Università degli Studi di Salerno
Dipartimento di Matematica ed Informatica
Dottorato di Ricerca in Scienze Matematiche, Fisiche ed Informatiche
Curriculum in Informatica
XI Ciclo

Tesi di dottorato in:
Using Structural and Semantic Information to Support Software Refactoring

Anno Accademico 2013

PhD Candidate

Gabriele Bavota

PhD Program Coordinator

Prof. Patrizia Longobardi

Advisor

Prof. Andrea De Lucia

Advisor: Prof. Andrea De Lucia
Co-advisor: Dr. Rocco Oliveto

PhD Program Coordinator: Prof. Patrizia Longobardi

Day of the defense: February 7th, 2013

Acknowledgements

It would not have been possible to write this thesis without the help and support of many people during these three years of research. In this section I will attempt to thank you all. My sincere apologies to anyone inadvertently omitted.

Firstly, I would like to thank my advisor, Prof. Andrea De Lucia. He patiently provided me all the encouragement and advise necessary to proceed in my research program. Any discussion with him has always been an opportunity for learning something.

Special thanks to my co-advisor, Dr. Rocco Oliveto. Probably without him I would never start my phd. In one way or another, he is advising me by eight years becoming one of my best friends. Thanks for the hundreds of hours spent together in talking about our research projects and, more important, for the thousands spent in joking.

I am very grateful to Prof. Andrian Marcus for being actively involved in my work and providing me precious advice for my research activity. We started a strong collaboration that I hope to continue for a long time.

I would also thank Dr. Denys Poshyvanyk for the effort and enthusiasm he always bring when working together. The passion he puts in his work has made a deep impression on me.

Thanks to Prof. Max Di Penta for the work done together during my phd and for giving me the opportunity to work with him during my post-doc. I am glad to have this opportunity.

Thank to the SE@SA Lab group, and in particular to Dr. Carmine Gravino for the support and contribution provided to my research, to Annibale

Panichella for being a great colleague and friend with whom spend the days in the lab, and to Abdallah Qusef for all the work we made together.

I want to thank for the strong collaborations we had in these years: Sonia Haiduc, Malcom Gethers, Bogdan Dit, Dave Binkley, Giuliano Antoniol, and Yann-Gael Guhneuc.

Thank to my friends, and in particular to Marco Iannacone, Marco Scannapieco, Alessandro Pugliese, Paolo Sabatino, and Fabio Palomba, for sharing with me all these beautiful years in Salerno.

I want to thank my girlfriend Maura for the continuous encouragement she gave me in these years and for all the beautiful moments spent together. The best is yet to come.

Finally, I am very grateful to my parents and my sister for their never-ending support and encouragement.

Abstract

In the software life cycle the internal structure of the system undergoes continuous modifications. These changes push away the source code from its original design, often reducing its quality. In such cases refactoring techniques can be applied to improve the design quality of the system.

Approaches existing in literature mainly exploit structural relationships present in the source code, e.g., method calls, to support the software engineer in identifying refactoring solutions. However, also semantic information is embedded in the source code by the developers, e.g., the terms used in the comments.

This research investigates about the usefulness of combining structural and semantic information to support software refactoring. In particular, a framework of approaches supporting different refactoring operations, i.e., Extract Class, Move Method, Extract Package, and Move Class, is presented.

All the approaches have been empirically evaluated. Particular attention has been devoted to evaluations conducted with software developers, to understand if the refactoring operations suggested by the proposed approaches are meaningful from their point of view.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

1.1 Research problem

During software evolution change is the rule rather than the exception (3). A software system evolves as changes in the environment and requirements are incorporated in it. Unfortunately, due to strict deadlines programmers do not always have a bunch of time to make sure the applied changes conforms to good design practices. Thus, in consequence of changes, often software quality decreases, resulting in more and more difficulties in maintaining existing software (4).

In such cases a refactoring of the system is recommended since several empirical studies provide evidence that low design quality is generally associated with lower productivity, greater rework, and more significant efforts for developers (5, 6, 7, 8, 9).

Refactoring has been defined as "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" (3, 10). Different refactoring operations[1] might improve different quality aspects of a system. As an example, in Object-Oriented systems low-cohesive classes, i.e., classes implementing unrelated responsibilities, can be removed by splitting their methods into different classes that group together strongly related responsibilities and are easier to comprehend and maintain (this operation is known as Extract Class refactoring). Typical advantages of refactoring include improved readability and reduced complexity of source code, a more expressive internal architecture and better software extensibility (3). For this reasons refactoring is advocate as a good programming

[1] A complete refactoring catalog can be found at http://refactoring.com/catalog/.

practice to be continuously performed during software development and maintenance (3, 11, 12, 13). In fact, as explained by Kerievsky (12) "*by continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code*".

Despite its advantages, perform refactoring in non-trivial software systems might be very challenging. Firstly, the identification of refactoring opportunities in large systems is very hard, due to the fact that the design flaws to correct are not always under the sunlight (3). Secondly, when a design problem has been identified, it is not always easy to apply the correct refactoring operation to solve it. As an example, splitting a non-cohesive class into different classes with strongly related responsibilities (i.e., Extract Class refactoring) requires the analysis of all the methods of the original class to identify groups of methods implementing similar responsibilities and that should be clustered together in new classes to be extracted. This task becomes harder and harder when the size of the class to split increases. Moreover, even when the refactoring solution has been defined, the software engineer must apply it without changing the external behavior of the system. All these observations highlight the need for (semi)automatic approaches supporting the software engineer in (i) identifying refactoring opportunities (i.e., design flaws) and (ii) designing and applying a refactoring solution. For these reasons, a lot of effort has been devoted to the definition of automatic and semi-automatic approaches for software refactoring (14, 15, 16, 17, 18, 19, 20, 21). These are also the motivation for the increasing interest in this field by the software engineering community which led to the organization of international events focused on the refactoring topic, like the ICSE 2011 4th Workshop on Refactoring Tools (22).

1.2 Motivation

Although several approaches are available in the literature to support different refactoring operations, most of them only exploit structural information extracted from source code (e.g., method calls) to suggest refactoring solutions. However, also semantic (i.e., textual) information is present in source code and in particular in the terms used by the developers in the comments and identifiers.

In the last years semantic information extracted from software artifacts have been used to support a wide range of software engineering tasks, like software reuse (23, 24, 25, 26), recovery and management of traceability links (27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42), concept location (43, 44, 45, 46, 47, 48, 49, 50, 51, 52), impact analysis (53, 54, 55, 56), clone detection (57, 58, 59), and computation of software quality metrics like cohesion (60) and coupling (55).

Moreover, in a recent study we analyzed how developers actually perceive coupling and which among the existing coupling measures align with this perception (61). The results indicated that the peculiarity of the semantic coupling measure allows it to better estimate the mental model of developers than the other coupling measures. This is because, in several cases, the interactions between classes are encapsulated in the source code vocabulary, and cannot be easily derived by only looking at structural relationships, such as method calls (61).

All these successful applications of textual analysis in software engineering are in someway based on the conjecture that artifacts containing similar terms are likely to "talk" about similar things and thus, to be related. Since the goal of several refactoring operations is to group together similar things (e.g., Extract Class, Move Method), we are confident that semantic information can be useful to support these kinds of refactoring, allowing to achieve better refactoring recommendations as compared to the only use of structural measures (e.g., method calls). As an example, if the vocabulary of two methods is very similar, it is likely that the developers used similar terms to describe similar responsibilities implemented by the two methods. This information, combined with structural ones like method calls and/or shared instance variables, can be useful to support, for instance, Extract Class refactoring.

1.3 Research Contributions

In this thesis we present a framework of approaches to support four refactoring operations, namely Extract Class, Move Method, Extract Package, and Move Class.

All the approaches exploit a combination of structural and semantic information extracted from the source code to identify relationships existing between the code components (i.e., methods, classes) object of the refactoring. For instance, to support Extract Class we identify structural (method calls and shared instance variables) and

semantic (textual similarity) relationships between the methods of the class to be split. In this way it is possible to understand which methods of the original class implement similar responsibilities and thus, should be grouped together in new extracted classes.

When all relationships between code components are computed, a weighted-graph is used to represent this information. In particular, each node in the graph represents one of the code components under analysis (e.g., the methods of the class to be split in case of Extract Class refactoring) while the weight on the edge connecting two nodes represents the structural and semantic relationships between them.

Finally, depending on the supported refactoring operation, each of the four proposed approaches applies a particular algorithm on the weighted graph to identify the refactoring solution. The two approaches supporting refactoring operations aimed at decomposing complex objects (i.e., Extract Class and Extract Package), exploit an algorithm based on subgraph analysis. The complex object to refactor is modeled as a weighted graph and the algorithm identifies sets of strongly connected nodes (i.e., subgraphs) representing the simpler objects to extract. For instance, in case of Extract Class refactoring the complex object represented by the original graph is the class implementing several different responsibilities and having low cohesion, while the identified subgraphs are the classes that can be extracted from it. On the other side, the two approaches supporting refactoring operations dealing with the movement of code components (i.e., Move Method and Move Class), uses as algorithm the Relational Topic Models (RTM) (62), a probabilistic topic model technique for analyzing document networks, their topics, and known relationships among them. In these cases RTM is used to identify for a code component under analysis (i.e., a method for Move Method and a class for Move Class refactoring), its strongest relationships in the network represented by the weighted graph. Then, using this information, is easy to identify Move Method or Move Class refactoring opportunities. For instance, given a method m, RTM can be used to identify the set S of methods in the system having the strongest relationships with it. Then, the class containing the highest number of S's methods can be suggested as the best one to place m.

Summarizing, the main contributions of this thesis are:

1. An *Extract Class* refactoring approach able to split a low-cohesive class (i.e., a Blob class (63)) grouping together heterogeneous responsibilities into several

classes having high cohesion. The proposed approach has been experimented on real Blobs of open source systems to evaluate (i) how good the proposed solution is from a cohesion and coupling metrics point of view, (ii) how good the proposed refactoring solution is considered by software engineers as it is, (iii) how much the proposed solution is useful as starting point to perform a refactoring, and (iv) how well the proposed refactoring solution approximate a refactoring made by the original developers. The results show that (i) the refactoring solutions proposed by our approach strongly increases the cohesion of the refactored classes without leading to significant increases in terms of coupling, (ii) the refactoring solutions proposed by our approach are considered useful to developers performing extract class refactoring and (iii) the proposed approach is able to approximate a manually performed refactoring at 91% on average.

2. An *Extract Package* refactoring technique to remove from object oriented software systems promiscuous-low cohesive packages, decomposing them into smaller and meaningful packages having higher cohesion. The approach has been evaluated with developers on five software systems. The results show that our technique is able to identify meaningful refactoring operations from the developers' point of view.

3. A *Move Method* refactoring approach coined as Methodbook, able to identify for each method in an Object-Oriented system the best class in which it should be placed (i.e., the one grouping the responsibilities most similar to those implemented by it). We evaluated Methodbook in two case studies. The first study has been executed on five software systems to analyze if the move method operations suggested by Methodbook help to improve the design quality of the systems. The second study has been conducted with 40 developers that evaluated the refactoring recommendations produced by Methodbook. The results show that Methodbook is able to significantly improve cohesion and coupling of the subject systems and that it provides meaningful recommendations for move method refactoring from a developer's point of view.

4. A *Move Class* refactoring approach, coined as *R*3, that exploits RTM to analyze underlying latent topics in classes and packages as well as structural dependencies

to recommend refactoring operations aiming at moving classes to more suitable packages. *R*3 has been evaluated in two empirical studies. In the first study we analyzed the ability of *R*3 to propose refactoring operations that lead to reduced coupling among software modules in nine software systems. The results achieved indicated that *R*3 provides a strong coupling reduction among the software modules. Then, in a second study we evaluated *R*3 refactoring recommendations with developers in two case studies, one conducted with 14 original developers of four software systems and one with 44 students and academics plus 4 professional software developers on another open source software system. The results achieved in this second case study indicated that more than 70% of the recommendations provided by *R*3 are considered meaningful by developers.

The framework of approaches has also been implemented in an Eclipse plug-in called *ARIES* (Automated Refactoring In EclipSe).

1.4 Thesis organization

This thesis is composed of 9 chapters including this introduction. Chapter 2 discusses the state of the art and in particular approaches to automate refactoring operations and previous work using semantic information in software engineering.

