JDeodorant was able to identify a high number of appreciated refactoring suggestions ignored by Methodbook (overall 35). This result confirms that JDeodorant is less conservative than Methodbook, finding several good suggestions surrounded, however, by more noisy suggestions as compared to Methodbook. Finally, it is worth noting that all the suggestions (by both Methodbook and JDeodorant) evaluated with *yes* and *absolutely yes* by developers have been applied on GESA and are part of its new release in operation at the University of Molise. On SMOS this was not done since it is not in operation.

6 Threats to Validity

This section describes some threats that could affect the validity of the results achieved in our studies [56].

6.1 Evaluation based on Metrics

A first possible threat affecting the validity of the results is related to the choice of the employed quality metrics. We evaluated the goodness of the Methodbook's suggestions from a quality metrics point of view through two cohesion (i.e., Connectivity and C3) and two coupling (i.e., MPC and CCBC) metrics. Since the choice of the metrics could strongly influence the results, we carefully selected them. Firstly, for both cohesion and coupling we employed one structural and one semantic metric in order to have a complete picture of the changes obtained in the system quality by applying the suggested move method operations. Among the structural metrics, Connectivity and MPC were preferred for several reasons: (i) unlike other structural cohesion metrics, e.g., Lack of Cohesion of Methods (LCOM) [36], Connectivity considers two methods to be cohesive not only if they share an instance variable, but also if they have a call among them, (ii) MPC is able to capture coupling at a finer granularity level (i.e., method-calls interaction) compared to other coupling metrics, e.g., Coupling Between Object classes (CBO) [36], and (iii) the same structural metrics were also used in the evaluation of JDeodorant [9]. To the best of our knowledge, on the semantic side the C3 metric is the only semantic cohesion metric available in the literature while the CCBC was preferred to the semantic coupling metric presented by Gethers and Poshyvanyk [38], since the latter is based on RTM, which represents the foundation of Methodbook.

A second threat for this evaluation is due to the tool used to measure the quality metrics. The values of all used measurements are computed by the tool we developed. We observed as the MPC and Connectivity values measured by our tool are different from those reported by Tsantalis and Chatzigeorgiou [9] for jEdit and JFreeChart during the JDeodorant evaluation. However, the improvement trend achieved by JDeodorant in our experimentation and in the original evaluation is exactly the same for both metrics and on both systems (compare our graphs with Figures 9 and 10 in the paper by Tsantalis and Chatzigeorgiou [9]). Thus, we are confident that the findings of our study are correct.

In our study we observed changes in cohesion and coupling obtained by applying the refactoring suggestions generated by Methodbook and JDeodorant. When considering the improvements of cohesion and coupling in terms of percentage, they are limited to few points. For example, on the jEdit system (see Figure 10) Methodbook is able to improve the Connectivity, MPC, and CCBC metrics of about 2%, and the C3 metric of about 1%. These results might seem very marginal. However, it is worth noting that, as shown in Table II, the number of methods moved by both Methodbook and JDeodorant just represent a very small percentage of the system's methods (i.e., around 2% for Methodbook and 4% for JDeodorant). For example, on jEdit Methodbook just suggests to move 8 out of the 1,864 existing methods, i.e., 0.05%. Thus, it is important to put the observed improvements in the context of move method refactoring approaches, where it is very unlikely to obtain strong changes in cohesion and coupling.

It is also important to acknowledge the limitations of an evaluation based on quality metrics. The achieved results indicate that Methodbook is able to improve the quality of the object systems in terms of class cohesion and coupling. Moreover, on three out of the six employed systems it is able to perform better than the state-of-the-art tool JDedorant [9]. While this result can be encouraging it is not enough to state superiority of the move method operations suggested by Methodbook. In fact, the refactoring operations suggested by a tool should not only improve the value of some quality metrics but, more importantly, be meaningful from the point of view of developers. For this reason, we not only based our evaluation on metrics measurement, but also performed two user studies reported in Section V.

Finally, while we analyzed the benefits of applying the refactoring recommendations generated by Methodbook and JDeodorant (i.e., improvements in quality metrics), we did not assess the cost of applying them (i.e., what is the impact on the system source code). Generally, the application of a refactoring moving a method m_i from its class C_j to the envied class C_k is limited to the update of the calls performed by other methods to m_i, that is not placed in C_j anymore, but in C_k. This can be easily automated by using, for example, the Eclipse AST (as currently done in JDeodorant). Also, since Methodbook and JDeodorant verify the same behavior-preserving preconditions before recommending a move method refactoring, we expect the cost of applying their suggestions to be roughly the same. To verify this and to have an indication of the effort needed to apply the generated recommendations, we compute the number of method calls that should be updated in consequence of all refactoring operations suggested by Methodbook and JDeodorant on JFreeChart. On average, the 18 JDeodorant's recommendations require to fix 1.5 method calls (min=1, max=3), while the 10 Methodbook's recommendations require to fix 1.4 method calls (min=1, max=2). Thus, as expected, applying the refactoring recommendations by Methodbook and JDeodorant generally require a similar (low) effort.

6.2 Evaluation with External Developers

As for the generalization of the results achieved in our experiment conducted with external developers, the population of the participants represents the main threat. As previously explained, the less experienced participants involved in our study were third year Bachelors students that in the context of the Software Engineering course had participated in software projects, where they practiced software development and documentation production. As a check, we statistically compared the scores assigned by Bachelors students with those assigned by the other kinds of participants to verify if less experienced participants were less/more prone to accept/reject the evaluated refactoring suggestions (both the Methodbook and the JDeodorant ones). We used the Wilcoxon test [53] by adjusting the obtained p-values using the Holm's correction procedure [54]. The results showed no statistically significant difference between the ratings assigned by Bachelors students and those assigned by the other kinds of participants on both systems (p-value always higher than 0.05). This result somehow confirms what has been highlighted by Arisholm and Sjoberg [57]: the difference between students and professionals is not always easy to identify.

Another threat is represented by the lack of system domain knowledge by the participants. However, we also executed the experiment with original developers where this threat is not present.

6.3 Evaluation with Original Developers

In our second user study we were able to involve the original developers of two systems, i.e., GESA and SMOS, in the evaluation of the refactoring operations suggested by Methodbook and JDeodorant. Thus, the type of participants involved in our two user studies is quite different, allowing a good generalization of the achieved results. On the other side, it is worth noting that both GESA and SMOS are *Java EE* web applications, thus our findings might be in part due to the specific nature of these systems. However, the systems used in the evaluation with external developers (i.e., jEdit and JFreeChart) are standard Java applications, thus mitigating this threat.

The system domain knowledge could also represent a threat in this experiment but in a different way. In fact, as explained before, some of the participants could be the "fathers" of some bad design choices and consequently not recognize a good move method refactoring as meaningful. However, the results obtained and the deep discussion with them about some of the good suggestions provided by the two approaches demonstrate that the participants provided an objective evaluation of the analyzed move method operations.

Finally, the suggestions generated by Methodbook and JDedodorant were verbatim copied in the questionnaire to exclude biases derived by the participants ability with the tools interfaces. Participants used their preferred IDE to analyze the source code of the object systems.

7 Conclusion

This paper proposed and evaluated Methodbook, an approach to automate Move Method refactoring. Methodbook uses RTM to analyze both structural and textual information gleaned from software to suggest move method refactoring operations.

We evaluated Methodbook in two case studies comparing its performance with the state-of-the-art tool JDeodorant. In the first case study we analyzed if move method suggestions produced by Methodbook are able to improve the design of five software systems from a metrics point of view. The results indicate that Methodbook's suggestions having high confidence level (generally higher than 0.60) are able to improve cohesion and coupling of the object systems, while suggestions having low confidence level do not show a very stable trend. Moreover, on three out of five experimented systems Methodbook outperforms JDeodorant. In the second case study we evaluated the refactoring recommendations by Methodbook in two user studies, one conducted with ten original developers of two software systems and one with seventy academic and industrial software developers on two open source software systems.