Chapter 3 presents the framework, showing the sources of information used by our approaches, the problem representation, and the algorithms used to identify the refactoring solutions. Chapters from 4 to 7 presents the four proposed approaches and their evaluation in the following order: Extract Class refactoring (Chapter 4), Extract Package refactoring (Chapter 5), Move Method refactoring (Chapter 6), and Move Class refactoring (Chapter 7).

ARIES is presented in Chapter 8 while Chapter 9 gives conclusion remarks and directions for future work.

2

State of the Art

2.1 Introduction

This Chapter presents the state of the art on the areas of interests of this thesis. In particular, Section 2.2 discusses approaches existing in literature to support the software engineer when performing refactoring operations while Section 2.3 sheds light on the different usages of semantic (textual) information in software engineering.

2.2 Refactoring

There are more than 90 different refactoring operations in literature (64). Most of them are very simple source code transformations aimed at increasing source code comprehension. As example, Rename Method refactoring simply changes the name of a method in order to better reflect its purpose, while Remove Parameter removes a parameter not used anymore by the method body. Clearly, given their simplicity, researchers' efforts have been focused on providing support to developers when performing more complex refactoring operations, e.g., Extract Class (3). These latter are generally used to remove from the source code poor design solutions, known as *design antipatterns*. A design antipattern is "*something that looks like a good idea, but which back-fires badly when applied*" (65). It generally describes common pitfalls in object oriented programming that defeat good design rules like creating classes having high cohesion (i.e., grouping together strongly related responsibilities) and low coupling (i.e., having few dependencies among them) (66). Applying the correct refactoring operations it is possible to remove the design antipatterns increasing the internal software quality.

In the following we discuss the approaches existing in literature to support the different refactoring operations by focusing more in depth on approaches supporting the four tackled in this thesis, i.e., Extract Class, Move Method, Extract Package, and Move Class refactoring.

2.2.1 Extract Class refactoring

Extract Class refactoring is used to remove the Blob antipattern (63) from a software system. A Blob is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders, i.e., data classes. Implementing several different responsibilities, Blobs are generally characterized by low cohesion values. Moreover, due to the numerous dependencies with the data classes, Blobs also exhibit high levels of coupling.

Extract Class refactoring is applied to split the many responsibilities implemented in a Blob class into different classes having higher cohesion (i.e., grouping together strongly related responsiblities). The necessity of removing Blob classes from a system is due to the difficulty of performing comprehension and maintenance activities on them. In fact, several empirical studies provided evidence that lack of cohesion is generally associated with lower productivity, greater rework, and more significant design efforts for developers (5, 6, 7, 8, 9). In addition, classes with lower cohesion have been shown to correlate with higher defect rates (60, 67, 68). Given the complex nature of Blob classes, performing Extract Class refactoring is a very hard task since the following steps must be performed:

1. Analyzing the methods of the Blob class (that might be hundreds) trying to understand what is the main responsibility implemented by each of them;

2. Identifying clusters of cohesive methods that seem to implement very similar and related responsibilities;

3. Distributing the attributes of the Blob class among the identified clusters of methods, trying to assign each attribute to the cluster that uses it most;

4. Splitting the Blob class into the cohesive identified clusters of methods and attributes, representing the classes to be extracted;

Table 2.1: Trifu and Marinescu (2): metrics used to detect code smells

Metric	Description
AMW	The average statical complexity (i.e., McCabes cyclomatic number) of all methods in a class.
ATFD	#attributes from unrelated classes accessed directly or by invoking accessor methods
CINT	#distinct operations called from the measured operation
LAA	#attributes from the methods definition class, divided by #variables accessed
NAS	#public methods of a class, that are not overridden or specialized from the ancestors
NOAM	#accessor methods
NOM	#methods of the measured class
NOPA	#public attributes of the measured class
PNAS	#public methods of a class not overridden or specialized from the ancestors, divided by #public methods
TCC	#method pairs of a class that access in common at least one attribute of the measured class
WMC	the sum of the statical complexity of all methods in a class.
WOC	#functional public methods, divided by the total number of public members

5. Ensuring that no changes in the system behavior results from this refactoring.

It is clear that for very complex Blobs, implemented in complex systems, the Extract Class refactoring is very challenging without a (semi-) automatic tool support. This is the reason why researchers work on methods and tools supporting the software engineer in the identification of (i) Blob classes in software systems and (ii) extract class refactoring solutions to remove them.

As for the identification of Blob classes, Marinescu (69) proposes a mechanism called "detection strategies" for formulating metrics-based rules that capture deviations from good design principles and heuristics. The detection strategies are formulated in different steps. Firstly, the *symptoms* that characterize a particular bad smell should be defined (e.g., in case of Blob *high complexity*, *low cohesion*, and *access of "foreign" data*). Second, a proper *set of metrics* measuring these symptoms should be identified (e.g., *Weighted Method Count (WMC)* for high complexity, *Tight Class Cohesion (TCC)* for class cohesion, and *Access to Foreign Data (ATFD)* for measuring the access to external attributes of a class). Having this information the next step is to define thresholds to classify the class as affected (or not) by the defined symptoms. For example, establishing for which values of TCC a class should by identified as a "*low cohesive class*". Finally, AND/OR operators should be used to correlate the symptoms, leading to the final rule to detect the smells and thus, refactoring opportunities. The evaluation conducted on two software systems shows how using customized "detection strategies" it is possible to identify nine bad smells with an average accuracy of 70%.

Trifu and Marinescu (2) present an approach to support the decision making process in object oriented refactoring. In particular, they exploit correlation between structural anomalies, i.e., different types of code smells that often occur together, to build a pattern-like mapping of design problems to the adequate treatments. In particular, the 12 structural metrics reported in Table 2.1 are used in different combinations to capture symptoms that, when occurring together, can lead to design problems (e.g., Blob classes) that should be solved using well-known refactoring operations (e.g., Extract Class).

Joshi *et al.* (70) present a method for identifying less cohesive classes in a software system. In particular, their approach examines lattices based on the structural dependencies between attributes and methods. Unconnected nodes in a lattice signify the elements that are not related with each other. In this way, is not only possible to identify less cohesive classes but also to pick out which class members contribute to the lack of cohesion of the identified classes. This information can be used to find candidates for different refactoring operations, including Extract Class. In a reported preliminary evaluation the authors show how, applying the refactoring opportunities identified through their approach, it is possible to improve the value of some quality metrics.

Khomh *et al.* (71) propose an approach based on Bayesian Belief Networks (BBNs) to specify design smells and detect them in programs. In this work the authors focus the attention on the detection of Blob classes and thus, of Extract Class refactoring opportunities. In particular, given a class C as input, the output of the BBN is a probability that C is a Blob class. The evaluation is performed on two open source systems by measuring precision and recall of the model with manually located smells.

Moha *et al.* (72) introduced DETEX, a method for the specification and detection of code and design smells. DETEX uses a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Four design smells are identified by DETEX, namely Blob class, Swiss Army Knife, Functional Decomposition, and Spaghetti Code. The results achieved in the reported evaluation show that DETEX is able to reach a recall of 100% and a precision greater than 50% in the detection of the four above mentioned bad smells.

Other approaches are able not only to identify Extract Class refactoring opportunities, but also to propose a possible decomposition for the identified low cohesive classes

(i.e., the Extract Class solution). Simon *et al.* (73) provide a metric-based visualization tool able to identify, among others, Extract Class refactoring opportunities. In particular, each class is analyzed to verify what are the structural relationships (i.e., method calls and attribute accesses) between methods. If in the class it is possible to identify different sets of cohesive attributes and methods, an Extract Class refactoring opportunity is identified and, implicitly, the classes to be extracted represented by the sets of cohesive attributes and methods. Examples of the proposed tool reported in the work show its potential usefulness in integrated development environments.

Similarly, Fokaefs *et. al.* (19) use a clustering algorithm to perform Extract Class refactoring. Their approach analyze the structural dependencies existing between 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, of the class. Thus, the Jaccard distance between all 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 this aim. Differently from our approach for Extract Class refactoring presented in Chapter 4, in (19) only structural information is taken into account. Moreover, while our approach is able to automatically identify the appropriate number of classes that should be extracted from a Blob class, the approach presented in (19), like all the approaches based on hierarchical clustering, requires the definition of a threshold to cut the dendrogram. The authors tried to mitigate this issue by proposing different refactoring opportunities that can be obtained using different thresholds. However, the software engineer needs to analyze the different solutions in order to identify the one that provides the most adequate division of responsibilities.

We proposed in (1) a different approach based on graph theory for Extract Class refactoring, that presents commonalities and differences with the new approach presented in this thesis. Similarly to the approach presented in this thesis, in (1) a class to be split is represented by a weighted graph, where each node represents a method of the class and the weight of an edge that connects two nodes (methods) is a combination of structural and semantic similarity measures between the two methods. However, while in (1) the graph is always split in two sub-graphs (corresponding to the two extracted classes) using a MaxFlow-MinCut algorithm, the new two-steps algorithm presented

in this thesis overcomes this limitation since it is able to split the graph in more subgraphs by automatically identifying the appropriate number of classes (which may be more than two) that should be extracted from a Blob class, thus resulting in a better division of responsibilities.

2.2.2 Extract Package refactoring

Strongly related to Extract Class refactoring, but at a different level of granularity, is the Extract Package refactoring. As it is possible to have complex classes grouping together unrelated responsibilities (i.e., Blobs), it is also possible to observe the same phenomenon at a higher granularity level: packages grouping together classes implementing unrelated responsibilities. These packages are known as "Promiscuous packages" (64) and clearly exhibit low cohesion. The aim of Extract Package refactoring is to split a promiscuous package into different cohesive packages grouping together classes implementing similar responsibilities. Also here (semi-) automatic support is desirable. In fact, promiscuous packages often implement entire subsystems grouping together hundreds of classes. Similarly as seen with the Extract Class, to perform Extract Package refactoring the developer needs to analyze all classes in the promiscuous package, understand their responsibilities, and group together in new packages those implementing similar things. This is clearly a hard and time-consuming task. For this reason we present in Chapter 5 the first approach available in literature to explicitly support this kind of refactoring. However, some of the techniques proposed in the past for software re-modularization could be easily adapted to solve the problem of promiscuous packages. In fact, it is enough to consider the classes of the promiscuous package as the entire set of classes to re-modularize and apply one of the approaches existing in literature to organize the classes into new packages (i.e., to split the promiscuous package into new ones). For this reason in the following we discuss the state of the art in software re-modularization.

Most of the work on re-modularization is based on clustering techniques. Wiggerts (74) provides the theoretical background for the application of cluster analysis in software re-modularization. The paper discusses how to establish similarity criteria between the entities to cluster and gives a summary of possible clustering algorithms to use in software re-modularization. Anquetil *et al.* (75) tested some of the algorithms proposed by Wiggerts studying different issues that may influence the clustering results

when doing remodularization. In particular, they studied the impact of the following choices:

- how the *entities to be clustered* are *described*;
- how the *coupling* between entities is *computed*:
- what *clustering algorithm* should be *used*.

They performed this study on several software systems, including a real world legacy system, evaluating the effect of these parameters on the proposed clusterings. Among the results achieved in this study, one of the more interesting is that the used clustering algorithm impacts the results less than the technique used to describe the entities and to measure coupling between them (75).

Mitchell and Mancoridis (76) proposed some guidelines to compare the performance of different clustering algorithms for source code decomposition. In particular, they introduced two new measures, i.e., Edge Similarity (*EdgeSim*) and Merge Clusters (*MeCl*), to compare a decomposition produced by a clustering algorithm with an expert partition, measuring their similarity.

Wu *et al.* (77) conducted a comparative study of clustering algorithms in the context of software evolution. In particular, the authors focused their attention on the stability of clustering algorithms, i.e., a clustering algorithm is stable if it produces very similar output given very similar input, like two subsequent versions of the same system with no major changes. Their results show that for large systems the analyzed clustering algorithms are not ready to be widely adopted.

Maqbool and Babri (78) focused on the application of hierarchical clustering in the context of software architecture recovery and modularization. They investigated the measures to be used in this domain, categorizing various similarity and distance measures into families according to their characteristics. The behavior of various clustering algorithms was also studied on four large legacy systems.