The results indicate that Methodbook's refactoring recommendations having high confidence level are meaningful from a developer's point of view, supporting the potential usefulness of Methodbook in integrated development environments.

The extensive evaluation conducted for Methodbook provided us with a number of lessons learned. First, we noticed that the confidence level seems to be crucial for the suggested refactoring operations. In fact, the confidence level turned out to be a very good indicator of the goodness of the suggested refactoring operations in all the performed evaluations. We now know that developers can mostly ignore the Methodbook's suggestions having a low confidence level (i.e., < 0.5), since the likelihood of having a meaningful suggestion with such a confidence level is quite low.

The comparison performed with JDeodorant highlighted that Methodbook is generally more precise than JDeodorant, providing less suggestions to the developers of an average higher quality. However, the results also clearly highlighted as JDeodorant is able to identify good refactoring operations that are missed by Methodbook. This means that there is still room for improvement and that maybe some aspects of the two tools could be combined together in a new approach trying to exploit the strengths of both tools. This is part of our future agenda.

The evaluation conducted with the original developers highlighted how some refactoring operations, even if reasonable, do not justify the need to change the original design from the developer's point of view. In total, 136 JDeodorant's suggestions and 15 Methodbook's suggestions evaluated with a *maybe* by participants fall among these. From the study based on quality metrics we know that these operations were able to improve cohesion and coupling of the object systems, as it is particularly evident by the excellent performances reached by JDeodorant on GESA (see Figure 8) and SMOS (see Figure 4). These cases highlight the fact that an evaluation of a refactoring technique based only on software quality metrics is not sufficient and it should always be complemented with experiments performed with software developers in order to get real insights about the actual value of the technique. These observations also pinpoint that, even if better refactoring tools might be developed in the future, the final word about any refactoring operation should be left up to developers, discouraging the implementation of fully automated refactoring tools.

Appendix

Figures 8, 9, 10, and 11 report the quality metrics trends for GESA, eXVantage, jEdit, and JFreeChart respectively.

Acknowledgments

We would like to thank all the students and professional software developers who participated in the Methodbook's evaluation. We are also very grateful to Nikolaos Tsantalis for the support provided us in the computation of the JDeodorant's suggestions. Finally, we would also like to thank anonymous TSE reviewers for their careful reading of our manuscript and high-quality feedback.

References

[1] W. Stevens, G. Myers, and L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[2] R. Pressman, *Software Engineering: A Practitioner's Approach. 3rd Edition*. McGraw-Hill, 1992.

[3] I. Sommerville, *Software Engineering. 6th Edition*. Addison-Wesley, 2001.

[4] V. R. Basili, L. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1995.

[5] A. B. Binkley and S. R. Schach, "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures," in *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, 1998, pp. 452–455.

[6] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[7] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. John Wiley and Sons, 1998.

[8] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[9] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, pp. 347–367, RapidPost.

[10] J. Chang and D. M. Blei, "Hierarchical relational models for document networks," *Annals of Applied Statistics*, 2010.

[11] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*. Lisbon, Portugal: IEEE CS Press, 2001, pp. 30–38.

[12] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proceedings of the Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA, 2006, pp. 1909–1916.

[13] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 287–297.

[14] Y. Wang, "What motivate software engineers to refactor source code? evidences from professional developers," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 2009, pp. 413 –416.

[15] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the 20th International Symposium on Foundations of Software Engineering*, November 2012.

[16] K. Taneja, D. Dig, and T. Xie, "Automated detection of api refactorings in libraries," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07, 2007, pp. 377–380.

[17] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen, "Effective software merging in the presence of object-oriented refactorings," *Software Engineering, IEEE Transactions on*, vol. 34, no. 3, pp. 321 –335, may-june 2008.

[18] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," in *Proceedings of the 12th Working Conference on Reverse Engineering*. Pittsburgh, PA, USA: IEEE Press, 2005, pp. 155–164.