Mancoridis *et al.* (79) introduce a search-based approach using hill-climbing based clustering to identify the modularization of a software system. This technique is implemented in Bunch (80), a tool supporting automatic system decomposition. To formulate software re-modularization as a search problem, Mancoridis *et al.* define (i) a representation of the problem to be solved (i.e., software module clustering) and (ii) a way

to evaluate the modularizations generated by the hill-climbing algorithm. Specifically, the system is represented by the Module Dependency Graph (MDG), a language independent representation of the structure of the code components and relations (79). The MDG can be seen as a graph where nodes represent the system entities to be clustered and edges represent the relationships among these entities. An MDG can be weighted (i.e., a weight on an edge measures the strength of the relationship between two entities) or unweighted (i.e., all the relationships have the same weight).

Starting from the MDG (weighted or unweighted), the output of a software module clustering algorithm is represented by a partition of this graph. A good partition of an MDG should be composed by clusters of nodes having (i) high dependencies among nodes belonging to the same cluster (i.e., high cohesion), and (ii) few dependencies among nodes belonging to different clusters (i.e., low coupling). To capture these two desirable properties of the system decompositions (and thus, to evaluate the modularizations generated by Bunch) Mancoridis *et al.* (79) define the Modularization Quality (MQ) metric as:

$$MQ = \begin{cases} (\frac{1}{k}\sum_{i=1}^{k} A_i) - (\frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j}) & \text{if } k > 1 \\ A_1 & \text{if } k = 1 \end{cases}$$

where A_i is the Intra-Connectivity (i.e., cohesion) of the i^{th} cluster and $E_{i,j}$ is the Inter-Connectivity (i.e., coupling) between the i^{th} and the j^{th} clusters. The Intra-Connectivity is based on the number of intra-edges, that is the relationships (i.e., edges) existing between entities (i.e., nodes) belonging to the same cluster, while the Inter-Connectivity is captured by the number of inter-edges, i.e., relationships existing between entities belonging to different clusters.

Search-based approaches are also used in (81), (82), (83), and (84). In particular, in (81), (83), and (84) the authors used a genetic algorithm to improve the subsystem decomposition of a software system. The fitness function to be maximized is defined using a combination of quality metrics, e.g., coupling, cohesion, and complexity. However, hill-climbing have been demonstrated to ensure higher quality and more stable solutions than a single objective genetic algorithm (85).

Praditwong *et al.* (86) introduce two multi-objective formulations of the software remodularization problem, in which several different objectives are represented separately. The two formulations slightly differ for the objectives embedded in the multi-objective

function. The first formulation–named Maximizing Cluster Approach (MCA)—has the following objectives: (i) maximizing the sum of intra-edges of all clusters, (ii) minimizing the sum of inter-edges of all clusters, (iii) maximizing MQ, (iv) maximizing the number of clusters, and (v) minimizing the number of isolated clusters (i.e., clusters composed by only one class). The second formulation—named Equal-Size Cluster Approach (ECA)—attempts at producing a modularization containing clusters of roughly equal size. Its objectives are exactly the same as MCA, except for the fifth one (i.e., minimizing the number of isolated clusters) that is replaced with (v) minimizing the difference between the maximum and minimum number of entities in a cluster. The authors compared their algorithms with Bunch. The conducted experimentation provides evidence that the multi-objective approach produces significantly better solutions than the existing single-objective approach though with a higher processing cost.

The re-modularization approaches discussed above, even if applicable with little modification to the problem of Extract Package refactoring, differently from the approach presented in this thesis only exploit information derived by structural metrics. From this point of view our Extract Package approach (see Chapter 5) is closer to (87), (88), (89), and (90). In particular, Maletic and Marcus (89) combined semantic and structural measures to identify ADTs in legacy code. They used Latent Semantic Indexing (LSI) (91), an Information Retrieval (IR) technique, to capture semantic relationships between source artifacts.

Kuhn *et al.* (92) also used LSI to cluster together source artifacts that use a similar vocabulary. Moreover, they provided a visual notation that gives an overview of all the clusters and their semantic relationships. Their approach is focused on the identification of topics in the source code and, for this reason, only uses semantic information ignoring the structural ones.

Corazza *et al.* (87, 88) presented a clustering based approach to partition object oriented systems into subsystems. In particular, they extracted lexical information from the source code and use the K-Medoids partitioning algorithm (88) or the Hierarchical Agglomerative Clustering algorithm (87) to build subsystems containing semantically related classes. In (88) and (87) the structural dependencies between the classes are ignored.

A clustering based approach using both structural and lexical information is proposed by Scanniello *et al.* (90), but it is focused on recovering a layered architecture

from the source code of object oriented systems. In particular, the structural information is used by the Kleinberg algorithm (93) to identify software layers, while lexical information is employed to partition each identified layer into software modules using the k-means algorithm (94).

The approach presented in this thesis to support Extract Package refactoring, differently from all clustering techniques (e.g., k-means), does not require the number of package to be extracted as input, but is able to automatically infer it.

Lastly, it is worth mentioning the work by Ducasse *et al.* (95), that present the Package Surface Blueprint, a visual approach for understanding package relationships in complex software systems. The authors showed that their approach helps users in identifying poorly designed packages, and thus, good candidates for Extract Package refactoring operations.

2.2.3 Move Method refactoring

Other common refactoring operations deal with moving software entities (e.g., methods) in more appropriate parts (e.g., classes) of the software system. The final goal is always the same: trying to group together similar responsibilities in order to ease code comprehension and maintenance. An example of these refactoring operations is the Move Method.

Move Method refactoring is targeted to solve the bad smell known in literature as "Feature Envy" (3). This smell arises when a method seems to be more interested in a class other than the one it is implemented in, e.g., the method invokes getter methods on another object many times (3). Clearly, this negatively influences both the cohesion and the coupling of the class in which the method is implemented. In fact, the method suffering of feature envy reduces the cohesion of the class, because it likely implements different responsibilities with respect to those implemented by the other methods of the class and increases the coupling, due to the many dependencies with methods of the envied class.

Applying Move Method refactoring 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 especially in large software systems (20). For this reason, approaches

to identify feature envy bad smells (and thus, move method refactoring opportunities) are available in literature.

The visualization tool provided by Simon *et al.* (73), besides supporting the identification of Extract Class refactoring opportunities (see Section 2.2.1), also allows to identify Move Method opportunities by analyzing structural information in the source code. In particular, for each method in the system under analysis, the methods invoked and the instance variables accessed by it are analyzed. If a method uses many attributes and/or methods of a class (not the one it is implemented in) a move method refactoring opportunity is identified. Also the approaches by Marinescu (69) and Joshi *et al.* (70) discussed in Section 2.2.1 can be used to identify move method refactoring opportunites.

Du Bois *et al.* (96) analyze how refactoring impacts on coupling/cohesion characteristics and how refactoring opportunities that improve these characteristics can be identified. As an example, to identify good Move Method refactoring opportunities the authors suggest the two following guidelines:

- *G1 Localize dependencies*: move methods that (i) do not use local resources, (ii) are called upon seldom, and (iii) have dependencies mostly with a single external class.

- *G2 Separate concerns*: break up a method that depends on many different external classes into pieces which mostly refer to only a single external class. Apply *G1* on each of these extracted methods. Thereafter, the original method will then act as a coordinator which directs the collaborations between the responsible classes, and can be moved itself to a class which fits this coordination responsibility.

The evaluation performed on one of the Apache Tomcat[1] packages shows how it is possible to achieve quality improvements (as measured by quality metrics) with restricted refactoring efforts, following the proposed guidelines.

Atkinson *et al.* (97) present a low-cost, syntactic approach for automatically identifying poorly structured code, suitable for refactoring. The proposed approach uses the symbol table and reference information together with simple structural code metrics, such as line and statement counts, to identify refactoring opportunities of four different types: Encapsulate Field, Decompose Conditional, Replace Magic Number, and

[1] http://tomcat.apache.org/

Move Method. To evaluate the proposed approach, the authors run it on 10 C++ systems identifying a set of refactoring opportunities. Then, they manually validate their goodnesses, reporting 88% of the suggested refactoring opportunities as meaningful.

Seng *et. al.* (18) 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 performed evaluation show that their approach executed on an open source software system 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.

Genetic Algorithms are also used by Bodhuin *et al.* (98) in SORMASA, SOftware Refactoring using software Metrics And Search Algorithms, a refactoring decision support tool to optimize the quality of a software system, e.g., maximizing the cohesion and minimizing the coupling. In SORMASA refactoring operations involving eld and method movements between classes are considered. No evaluation about the tool is presented in (98).

Another approach to automate move method refactoring has been proposed by Tsantalis *et. al.* (20). In particular, for each method of the system, their approach forms a set of candidate target classes where a method should be moved. 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 the approach presented in this thesis, the approach presented in (20) also only exploits structural information extracted from the source code to identify the envied class for a method under analysis. The approach in (20) has been evaluated by (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 coupling 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 make 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 in (20) has also been implemented as an Eclipse plug-in, coined as JDeodorant[1].

Unlike the approaches described above, our approach to Move Method refactoring described in Chapter 6, also takes into account semantic (i.e., lexical) 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).

Note that, in (20) the authors thoroughly explain why their approach should be preferred to that one proposed by Seng *et. al.* (18). Among the most important weaknesses identified for the approach proposed in (18) are:

- *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 (10 in the example reported in the paper) and suggest as move method operations those present in 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 if the approach by Seng *et. al.* (18) is valid from a theoretical point of view, its practical application is quite hard. For this reason, we chose to compare our Move Method technique (see Chapter 6) to the approach presented in (20) through the JDeodorant plug-in in our case studies (see Sections 6.3 and 6.4).

[1] http://jdeodorant.com

2.2.4 Move Class refactoring

Move Class refactoring aims at solving one of the main reasons for architectural erosion in software systems: inconsistent placement of source code classes in software packages (82, 99). Such a scenario, on one hand negatively impacts the package cohesion and on the other hand increases the number of dependencies (coupling) between packages (100). In such cases, re-modularization of the system is necessary (3, 101). While most of the approaches existing in literature and discussed in Section 2.2.2 focus on proposing a whole new re-modularizations to the developer (e.g., (77, 79, 83)), using Move Class refactoring it is possible to perform a focused and fine-grained remodularization moving misplaced classes into a more suitable package of the system, i.e., one grouping classes more functionally related to it than the one it is placed in. Clearly, identifying Move Class refactoring opportunities in a large software system is not easy, due to the thousands of classes contained in it and to the intricate web of relationships existing among them. Despite this, the only approach existing in literature to support Move Class refactoring is the one by Abdeen *et al.* (82). They proposed a heuristic search-based approach for automatically reducing the dependencies between packages of a software system. Their technique, starting from an initial decomposition, optimizes the existing package structure by moving classes between the original packages. Our approach, described in Chapter 7, exploits not only structural information to derive refactoring operations, but also conceptual information derived from identifiers and comments. In addition, *R*3 is the first recommendation system able to provide an evaluation of the proposed re-modularization based on quantitative and qualitative data (see Chapter 7).

2.2.5 Other refactoring operations

Besides the above discussed refactoring operations, a wide range of other ones have been automated in the literature.

Casais (14) proposes an algorithm that analyses the redefinitions carried out on inherited properties when a class is added to a hierarchy, and restructures the hierarchy to maximize abstraction. In (15) the author presents Guru, a tool for restructuring inheritance hierarchies and refactoring methods simultaneously. All the restructuring

operations suggested by Guru are focused on the minimization of the source code duplication.

O'Keeffe *et al.* (17) formulate the task of refactoring as a search problem in the space of alternative designs. The alternative designs are generated applying a set of refactoring operations. In particular, the refactoring types considered in this work are: push down field, pull up field, pull down method, pull up method, extract hierarchy, and collapse hierarchy. 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 (9) that reflects refactoring goals. The results achieved in the reported experimentation show that the presented approach is able to improve the design quality of a given system from a quality metric point of view. No evaluation with software developers is reported. Note that while all the approaches described in this thesis exploits a combination of structural and semantic measures, in (17) only structural metrics are used.

Maruyama *et al.* (16) present a mechanism to improve the reusability of frameworks through Extract Method refactoring operations. 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 the code of their frameworks in the future in the same way that they often did in the past. The results of the reported experimentation show a reduction rate of up to 22% in the number of statements a programmer has to write when creating several applications.