[19] P. Joshi and R. K. Joshi, "Concept analysis for class cohesion," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, Kaiserslautern, Germany, 2009, pp. 237–240.

[20] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the 2009 Ninth International Conference on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 305–314.

[21] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code

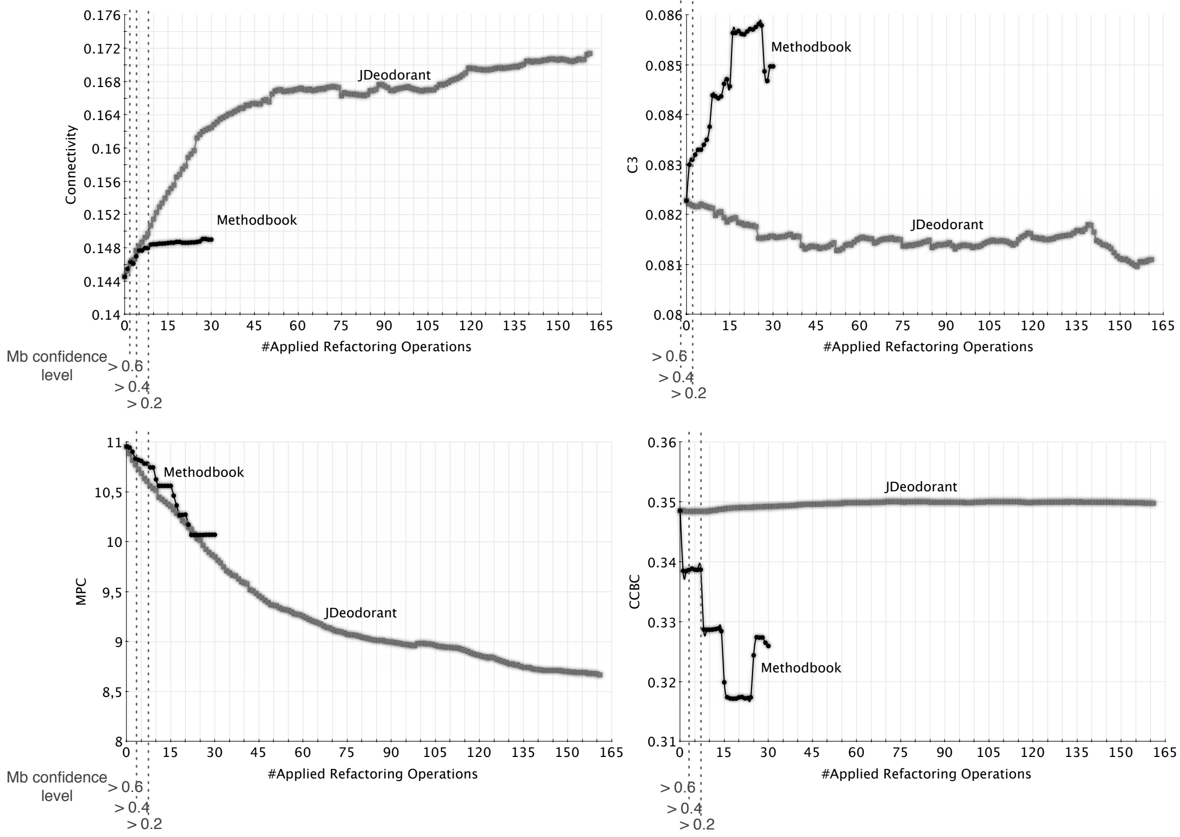


Fig. 8: Evolution of the four quality metrics on GESA by applying the refactoring operations suggested by Methodbook (30) and JDeodorant (165)

and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[22] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[23] O. Maqbool and H. A. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.

[24] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.

[25] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[26] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *Transactions on Software Engineering and Methodologies*, Accepted on Jan 2013.