Abadi *et al.* (102) propose the use of fine slicing to support the Extract Method refactoring, used to decompose a long method body into different methods, each with a precise responsibility. The fine slicing is used to compute program slices that are executable and extractable from their surrounding code and thus, useful to support Extract Method refactoring. Another approach to support Extract Method refactoring is presented by Tsantalis *et al.* (21). The approach in (21) automatically identifies Extract Method refactoring opportunities by employing two slicing techniques:

1. *A complete computation slice* is used to identify all the statements in a method affecting the computation of a given variable.

2. *An object state slice* is used to capture the statements affecting the state of a given object.

The evaluation reported in (21) shows as the proposed methodology is able to capture slices of code implementing a distinct and independent functionality compared to the rest of the original method and thus lead to extracted methods with useful functionality.

Van Emden and Moonen (103) present jCOSMO, a code smell browser that detects and visualizes code smells in Java source code. They focus the attention of two bad smells related to the Java programming language, i.e., *instanceof* and *typecast*. The first occurs when in the same block of code there is a concentration of *instanceof* operators making the code difficult to understand. In these cases, refactoring operations aimed at brake the method in different parts (e.g., Extract Method) could be recommendable. As for the *typecast* bad smell, it appears when an object is explicitly converted from one class type into another, possibly performing illegal casting which results in a runtime error.

2.3 On the use of Semantic Information in Software Engineering

In the last years semantic (textual) information extracted from software artifacts have been used to support a wide range of software engineering tasks. Note that, when talking about software artifacts, we do not refer only to source code but to the whole set of documents (e.g., requirements, use cases, test cases, etc.) created during software development and maintenance.

The earliest applications of text analysis in software engineering are focused on creating software libraries (104, 105) and supporting software reuse (23, 24, 25, 26). In these cases Information Retrieval (IR) (91, 106) techniques are used to analyze the text contained in software artifacts and understand which code components implement similar concepts (in case of libraries building) or which code component implements a particular functionality to be reused.

One of the most diffused application of IR in software engineering is the recovery and management of traceability links between software artifacts (27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42). An IR-based traceability recovery method compares a set of source artefacts (used as a query) against a set of target artifacts

and ranks the textual similarity of all possible pairs of artefacts (candidate traceability links). The conjecture is that artifacts having a high textual similarity probably share several concepts, so they are likely good candidates to be traced from one to another (28).

Strongly related to the traceability recovery problem is concept location, i.e., identification of concepts, features, concerns in source code. Also here textual analysis technique have been widely applied (43, 44, 45, 46, 47, 48, 49, 50, 51, 52). In particular, the developer formulates a query describing the concept for which she is looking for in source code and the IR engine retrieves a list of code components hopefully relevant for the provided query.

Another common application of textual analysis in software engineering is change impact analysis (53, 54, 55, 56). Also here, given a modified code component (e.g., a method), the text in it is used as query to look in the source code for components containing similar terms. The conjecture is that code component containing similar terms are likely to implement similar responsibilities and thus, to change together during software maintenance.

Textual analysis has also been applied to software clone detection (57, 58, 59). Again, the basic idea is always the same: code components containing similar terms are likely to implement similar responsibilities. Thus, textual analysis can identify "similar" code components, representing possible clone instances.

Finally, an application of textual analysis in software engineering strongly related to this thesis is the computation of software quality metrics like cohesion (60) and coupling (55). These measures are based on the semantic information (i.e., domain semantics) captured in the code by comments and identifiers. In particular, in (60) the authors define the Conceptual Cohesion of Classes (C3): a class exhibits a high conceptual cohesion if its methods have high textual similarity among them. On the other side, in (55) the authors define the Conceptual Coupling Between Classes (CCBC): two classes exhibit high coupling if their methods have a high textual similarity.

As it can be noticed, all these successful applications of textual analysis in software engineering are in someway based on the conjecture that artifacts containing similar terms are likely to "talk" about similar things and thus, to be related. Since the goal of several refactoring operations is to group together similar things (e.g., Extract

Class, Move Method), we are confident that semantic information can be useful to support these kinds of refactoring, allowing to achieve better refactoring recommendations as compared to the only use of structural measures (e.g., method calls). In fact, as demonstrated in (60) and (55), semantic information is able to identify related code components even if they are not explicitly (structurally) linked.

3

A Framework of Software Refactoring Methods

3.1 Introduction

When designing an approach to identify refactoring solutions three are the main challenges to deal with:

1. *Correctly capturing relationships existing between code components object of the refactoring.* This step is very important since the goal of several refactoring operations (e.g., Extract Class, Move Method, Move Field, Extract Subclass, Extract Package, Inline Class, etc), is to re-organize code components in such a way that related components (i.e., those implementing similar responsibilities) are grouped together. All the approaches presented in this thesis exploit both structural and semantic information to capture relationships between the code components object of the refactoring.

2. *Define a convenient representations of the problem to be solved.* Once the relationships between code components have been captured, it is important to correctly represent this information in order to simplify the problem to be solved, i.e., recommending the refactoring solution. When the refactoring operation is focused on the decomposition of complex objects (e.g., Extract Class, Extract Subclass, Extract Package) or on the movement of code components (e.g., Move Method, Move Class) a weighted graph is a suitable representation for the available information. Each node in the graph can represent one of the code components under

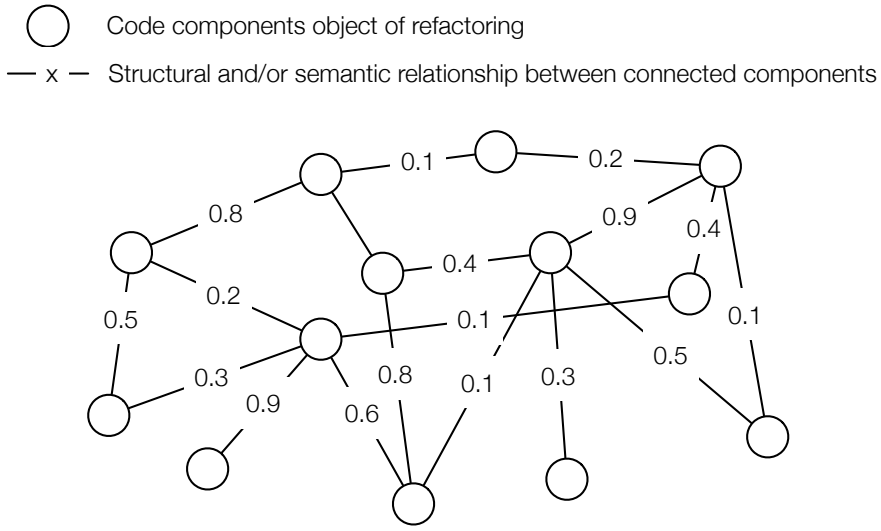

Figure 3.1: Weighted Graph representation of code components object of refactoring

analysis (see Figure 3.1). As example these components could be the classes of a promiscuous package in need of Extract Package refactoring. As for the weight on the edge connecting two nodes (i.e., code components), it can be used to represent the relationships between them (see Figure 3.1). Having a graph-based representation has two keys advantages:

- All the information needed to perform the refactoring can be easily stored in a very simple data structure: a squared matrix. In particular, if the graph has n nodes (i.e., the code components to analyze are n), the matrix has dimension $n \times n$ and its generic entry i, j shows the weight on the edge connecting the nodes i and j. Clearly, if the weight is equal to zero, the nodes i and j are not adjacent, i.e., there is no edge connecting them.
- A wide set of efficient algorithms from the graph-theory field available to analyze in different ways the (weighted) graph. The interested reader can find a complete treatment of graph-theory algorithms in (107).

All the approaches in this thesis exploit a graph-based representation of the information gathered from the analyzed source code.

3. *Define the algorithm to generate the refactoring solution.* Given the chosen problem representation it is crucial to define the algorithm to use to generate the

refactoring solution. Since the approaches in this thesis exploit a graph-based representation of the problem, it is possible to identify refactoring solutions by applying appropriate graph-theory algorithms.

Algorithms based on *subgraphs analysis* can be useful to identify solutions for refactoring dealing with decomposing complex objects into simpler ones (e.g., Extract Class refactoring). In fact, algorithms like the *Max Flow-Min Cut* or different clustering algorithms can be used to identify, inside the weighted graph representing the complex object to decompose, sets of strongly connected nodes (i.e., subgraphs) representing the simpler objects to extract. Applying these algorithms it is possible to automatically identify an appropriate number of objects in which to decompose the complex object to refactoring.

As for refactoring operations dealing with moving specific code components (e.g., Move Method refactoring), algorithms based on *network analysis* can be a suitable solution. These algorithms are generally designed to work on very large networks (i.e., very large graphs) and are able to identify, given a node, its main network relationships. As example, to identify the best class in which a method m should be moved (i.e., Move Method refactoring), a *network analysis* algorithm can be used to identify the set S of methods in the system having the strongest relationships with it. Then, the class containing the highest number of S's methods can be suggested as the best one in which placing m.

In the following section we describe (i) how to capture in source code structural and semantic information useful to support software refactoring and (ii) how the approaches composing our framework exploit graph theory to support refactoring operations. In this thesis our framework has been instantiated on four approaches presented in Sections 4, 5, 6, and 7. However, note that the foundation of our framework (i.e., the information exploited to capture relationships between code components, the problem representation, and the algorithm used to identify the solution) is general enough to support several refactoring operations acting at different granularity levels (e.g., methods, classes), and in particular those related to:

- *Decomposing Complex Components*: Extract Class, Extract Subclass, Extract Superclass, Extract Package;

- *Moving Misplaced Components*: Move Method, Inline Class, Inline Method, Remove Middle Man, Pull Up Method, Push Down Method.

3.2 Capturing Structural Relationships between Source Code Components

In this section we discuss the sources of information capturing structural relationships (i.e., structural coupling) between code components.

3.2.1 Method's Calls

The most obvious source of information that can be exploited to capture structural relationships between code components is the calls interaction. Methods generally call each other when co-operating in the implementation of some responsibilities. This source of information can be useful for refactoring operations acting at both method (e.g., Extract Class, Move Method) and class (e.g., Extract Package, Move Class) level.

Methods with high calls interaction are good candidates to be grouped together when performing refactoring operations acting at method level. As example, this source of information is precious for Extract Class refactoring (which methods of the class to be split should be grouped together), for Inline Method (which methods co-operate so much that merging one into another would be better to reduce coupling) and so on.

A measure capturing method calls interaction is the Call-based Dependence between Methods (CDM) (1). In particular, 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 $CDM_{i \to j} = 1$ it means that m_j is only called by m_i. Otherwise, if $CDM_{i \to j} = 0$ it means that m_i never calls m_j. To ensure that CDM represents a commutative measure the overall CDM of m_i and m_j is computed as follows:

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

The CDM measure is exploited by our Extract Class and Move Method refactoring approaches presented in Sections 4 and 6, respectively.

The calls between methods belonging to different classes also represent the most used information to measure coupling between classes. In fact, it is reasonable to think that classes having a high calls interaction co-operate to implement the same (or strongly related) responsibilities and thus, are very coupled. This information is particularly precious for refactoring operations aimed at improving the modularization quality of Object-Oriented systems, e.g., Extract Package, Move Class.

There is a variety of metrics available in literature to measure the coupling between classes based on their calls interaction. Examples are the Information-Flow-based Coupling (ICP) (108) and the Message Passing Coupling (MPC) (109).

MPC is the basic coupling metric for measuring 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.

As for the ICP, it measures the amount of information flowing into and out of a class via parameters through method invocation, i.e., the measure sums the number of parameters passed at each method invocation. Like the majority of coupling metrics in literature, this metric is defined at the system level, i.e., for a given class c all method calls between c and *all* other classes in the system are taken into account. However, to capture coupling between pairs of classes (information needed when performing refactoring) the ICP metric has been redefined in (55); the information-flow-based coupling between a pair of classes c_i and c_j is measured as the number of method invocations in the class c_i to methods in the class c_j, weighted by the number of parameters of the invoked methods:

$$ICP_{i \to j} = \sum_{k=1}^{|calls(c_i,c_j)|} p(call(c_i, c_j)_k)$$

where $p(call(c_i, c_j)_k)$ is the number of parameters in the $k-th$ call from c_i to c_j.

To ensure that ICP represents a commutative measure, the overall information-flow-based coupling between the classes c_i and c_j is defined as follows:

$$ICP(c_i, c_j) = ICP(c_j, c_i) = \max\{ICP_{i \to j}, ICP_{j \to i}\}$$

The ICP measure is exploited by our Extract Package and Move Class refactoring approaches presented in Sections 5 and 7, respectively.