[27] K. Maruyama and K. Shima, "Automatic method refactoring using weighted dependence graphs," in *Proceedings of 21st International Conference on Software Engineering*. Los Alamitos, California, USA: ACM Press, 1999, pp. 236–245.

[28] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique," in *Proceedings of the 25th International Conference on Software Maintenance*, Edmonton, Canada, 2009, pp. 93–101.

[29] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, pp. 397–414, 2011.

[30] G. Bavota, R. Oliveto, A. D. Lucia, G. Antoniol, and Y.-G. Guéhéneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *Proceedings of the 26th IEEE International Conference on Software Maintenance*, 2010.

[31] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "A two-step technique for extract class refactoring," in *Proceedings of 25th IEEE International Conference on Automated Software Engineering*, 2010, pp. 151–154.

[32] M. O'Keeffe and M. O'Cinneide, "Search-based software maintenance," in *Proceedings of 10th European Conference on Software Maintenance and Reengineering*. Bari, Italy: IEEE CS Press, 2006, pp. 249–260.

[33] A. Abadi, R. Ettinger, and Y. A. Feldman, "Fine slicing for advanced method extraction," in *3rd Workshop on Refactoring Tools*, 2009.

[34] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. ACM, 2008, pp. 421–430.

[35] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, Accepted on Apr 2013.

[36] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994.

[37] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *The Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[38] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *ICSM*, 2010, pp. 1–10.

[39] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *In Proceedings of 27th IEEE International Conference on Software Maintenance*, 2011, pp. 133–142.

[40] S. Bajracharya and C. Lopes, "Mining search topics from

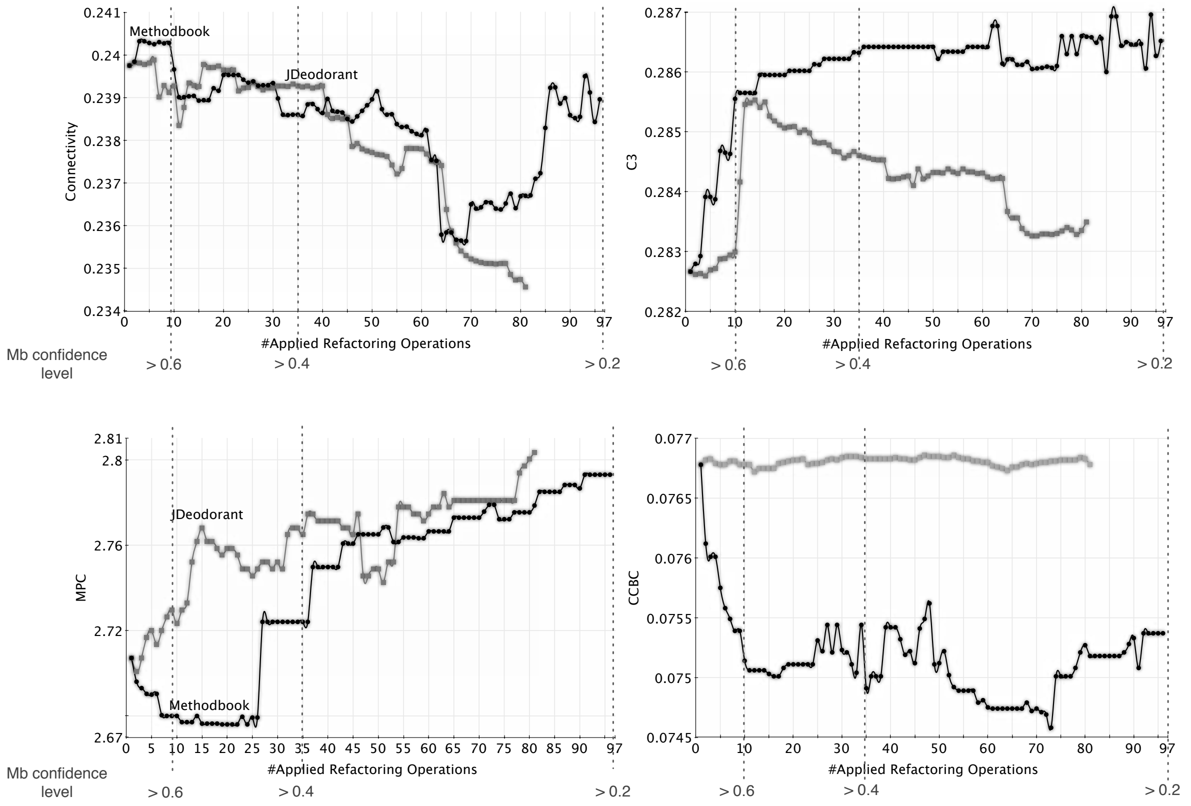


Fig. 9: Evolution of the four quality metrics on eXVantage by applying the refactoring operations suggested by Methodbook (95) and JDeodorant (80)

a code search engine usage log," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–120. [Online]. Available: http://dx.doi.org/10.1109/MSR.2009.5069489

[41] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides, "Modeling class cohesion as mixtures of latent topics," in *Proceedings of 25th IEEE International Conference on Software Maintenance*. Edmonton, Canada: IEEE CS Press, 2009, pp. 233–242.

[42] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Codetopics: Which topic am i coding now?" in *Proceedings of 33rd IEEE/ACM International Conference on Software Engineering*. Honolulu, Hawaii, USA: ACM Press, 2011.

[43] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proceedings of the 35th IEEE/ACM International Conference on Software Engineering*, ser. ICSE'13. IEEE Computer Society, 2013, pp. 522–531.

[44] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *Proceedings of 19th IEEE International Conference on Program Comprehension*. Kingston, Canada: IEEE CS Press, 2011.

[45] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[46] G. Gui and P. D. Scott, "Coupling and cohesion measures for evaluation of component reusability," in *Proceedings of the 5th International Workshop on Mining Software Repositories*. Shanghai, China: ACM Press, 2006, pp. 18–21.

[47] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Wiley-Interscience, 1991.

[48] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-orientedsystems," *Empirical Software Engineering.*, vol. 3, pp. 65–117, July 1998.

[49] *Maintenance metrics for the object oriented paradigm*, 1993.

[50] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[51] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[52] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending move method refactorings via relational topic models," University of Sannio, http://www.dmi.unisa.it/people/bavota/www/reports/methodbook/, Tech. Rep., 2013.

[53] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[54] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[55] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[56] R. K. Yin, *Case Study Research: Design and Methods*, 3rd ed. SAGE Publications, 2003.

[57] E. Arisholm and D. Sjoberg, "Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 521–534, 2004.