3.2.2 Shared Instance Variables

The instance variables shared by two methods are a precious source of information for refactoring operations acting at method level (e.g., Extract Class, Move Method). In fact, they also represent a form of communications between methods (performed through shared data). Thus, methods sharing instance variables are more coupled that methods do not sharing any data.

A measure to capture this form of coupling between methods is the Structural Similarity between Methods (SSM) (110), used to compute the cohesion metric ClassCoh (110). 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 variables referenced by the two methods:

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

SSM has values in [0, 1]; the higher the number of instance variables the two methods share, the higher the SSM value and thus, the coupling between methods. The ClassCoh is defined as the ratio of the sum of the similarities between all pairs of methods to the total number of possible pairs of methods.

The SSM measure is exploited by our Extract Class and Move Method refactoring approaches presented in Sections 4 and 6, respectively.

3.2.3 Inheritance Relationships

A source of structural information to capture relationships between classes (and thus, useful to refactoring acting at class level) are the inheritance dependencies existing among them. Exploiting this information is mandatory when working on approaches aimed at supporting refactoring operations that modify the class hierarchy, e.g., Extract Subclass, Extract Superclass, Pull Up Method, Push Down Method.

Generally, the measurement of inheritance relationships between two classes is performed through a simple boolean value equals to *true* if two classes have inheritance relationships, to *false* otherwise.

3.2.4 Original Design

A last source of structural information that can be exploited to capture relationships between code components is the original design or, in other words, the choices made by the developers when designing the system. For example, if the developers put two methods inside the same class it is reasonable to think that from their point of view these two methods are in some way related. For instance, in case of Move Method refactoring, this information can be used to take into account the choices made by the original developers when suggesting refactoring operations.

The same conjecture can be also made at class level: if two classes were put in the same package by the developers it is likely that from their point of view these two classes were in some way related.

We exploited information about the original design in our Move Method and Move Class refactoring approaches presented in Sections 6 and 7, respectively.

3.3 Capturing Semantic Relationships between Source Code Components

The semantic relationships between code components are computed by measuring their textual similarity. If the vocabulary of two code components (i.e., methods or classes) is very similar, it is likely that the developers used similar terms to describe similar responsibilities implemented by the two components. This information can be useful to support all kinds of refactoring aimed at grouping together similar code components, at both method and class level.

To measure the textual similarity between code components Information Retrieval (IR) (106) techniques are employed. In the following we describe the process used to measure the textual similarity between code components using IR methods. Then, we present the metrics existing in literature exploiting IR techniques to compute semantic coupling between code components.

3.3.1 Measuring Textual Similarity between Code Components through IR Methods

The first step to compute the textual similarity between code components (in the following, simply "documents") is the *text normalization phase*, generally composed of

two sub-steps:

1. *Text pre-processing*: white spaces and most non-textual tokens (e.g., special symbols, some numbers) are pruned out from the text contained in the documents and all capital letters are transformed into lower case letters. Moreover, code identifiers composed of two or more words separated by using the under score or camel case separators are split into separate words, e.g., *getName* is split into *get* and *name*;

2. *Word extraction and filtering*: documents usually also contain common words (i.e., articles, adverbs, programming language keywords, etc) that are not useful to characterize their semantic. Thus, a stop word list is applied to discard such words (*term filtering*) (106).

The output of the indexing process is represented by a $m \times n$ matrix (called *term-by-document* matrix), where m is the number of all terms that occur within the documents, and n is the number of considered documents. 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 (106). Various methods for weighting terms have been developed in the IR field. However, two main factors come into play in the final term weight formulation:

1. *Term Frequency* (or *tf*): words that repeat multiple times in a document are considered salient. Term weights based on *tf* have been used since the 1960s (106).

2. *Document Frequency*: words that appear in many documents are considered common and are not very indicative of document content. A weighting method based on this, called inverse document frequency (or *idf*) weighting, was proposed by Sparck-Jones early 1970s (111).

Generally, these two factors are combined in a weighted schema known as *tf-idf* (106), which gives more importance to words having a high frequency in a document (high *tf*) and appearing in a small number of documents, thus having a high discriminant power (high *idf*).

Starting from the *term-by-document matrix* the computation of the textual similarity between pairs of documents (pairs of code components in our case) depends on the particular IR method adopted. The two of most interest for this thesis are the Vector

Space Model (VSM) (106) and its most important extension, namely Latent Semantic Indexing (LSI) (91).

3.3.1.1 Vector Space Model

In the VSM (106) each document is represented by a vector of terms (112). To compute the similarity between pairs of documents the model measures the similarity between the their vector representations. The similarity between two vectors is not inherent in the model. Typically, the angle between two vectors is used as a measure of divergence between the vectors, and cosine of the angle is used as the numeric similarity (since cosine has the nice property that it is 1.0 for identical vectors and 0.0 for orthogonal vectors). If $\overrightarrow{D_1}$ and $\overrightarrow{D_2}$ are two vectors representing two documents for which we are interested in computing their textual similarity, it can be calculated as follow (106):

$$sim(D_1, D_2) = \frac{\overrightarrow{D_1} \cdot \overrightarrow{D_2}}{\|\overrightarrow{D_1}\| \cdot \|\overrightarrow{D_2}\|} = \frac{\sum_{t_i \in D_1, D_2} w_{t_i D_1} \cdot w_{t_i D_2}}{\sqrt{\sum_{t_i \in D_1} w_{t_i D_1}^2} \cdot \sqrt{\sum_{t_i \in D_2} w_{t_i D_2}^2}} \tag{3.1}$$

where $w_{t_i D_1}$ is the value of the i^{th} component in the vector $\overrightarrow{D_2}$, and $w_{t_i D_1}$ is the i^{th} component in the vector $\overrightarrow{D_1}$. Since any word not present in either the documents has a $w_{t_i D_2}$ or $w_{t_i D_1}$ equals to 0 it is possible do the summation only over the terms common in the two documents. How $w_{t_i D_2}$ and $w_{t_i D_1}$ are determined is not defined by the model, but is dependent on the chosen weighting schema.

3.3.1.2 Latent Semantic Indexing

A common criticism of VSM is that it does not take into account relations between terms (91). For instance, having "automobile" in one document and "car" in another document does not contribute to the similarity measure between these two documents. LSI (91) was developed to overcome the synonymy and polysemy problems, which occur with the VSM model. In LSI the dependencies between terms and between documents, in addition to the associations between terms and documents, are explicitly taken into account. LSI assumes that there is some underlying or "latent structure" in word usage that is partially obscured by variability in word choice, and uses statistical techniques to estimate this latent structure. For example, both "car" and "automobile" are likely to co-occur in different documents with related terms, such as "motor", "wheel", etc. LSI

exploits information about co-occurrence of terms (latent structure) to automatically discover synonymy between different terms.

LSI starts, as well as VSM, by a term-by-document matrix A. Then it applies the Singular Value Decomposition (SVD)(113) to decompose the term-by-document matrix into the product of three other matrices:

$$A = T_0 \cdot S_0 \cdot D_0 \tag{3.2}$$

where T_0 is the $m\ x\ r$ matrix of the terms containing the left singular vectors (rows of the matrix), D_0 is the $r\ x\ n$ matrix of the documents containing the right singular vectors (columns of the matrix), S_0 is an $r\ x\ r$ diagonal matrix of singular values, and r is the rank of A. T_0 and D_0 have orthogonal columns, such that:

$$T_0^T \cdot T_0 = D_0^T \cdot D_0 = I_r \tag{3.3}$$

SVD can be viewed as a technique for deriving a set of uncorrelated indexing factors or concepts (91), whose number is given by the rank r of the matrix A and whose relevance is given by the singular values in the matrix S_0. Concepts "represent extracted common meaning components of many different words and documents" (91). In other words, concepts are a way to cluster related terms with respect to documents and related documents with respect to terms. Each term and document is represented by a vector in the r-space of concepts, using elements of the left or right singular vectors. The product $S_0 \cdot D_0$ ($T_0 \cdot S_0$, respectively) is a matrix whose columns (rows, respectively) are the document vectors (term vectors, respectively) in the r-space of the concepts. The cosine of the angle between two vectors in this space represents the similarity of the two documents (terms, respectively) with respect to the concepts they share. In this way, SVD captures the underlying structure in the association of terms and documents. Terms that occur in similar documents, for example, will be near each other in the r-space of concepts, even if they never co-occur in the same document. This also means that some documents that do not share any word, but share similar words may none the less be near in the r-space.

3.3.2 Semantic Coupling Measures

Semantic coupling can be captured at both method and class level. The Conceptual Similarity between Methods (CSM) has been introduced in (60) to define the Conceptual Cohesion of Classes and used in (55) to define the Conceptual Coupling between

classes. Two methods are conceptually related if their (domain) semantics are similar, i.e., they perform conceptually similar actions. To measure the CSM between two methods, LSI is used to compute their textual similarity. The CSM measure is exploited by our Extract Class refactoring approach presented in Section 4.

Concerning the classes, also in this case the conjecture is that classes having a high textual similarity are likely to implement similar responsibilities and thus, to be related (i.e., coupled).

To measure the semantic coupling between two classes, the Conceptual Coupling Between Classes (CCBC) (55) is used. The conceptual coupling between two classes c_i and c_j is defined as as:

$$CCBC(c_i, c_j) = \frac{\sum_{m_h \in c_i} \sum_{m_k \in c_j} CSM(m_h, m_k)}{|c_i| \times |c_j|}$$

where $|c_i|$ ($|c_j|$) is the number of methods in c_i (c_j). Thus, $CCBC(c_i, c_j)$ is the average of the coupling between all unordered pairs of methods from class c_i and class c_j. The definition of this measure ensures that CCBC is symmetrical, i.e., $CCBC(c_i, c_j) = CCBC(c_j, c_i)$.

The $CCBC$ measure is exploited by our Extract Package refactoring approach presented in Section 5.

3.4 An Approach for Decomposing Complex Components

In this section we present an approach to support the decomposition of complex components in software system. This approach can be instantiated to all the refactoring having this aim, like Extract Class and Extract Package refactoring. The approach is depicted in Figure 3.2.

Each complex component is built by several entities. For example, in case of Extract Class refactoring the object to decompose is the Blob class composed by several methods while in case of Extract Package refactoring it is the promiscuous package grouping together several different classes. The first phase of the presented process (shown in the top part of Figure 3.2) aims at building an *entity-by-entity* matrix representation of the component to be refactored. The *entity-by-entity* matrix is a square matrix of dimension $n \times n$ where n is the number of entities in the complex component to be refactored. The generic entry $c_{i,j}$ of the matrix represents the likelihood that entity

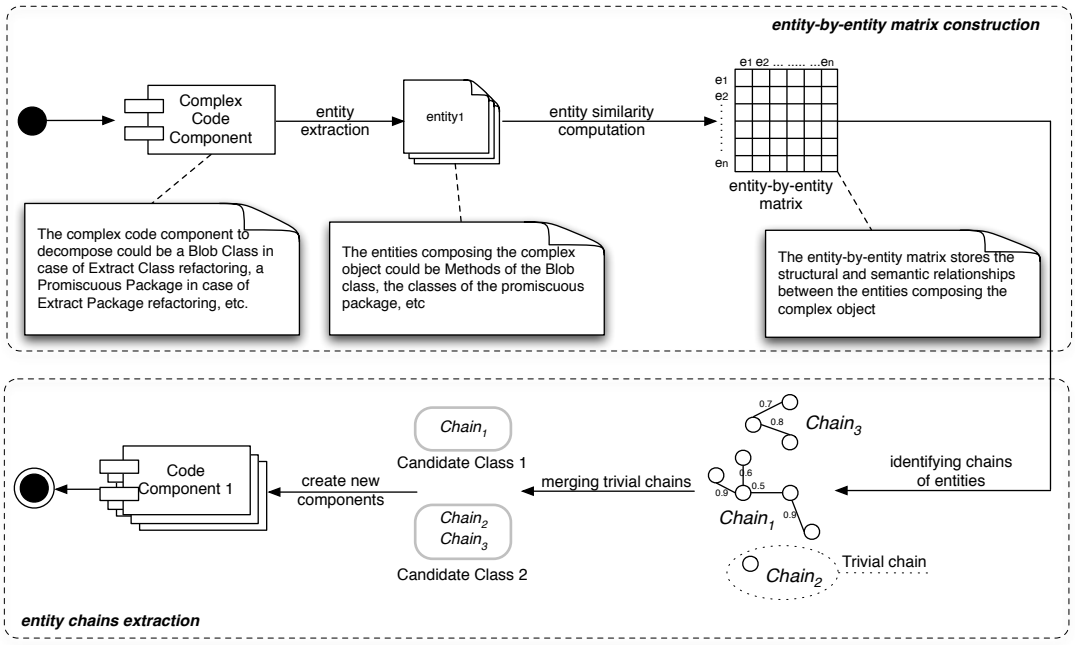

Figure 3.2: An Approach for Decomposing Complex Components

e_i and entity e_j should be in the same extracted component[1]. This likelihood can be computed by exploiting a combination of structural and semantic measures presented in Sections 3.2 and 3.3. The particular measures to exploit and the importance to assign at each of them must be chosen on the basis of the particular refactoring to support.