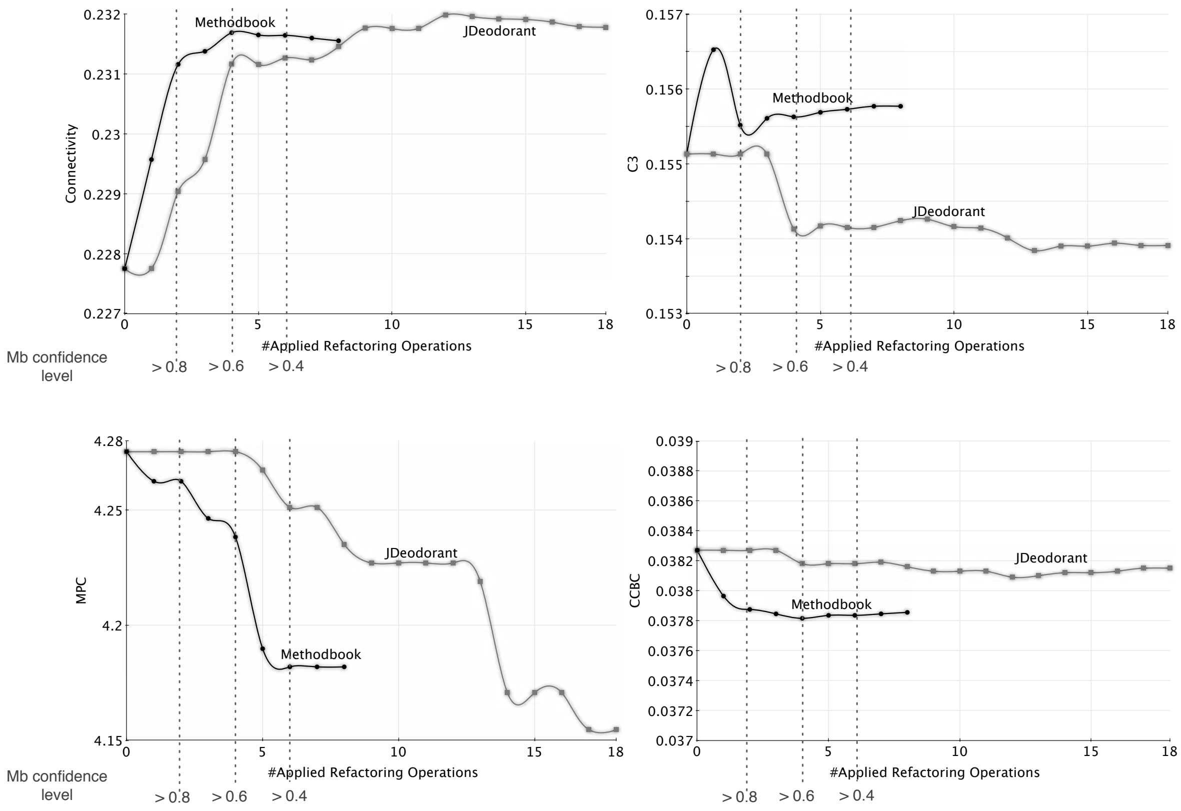


Fig. 10: Evolution of the four quality metrics on jEdit by applying the refactoring operations suggested by Methodbook (8) and JDeodorant (18)

Gabriele Bavota is a research fellow at the Department of Engineering of the University of Sannio (Italy). He received the Ph.D. in Computer Science from the University of Salerno (Italy) in 2013. His research interests include refactoring and remodularization, software maintenance and evolution, and empirical software engineering. He serves and has served on the organizing and program committees of international conferences in the field of software engineering. He is member of IEEE.

Rocco Oliveto is Assistant Professor in the Department of Bioscience and Territory at University of Molise (Italy). He is the Director of the Laboratory of Computer Science and Scientific Computation of the University of Molise. He received the PhD in Computer Science from University of Salerno (Italy) in 2008. His research interests include traceability management, information retrieval, software maintenance and evolution, search-based software engineering, and empirical software engineering. He serves and has served as organizing and program committee member of international conferences in the field of software engineering. He is a member of IEEE Computer Society, ACM, and IEEE-CS Awards and Recognition Committee.

Malcom Gethers is an Assistant Professor in the Department of Information Systems at the University of Maryland, Baltimore County (UMBC). Malcom obtained his Ph.D. in Computer Science from the College of William and Mary in 2012 where he was a member of the SEMERU research group. He was advised by Dr. Denys Poshyvanyk. Malcom obtained his B.S. from High Point University and his M.S. from the University of North Carolina at Greensboro. His research interests include software engineering, software maintenance and evolution, mining of software repositories, feature location, software measurement, and traceability link recovery and management. He is a member of the IEEE and ACM.

Denys Poshyvanyk is an Assistant Professor at The College of William and Mary in Virginia. He received his Ph.D. degree in Computer Science from Wayne State University in 2008. He also obtained his M.S. and M.A. degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine and Wayne State University in 2003 and 2006, respectively. His research interests are in software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. He is a member of the IEEE and ACM.

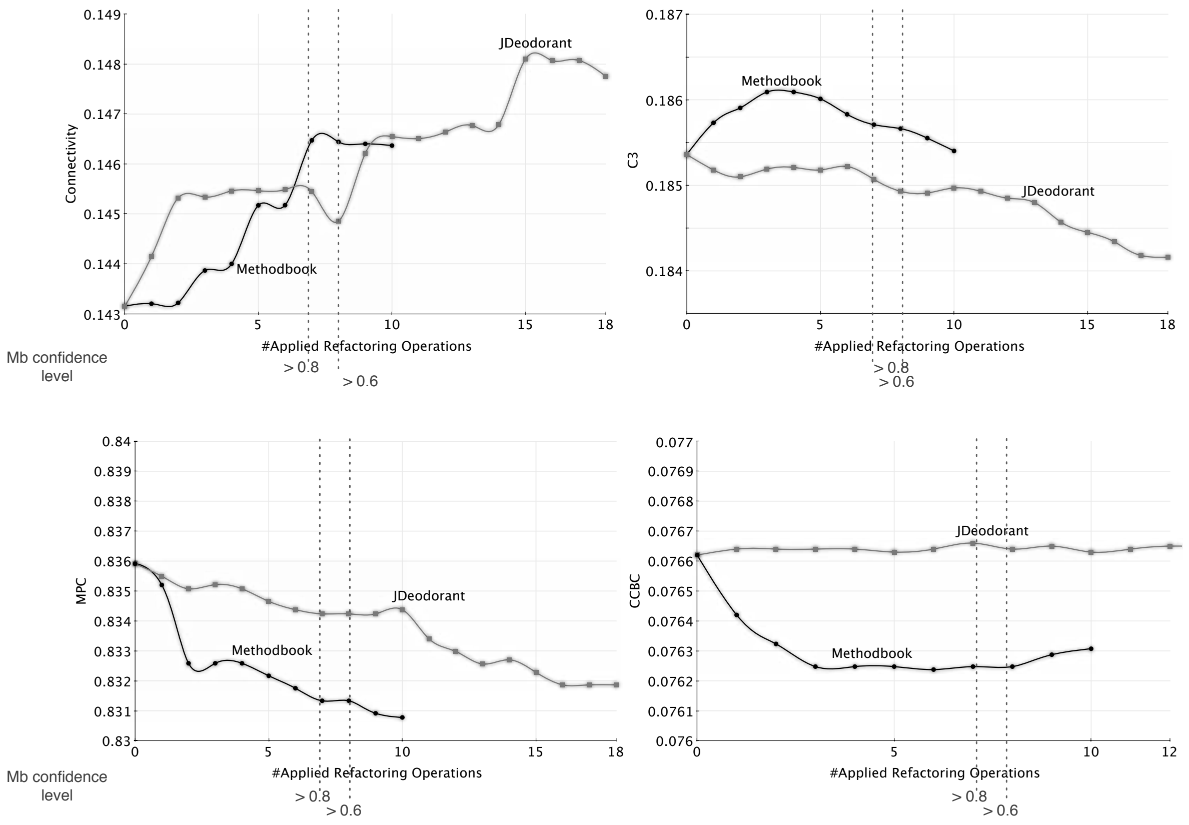


Fig. 11: Evolution of the four quality metrics on JFreeChart by applying the refactoring operations suggested by Methodbook (10) and JDeodorant (18)

Andrea De Lucia is a full professor of software engineering at the Department of Management & Information Technology of the University of Salerno, Italy, head of the Software Engineering Lab, and Director of the International Summer School on Software Engineering. He received his PhD in Electronic Engineering and Computer Science from the University of Naples ”Federico II”, Italy, in 1996. His research interests include software maintenance and testing, reverse engineering and reengineering, empirical software engineering, sarch-based software engineering, collaborative development, workflow and document management, and e-learning. He has published more than 200 papers on these topics in international journals, books, and conference proceedings and has edited books and jornal special issues. He also serves on the editorial boards of international journals and on the organizing and program committees of international conferences. Prof. De Lucia is a senior member of the IEEE and the IEEE Computer Society and was also at-large member of the executive committee of the IEEE Technical Council on Software Engineering (TCSE).