Using the information in the *entity-by-entity* matrix the second part (bottom path of Figure 3.2) of the process decomposes the complex component into simpler ones. Depending on the chosen measures, the weighted graph represented in the *entity-by-entity* matrix could have very different characteristics. For example, if only structural measures are used it is likely that the initial graph representation would be already a disconnected graph and thus, no further steps would be required. However, since our approache exploits a combination of structural and semantic measures it is very likely that the initial graph representation would be in general a complete graph (i.e., it contains all possible edges), or at least a connected graph, since the semantic similarity between two code components from the same system is very un-likely equal to zero. Thus, a filtering step is used to remove spurious links and split the initial

[1]It is worth noting that the *entity-by-entity* matrix is just a convenient way of storing a weighted graph representing the complex component to decompose.

graph represented in the *entity-by-entity* matrix into disconnected subgraphs. Then, the transitive closure of the *entity-by-entity matrix* is computed to identify chains of strongly related (or coupled) entities on the filtered graph (see Figure 3.2). Each computed chain represents a new component to be extracted from the original complex component. However, some of these chains could have a very short length grouping together very few entities. We refer to these chains as *trivial chains*. For some kinds of refactoring extracting components grouping together very few entities might not be desirable. For example, in case of Extract Package refactoring could be not desirable to extract from the promiscuous package a new package containing only one class, since singleton modules are not considered a good design choice. To avoid the extraction of components with a very low number of entities, in the second step of the approach each trivial chain is merged with the most similar non-trivial chain to obtain the final set of components to be extracted from the original complex component. It is worth noting that (i) the threshold length for discriminating among trivial and non-trivial chains should be customized on the basis of the particular refactoring to support and (ii) the similarity between chains is also in this case computed exploiting the selected structural and semantic measures.

We have instantiated this approach to support Extract Class (Section 4) and Extract Package (Section 5) refactoring.

3.5 An Approach for Moving Misplaced Code Components

The approach presented in this section can be used to support the movement of misplaced code components in software systems. The approach can be instantiated to all the refactoring operations having this aim, like Move Method, and Move Class refactoring. The approach is depicted in Figure 3.3.

When supporting this kind of refactoring the challenge is to identify inside the system of interest the misplaced code components among the hundreds contained in it. For example, in case of move method refactoring, it is necessary to identify among the hundreds of methods of the system those suffering of the Feature Envy bad smell (3). For this reason, an approach supporting refactoring operations dealing with the movement of misplaced code components generally works on very complex graphs representing

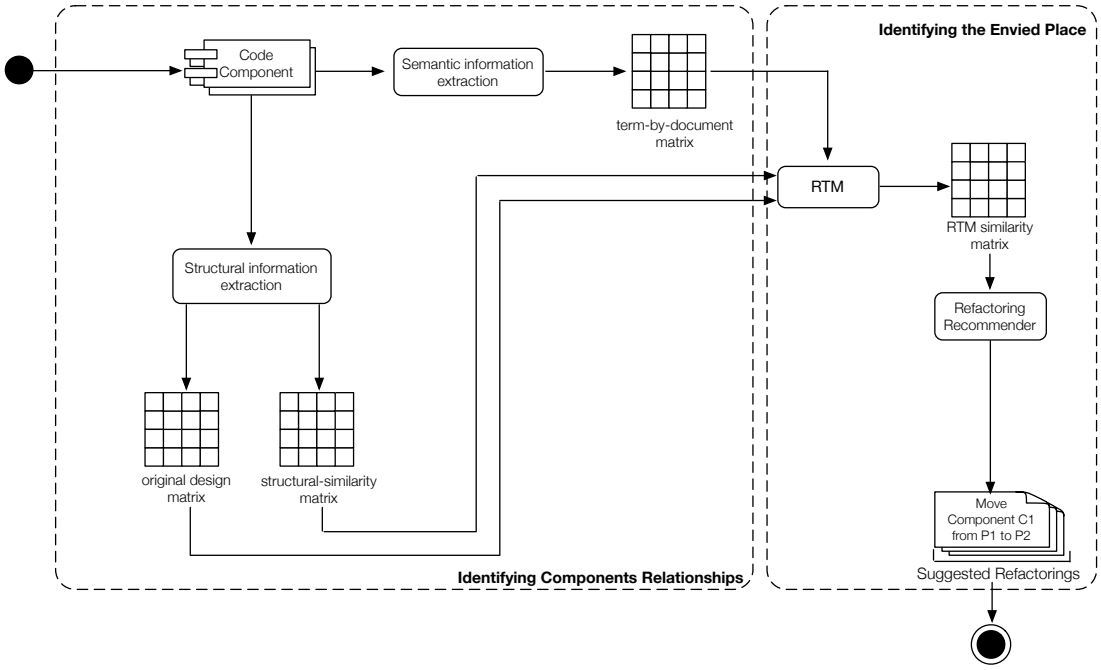

Figure 3.3: An Approach for Moving Misplaced Code Components

all the components of a system (e.g., all the methods for Move Method refactoring) and the relationships between them. Thus, to identify the refactoring solution it is not possible to apply simple splitting algorithms like the one seen in the previous section (Section 3.4), but network analysis algorithms are needed to gather information from the entire graph by efficiently analyzing all the relationships between the interested code components.

Our approach starts by analyzing the structural and semantic relationships between all components of the system (see Figure 3.3). To this aim, all the components of the software system under analysis are pre-processed to extract necessary information, that is, the current system design (i.e., which components are grouped together in the system), structural relationships between components, and textual content of components. Depending on the refactoring operation to support, the considered code components can have different granularity levels (e.g., will be methods in case of Move Method refactoring, classes in case of Move Class refactoring, etc.).

Information about the current system design is stored the *original design* matrix, a simple boolean $n \times n$ matrix, where n is the number of components composing the

software system to analyze. A generic entry $o_{i,j}$ of this matrix equals to 1 if the component c_i and the component c_j are grouped in the same place in the original design, otherwise it is equal to 0. For example, in case of Move Method refactoring it is possible to know which methods have been grouped in the same class by the original developers by analyzing the *original design* matrix. This information is used to take into account the choices made by the original developers when suggesting refactoring operations.

Structural dependencies among code components can be derived by using the measures described in Section 3.2 and are stored in the *structural-similarity matrix*. Also in this case, the particular measures to use depend on the refactoring operation on which our approach is instantiated. As for the semantic information, terms present in the identifiers, comments, and string literals are extracted from the code components of interest and stored in a *term-by-document* matrix. Thus, at this stage three graph-based representation of the relationships existing between the code components to analyze are available in the three computed matrices. The next step is to identify, for each code component, possible "envied places", i.e., places in the system where a code component should be moved.

To identify the envied place, the three computed matrices are supplied to RTM (Relational Topic Model) (62), a hierarchical probabilistic model of document attributes and network structure (*i.e.*, links between documents). The basic idea behind RTM is that textual documents (that is, the source code extracted by the components and stored in the *term-by-document matrix*) are modeled as mixtures of 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 (62). RTM (62) 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 of Chang *et al.* (62)), 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.

RTM requires two steps to generate a model, (1) model the documents in a given corpus as a probabilistic mixture of latent topics and (2) model the links between document pairs as a binary variable. Established as an extension of *Latent Dirichlet*

Allocation (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 (114):

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 $y_{d_i,d_j}|t_i, t_j \sim \psi\left(\eta \cdot |t_i, t_j, \right)$ where $t_i = \{t_{i,1}, t_{i,2}, \ldots, t_{i,n}\}$

The link probability function ψ_ϵ is defined as:

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

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

Thus, the peculiarity of RTM as compared to other topic modeling techniques is in its ability to adjust the probability distribution of each topic taking into account explicit relationships among the documents. In our approach, explicit relationships among the documents (code components) are modeled through structural dependencies among them and original design (stored in the *structural-similarity matrix* and *original design matrix*, respectively).

The model derived by RTM is then used to compute similarities among code components based on both probabilistic distributions of latent topics and underlying dependencies. After obtaining similarities among all the code components for a given system

(*RTM similarity matrix* in Figure 3.3), for each component the approach identifies a set of highly similar components (that is, code components sharing similar topics and/or having structural relationships). This set is then used to determine refactoring operations aiming at moving the code components in places of the system containing the higher number of similar components. For example, given a method m_i, it is possible to identify its most similar methods in the system and suggest to move m_i in the class containing the higher number of m_i's similar methods. Clearly, if the identified class coincides with the original class (i.e., the class in which m_i is already implemented), no refactoring is required.

We instantiate our approach to Move Method and Move Class refactoring in Sections 6 and 7, respectively.

4

Extract Class Refactoring

The material in this Chapter has been presented in (1, 115, 116).

4.1 Introduction

As discussed in Chapter 2, Extract Class refactoring is a technique for splitting classes with many responsibilities (i.e., Blobs) into different classes. Two good heuristics were proposed in (3) for class extraction:

- *responsibility-based*: identifying a subset of the data and a subset of the methods having similar responsibilities, i.e., having high (syntactic and semantic) cohesion;
- *change-based*: identifying subsets of the data attributes that usually change together or are dependent on each other.

Following the responsibility-based heuristic, several approaches have been proposed to support the Extract Class refactoring. In (1) we proposed an approach based on graph theory that is able to split a class with low cohesion in two classes having a higher cohesion, using a MaxFlow-MinCut algorithm. An important limitation of this approach is that often classes need to be split into more than two classes. Such a problem can be mitigated using partitioning or hierarchical clustering algorithms. However, such algorithms suffer from important limitations as well. The partitioning algorithms require as input the number of clusters, i.e., the number of classes to be extracted, while the hierarchical clustering algorithms requires a threshold to cut the dendogram. Unfortunately, no heuristics have been derived to suggest good default values for all these

parameters. Indeed, in (19) the authors tried to mitigate this issue by proposing different refactoring opportunities that can be obtained using different thresholds. However, this approach requires an additional effort by the software engineer who has to analyze different solutions in order to identify the one that provides the most adequate division of responsibilities.

To overcome these limitations we instantiated the approach aimed at decomposing complex object presented in Section 3.4 to the Extract Class refactoring. Given a class to be refactored, the approach computes a similarity measure between all possible pairs of methods in the class. This is a composite measure that captures relationships between methods, which impacts class cohesion (e.g., attribute references and semantic content) and coupling (e.g., method calls). Then, a weighted graph is built where each node represents a method and the weight of an edge that connects two nodes is given by the similarity of the two methods. The higher the similarity between two methods, the higher the likelihood that they should be in the same class. The extract class refactoring method is composed of two steps. In the first step the proposed approach split the graph in disconnected subgraphs by filtering out spurious relationships between methods and then identifies chains of strongly related methods by performing a transitive closure of the filtered graph. The extracted chains are then refined by merging trivial chains (i.e., chains with very few methods) with non-trivial chains. Using the extracted chains of methods it is possible to create new classes—one for each chain—having higher cohesion than the original class.

The approach has been deeply assessed and evaluated in the following studies:

1. *Assessment of the configuration parameters of the approach*: we empirically assessed the proposed approach on a massive number of Blobs artificially created in five open-source software systems, namely ArgoUML[1], GanttProject[2], Eclipse[3], JHotDraw[4], and Xerces[5]. The artificial Blobs are created by merging classes of the original system and the proposed approach is used to reconstruct the original classes. This assessment was conducted to analyze the impact of the configuration parameters on the performances of the proposed approach. As a result of this

[1]http://argouml.tigris.org/
[2]http://www.ganttproject.biz/
[3]http://www.eclipse.org/
[4]http://www.jhotdraw.org
[5]http://xerces.apache.org/

study we define a heuristic based on Principal Component Analysis to identify a customized optimal configuration of the parameters of the approach based on the characteristics of the software system under analysis.

2. *Evaluation of the refactoring solutions proposed by our approach from a quality metrics and a developer's point of view*: we conducted a user study with 50 Master students asking them to rate the refactoring suggested by the proposed approach on 17 Blobs of two open-source systems, namely GanttProject and Xerces. In this study we also evaluated the impact of the refactoring operations proposed by our approach on the cohesion and coupling of the object systems.

3. *Evaluation of the usefulness of the refactoring solutions proposed by our approach*: we conducted a second user study involving 15 Master students. This study has been conducted on eleven classes identified in different versions of open source systems that actually underwent extract class refactoring by the developers. We identified these cases of extract class refactoring operations by using Ref-Finder (117), an existing tool able to identify the refactoring operations performed between two subsequent versions of the same system. We provided each subject the eleven classes together with the refactoring solution provided by our approach. Then, we asked the subjects to provide a qualitative feedback about the usefulness of the proposed refactoring solutions. Moreover, we also verified to what extent (i) subjects were able to approximate the refactoring performed on the same classes by the original developers given the suggestion of our approach and (ii) how well the refactoring solution suggested by our approach is able to approximate the refactoring made by the original developers.

The results show that (i) the refactoring solutions proposed by our approach strongly increases the cohesion of the refactored classes without leading to significant increases in terms of coupling, (ii) the refactoring solutions proposed by our approach are considered useful to developers performing extract class refactoring and (iii) the proposed approach is able to approximate a manually performed refactoring at 91% on average. In addition, we also compare the proposed extract class refactoring method with a previous approach proposed in (1), which uses the same graph representation of the class to be refactored but a different algorithm based on Max Flow-Min Cut. The results

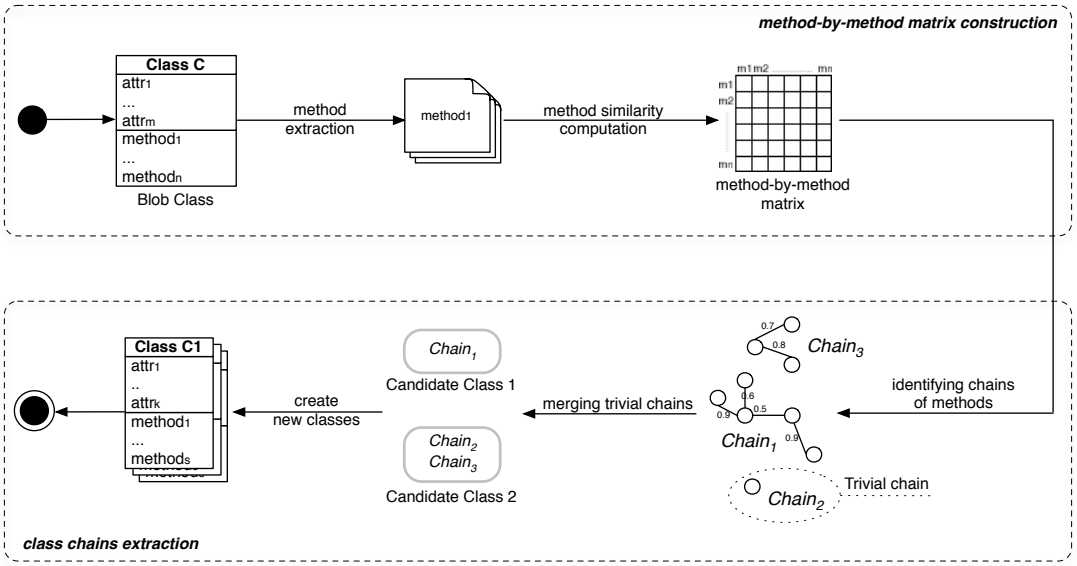

Figure 4.1: Class extraction process

clearly indicate that the new approach strongly outperforms the previous one. The experimental material and the raw data are available online for replication purposes(118).

4.2 The Approach

The approach takes as input a class previously identified by the software engineer (or automatically) as a candidate for refactoring. Figure 7.1 shows the Extract Class Refactoring process. In the top path of the process the candidate class is parsed to build a *method-by-method* matrix, a $n \times n$ matrix where n is the number of methods in the class to be refactored. A generic entry $c_{i,j}$ of the method-by-method matrix represents the likelihood that method m_i and method m_j should be in the same class. This phase of the refactoring process is described in Section 4.2.1.

Using the information in the *method-by-method* matrix the second part (bottom path) of the refactoring process shown in Figure 7.1 extracts the new classes from the input Blob. In particular, a filtering step is used to remove spurious links and split the initial graph represented in the *method-by-method* matrix into disconnected subgraphs. Then, the transitive closure of the *method-by-method matrix* is computed to identify chains of strongly related (or coupled) methods on the filtered graph. Each computed

chain represents a class to be extracted from the original class. However, some of these chains could have a very short length (trivial chains). To avoid the extraction of classes with a very low number of methods, in the second step of our algorithm we merge each trivial chain with the most coupled non trivial chain to obtain the final set of classes to be extracted from the original class. In subsections 4.2.2 and 4.2.3 we explain in detail these two steps of our algorithm.

4.2.1 Method-by-method matrix construction

The first phase of the refactoring process aims at building a method-by-method matrix representation of the class to be refactored, where a generic entry $c_{i,j}$ of the matrix represents the likelihood that method m_i and method m_j should be in the same class. This likelihood is computed as a hybrid coupling measure between methods (degree to which they are related) obtained by combining three structural and semantic measures, i.e., Structural Similarity between Methods (SSM) (110), Call-based Dependence between Methods (CDM) (1), and Conceptual Similarity between Methods (CSM) (55). The definition of these three measured can be found in Sections 3.2 (SSM and CDM) and 3.3 (CSM).

Since all the used similarity measures have values in $[0, 1]$, we compute the likelihood that methods m_i and m_j should be in the same class as:

$$c_{i,j} \quad = \quad w_{SSM} \cdot SSM(m_i, m_j) + w_{CDM} \cdot CDM(m_i, m_j) + w_{CSM} \cdot CSM(m_i, m_j)$$

where $w_{SSM} + w_{CDM} + w_{CSM} = 1$ and their values express the confidence (i.e., weight) in each measure.

It is worth noting that our choice of measures to use is not random, but it is based on the results previously achieved in (1), where we have shown that these measures are orthogonal, they capture different aspects of coupling between methods, and are suitable for automating extract class refactoring. However, the refactoring method defined in this paper is different than the refactoring method defined in (1), as it is able to split the original class in more than two classes. Therefore, we cannot use the same values of the weights identified in (1), but we need to empirically identify these values for the new method. The experimental assessment of the parameters of the approach is presented in Section 6.3.

4.2.2 Identifying Chains of Methods

The aim of this step is to remove from the graph represented by the *method-by-method* matrix spurious (but light) structural and/or semantic relationships between methods (119). Indeed, due to the use of the semantic similarity between methods (that very unlikely is equal to zero) the initial graph representation would be in general a complete graph (i.e., it contains all possible edges), or at least a connected graph. We split the graph representing the class to be refactored into disconnected subgraphs, containing strongly related methods. We filter the *method-by-method matrix*, based on a threshold *minCoupling*. All similarity values less than the threshold *minCoupling* are converted to zero:

$$\widetilde{c}_{i,j} = \begin{cases} c_{i,j} & \text{if } c_{i,j} > minCoupling; \\ 0 & \text{otherwise.} \end{cases}$$

There are many ways to define a threshold aimed at removing spurious relationships between methods. A simple classification allows identifying two different kinds of thresholds:

- *constant threshold*: the value of the threshold is fixed *a priori*, e.g., *minCoupling* = 0.1. This kind of threshold is simple to implement, but in general it is very difficult to choose *a priori* a constant value to prune spurious relationships. Indeed, the values in the *method-by-method* matrix depend on the Blob chosen to be refactored. In fact, there may be cases where the matrix contains a lot of high values. In this case, if the fixed threshold is high, it will probably remove the noise from the matrix, e.g., spurious relationships between the methods of the class. Otherwise, almost all the values will be left in the matrix. On the other hand, there may be cases where the matrix contains a large number of very low values. In this case, a high constant threshold will remove almost all the values from the matrix.

- *variable threshold*: the value of the threshold is automatically selected taking into account the characteristics of the given input. For example, *minCoupling* can be set as the median of the values present in the *method-by-method* matrix. This kind of threshold should resolve the problems derived by the use of a *constant threshold* and should ensure more stable filter performances across the different inputs. Choosing the best threshold in this case is also far from trivial.

We experimented with both *constant* and *variable* thresholds to empirically define a heuristic for selecting the best threshold (see Section 6.3 for details).

After filtering the *method-by-method matrix*, the transitive closure of the matrix is computed in order to extract chains of strongly coupled methods that represent the new classes to be extracted from the original class.

4.2.3 Merging Trivial Chains

The set of computed chains (i.e., extracted classes) may include chains with a very short length. To avoid the extraction of classes with a very low number of methods, we use a length threshold *minLength* to identify trivial chains, i.e., chains with a length less than *minLength*. In our approach we decided to set $minLength = 3$ since it is unusual that a class extracted from a Blob and implementing a well-defined set of responsibilities contains less than three methods. This minimum length can be easily changed if needed. Then, we compute the (structural and semantic) coupling between trivial and non-trivial chains and merge each trivial chain with the non-trivial chain it is most coupled with. The coupling between chains is calculated using the same measures used to calculate the coupling between methods. Specifically, the coupling between chains C_i and C_j is computed as the average coupling between all possible pairs of methods from C_i and C_j:

$$Coupling(C_i, C_j) = \frac{1}{|C_i| \times |C_j|} \sum_{m_i \in C_i, m_j \in C_j} c_{i,j}$$

where $|C_k|$ is the number of methods belonging to the chain C_k.

The methods of the original classes are distributed in different classes according to the extracted chains. The attributes of the original class are also distributed among the extracted classes according to how they are used by the methods in the new classes, i.e., each attribute is assigned to the new class having the higher number of methods using it[1]. At the end of the automated process, the extracted classes are analyzed by the software engineer who can accept the proposed restructuring as is, or change it by moving methods and attributes from one class to another.

[1]If a private field needs to be shared by two or more of the extracted classes, the implementation of the needed getter and/or setter methods is left to the developer.

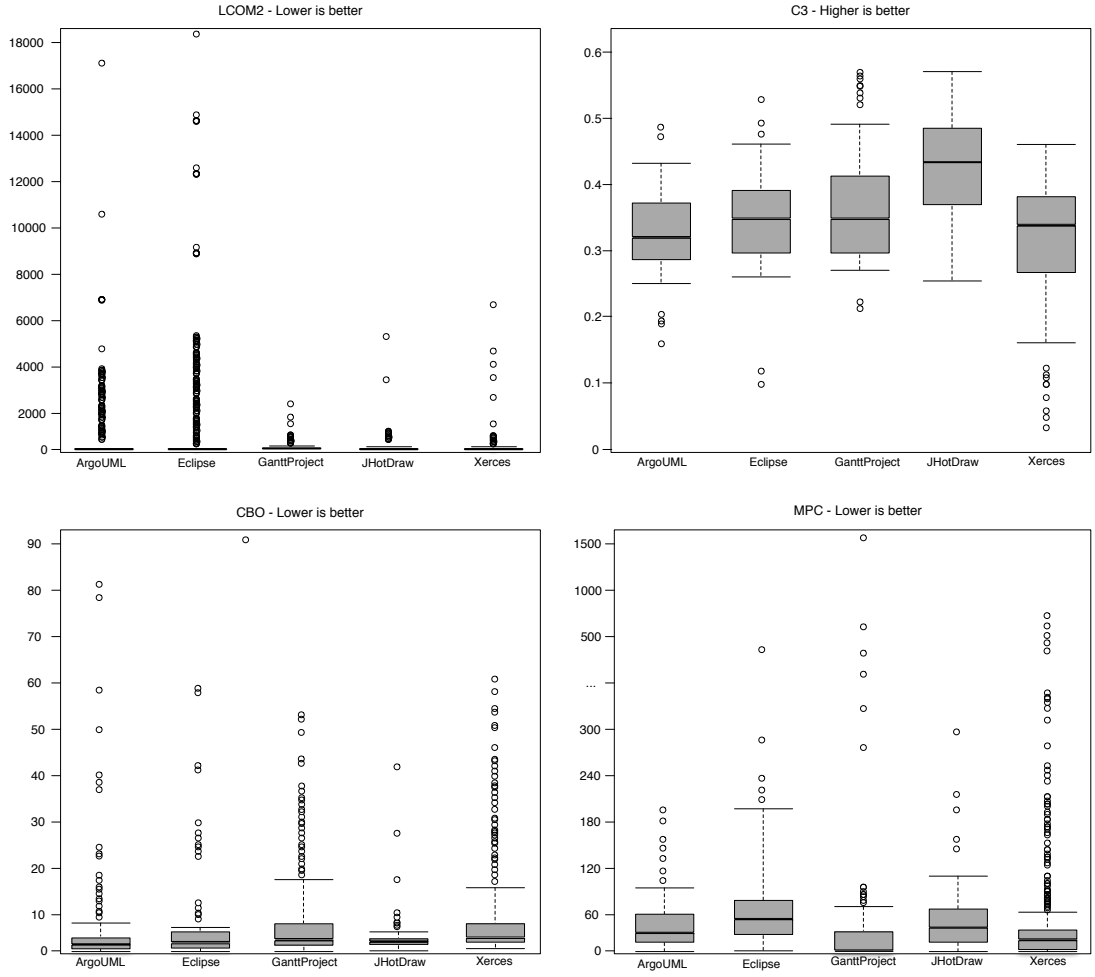

Figure 4.2: Box plots of quality metrics for the systems used in the case study.

4.3 Assessment of the Proposed Approach

The proposed approach has several configuration parameters, i.e., the weights of the similarity measures (w_{SSM}, w_{CDM}, and w_{CSM}) and the threshold used to prune out the spurious relationships between methods (*minCoupling*). While an assessment of these parameters has been made in (1), we cannot just use the previously achieved results, as the extract class refactoring algorithms are different and likely the values of these parameters have a different impact on the different algorithms. For this reason, in this section we conduct an empirical assessment of our approach with the goal of defining a heuristic to identify an optimal setting for these parameters.

The context of our study is represented by five open source software systems, namely ArgoUML 0.16, Eclipse 3.2, GanttProject 1.10.2, JHotDraw 6.0, and Xerces 2.7.0. ArgoUML (1,071 classes and 97 KLOC) is a UML modeling CASE tool with reverse engineering and code generation capabilities. Eclipse (23,462 classes and 1,710 KLOC) is a multi-language integrated development environment with an extensible architecture through plug-ins. GanttProject (273 classes and 28 KLOC) is a cross-platform desktop tool for project scheduling and management. JHotDraw (275 classes and 29 KLOC) is a Java GUI framework for structured drawing editors. Xerces (589 classes and 240 KLOC) is a family of packages for parsing and manipulating XML files. It implements a number of standard APIs for XML parsing, including DOM, SAX, and SAX2. Three of these systems, namely ArgoUML, JHotDraw, and Eclipse, have also been used to assess the parameters of the method proposed in (1).

Figure 4.2 reports the box plot for some commonly used class quality metrics, namely Lack of Cohesion of Methods (LCOM), Conceptual Cohesion of Classes (C3), Coupling Between Object classes (CBO), and Message Passing Coupling (MPC) calculated considering all the classes of the object systems. The LCOM metric counts the sets of methods in a class that are not related through the sharing of some of the fields of a class. It is an inverse metric—i.e., the higher the value of LCOM, the lower the class cohesion. C3 is a conceptual cohesion metric (60), complementary to structural cohesion, which exploits LSI (Latent Semantic Indexing) to compute the overlap of semantic information in a class expressed in terms of textual similarity among methods. Higher values of C3 indicate higher class cohesion. The CBO metric (9) represents the number of classes coupled to a given class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions. The higher the value of CBO, the higher the class coupling. Finally, the Message Passing Coupling (MPC) (109) is another 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.

The analysis of these metrics shows that the overall quality of the object systems, in terms of coupling and cohesion, is quite high and comparable to each other. Even if we do not have a quality model, this claim is supported by the comparable quality of

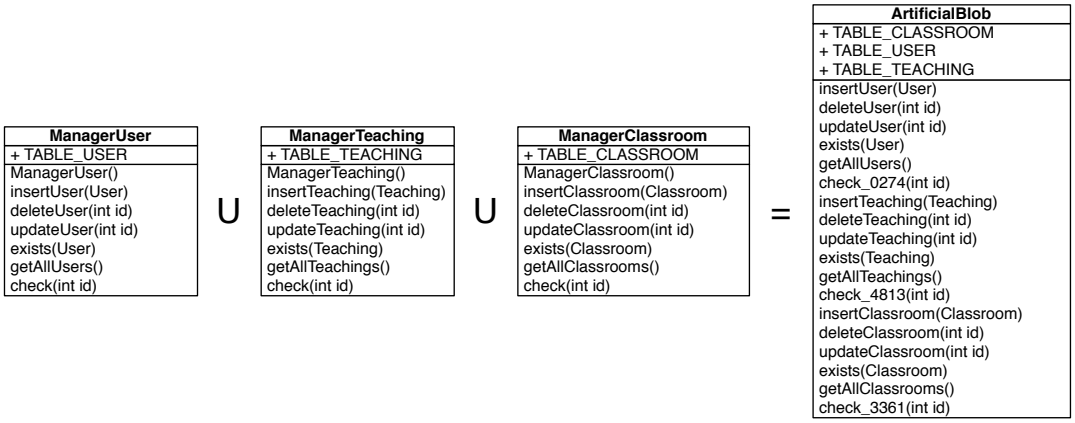

Figure 4.3: Example: creating an artificial Blob.

the object systems with JHotDraw, which has been developed as a "design exercise" and its design relies heavily on some well-known design patterns.

4.3.1 Planning and Execution

To analyze the influence of the configuration parameters we identified different refactoring solutions on the same classes using different weights for the adopted similarity measures and different values for the *minCoupling* threshold. For each metric weight we varied this parameter (**Weights**) starting at 0 and increasing it until 1 by a step of 0.1. We exercised all possible combinations of such values assuring that $w_{SSM} + w_{CDM} + w_{CSM} = 1$. Concerning the parameter *minCoupling* (**Threshold**) we experimented two kinds of thresholds: *constant* and *variable*. In particular, we used four different *constant thresholds* and three different *variable thresholds*. The *constant thresholds* we used are: 0.1, 0.2, 0.3, and 0.4. The *variable thresholds* we considered are: the first (Q_1), the second (Q_2), and the third (Q_3) quartile, respectively, of the non-zero values in the method-by-method matrix. Note that the use of quartiles allows to define a threshold that is less impacted—as compared to the other descriptive statistics (e.g., mean)—by problems caused by skewed distributions of values in the method-by-method matrix.

To have a high number of classes to assess the proposed approach, we artificially created Blob classes with more responsibilities and low cohesion from classes of the original systems. Specifically, we used a tool that randomly selects $m \geq 2$ classes of the

system—from the same package and/or from different packages—and merges them in a single class $\widehat{C}_m$. We selected different values for the number of classes to merge[1], i.e., $m \in \{2, 3\}$, and for each value of m we performed $n = 50$ different merging operations. Thus, on each of the five object systems, we created 100 artificial Blobs, 50 obtained by merging together 2 classes and 50 obtained by merging together 3 classes. The $\widehat{C}_m$ class is obtained by merging methods and attributes of the selected classes. We excluded the constructors of the classes when merging them in the artificial Blobs. If the classes to be merged contain methods having the same signature, we renamed these methods adding to their name a suffix composed by a random unique 4 digits number, e.g., *_0343*, and changed all the calls to them consistently. Finally, we ignored methods inherited from superclasses. Figure 4.3 shows an example of creating an artificial Blob by merging three different classes, namely ManagerUser, ManagerTeaching, and ManagerClassroom. All the merged classes have a method, i.e., *check(int id)*, with the same signature that has been renamed in the artificial Blob following the above rule. Moreover, all the constructors have been ignored in the creation of the Blob.

By construction the merged classes have a worse cohesion than the original classes (see the online Appendix (118) for the details). Note also that the randomly selected classes are merged only if their cohesion is higher than the average class cohesion in the system. The choice of this threshold was guided by the analysis of the box plots reported in Figure 4.2. As we can see most of the classes of the object systems have a good cohesion but there is a small set of outliers with a really low cohesion. By considering the average cohesion as a threshold we exclude from our set of classes these outliers, ensuring that the quality of the selected classes is rather good.

Once the mutated system is obtained, the proposed approach is applied to each artificial Blob with the goal of reconstructing the original classes previously merged. This is why it was important to select classes with high cohesion, because we can consider them as the "golden standard". Hence, to evaluate the results, the refactored classes are compared with the original classes aiming at identifying the total number of methods correctly and incorrectly moved in the split classes. To measure the accuracy of the refactoring solutions we computed the MoJo eFfectiveness Measure (MoJoFM)

[1]It is worth noting that while the experimental design is the same, the method proposed in (1) was experimented only on artificial Blobs created merging two classes, as it is only able to split a Blob in two classes.

(120) between the original classes and those extracted by our approach. The MoJoFM is a normalized variant of the MoJo distance and it is computed as follows:

$$MoJoFM(A,B) = 1 - \frac{mno(A,B)}{max(mno(\forall A,B))}$$

where $mno(A,B)$ is the minimum number of *Move* or *Join* operations one needs to perform in order to transform the partition A into B, and $max(mno(\forall\ A,B)$ is the maximum possible distance of any partition A from the gold standard partition B. Thus, $MoJoFM$ returns 0 if a clustering algorithm produces the farthest partition away from the gold standard; it returns 1 if a clustering algorithm produces exactly the gold standard.

Summarizing, we performed on each object system 924 experiments, i.e., all the possible combinations of weights, thresholds, and number of classes to be merged, leading to 231,000 refactoring operations. The large number of refactoring operations required to exercise the configuration parameters made a manual evaluation prohibitive and justifies our choice to refactor artificial Blobs and compare the results automatically with the original classes.

4.3.2 Analysis of the Results and Heuristics to Define the Configuration Parameters

Tables 4.1 and 4.2 report the best results—in terms of **MoJoFM**—achieved using *constant* and *variable thresholds* respectively[1]. The analysis of the results reveals that:

- *the variable threshold generally provides better performances than the constant threshold for the definition of minCoupling.* We obtained comparable results between constant and variable thresholds only on GanttProject. On the other systems, the *variable thresholds* provide an average improvement in terms of **MoJoFM** of about 0.06. In addition, unlike the constant threshold, the best results are always achieved using as variable threshold the median (Q_2) of the values of the matrix on all the systems. In other words, the *variable thresholds* ensure a more stable filtering performance across the different inputs, i.e., the different artificial Blobs to be refactored. Regarding the *constant thresholds*, generally better results can be achieved using a low value. As we can see in Table 4.1, none of the best configurations use 0.4 as the constant threshold;

[1]The complete results achieved with all possible combinations of parameters can be found in (118).

Table 4.1: Best results achieved using *constant thresholds*

System	Best Configuration	Merging 2 Classes		
		Mean	Median	Std.dev.
ArgoUML	$w_{CDM} = 0.1$, $w_{SSM} = 0.3$, $w_{CSM} = 0.6, minCoupling = 0.3$	0.817	0.859	0.150
Eclipse	$w_{CDM} = 0.0$, $w_{SSM} = 0.8$, $w_{CSM} = 0.2, minCoupling = 0.1$	0.795	0.801	0.136
GanttProject	$w_{CDM} = 0.3$, $w_{SSM} = 0.5$, $w_{CSM} = 0.2, minCoupling = 0.1$	0.865	0.899	0.203
JHotDraw	$w_{CDM} = 0.3$, $w_{SSM} = 0.6$, $w_{CSM} = 0.1, minCoupling = 0.1$	0.810	0.877	0.189
Xerces	$w_{CDM} = 0.1$, $w_{SSM} = 0.3$, $w_{CSM} = 0.6, minCoupling = 0.1$	0.768	0.756	0.133
System	Best Configuration	Merging 3 Classes		
		Mean	Median	Std.dev.
ArgoUML	$w_{CDM} = 0.2$, $w_{SSM} = 0.4$, $w_{CSM} = 0.4, minCoupling = 0.2$	0.722	0.818	0.232
Eclipse	$w_{CDM} = 0.1$, $w_{SSM} = 0.4$, $w_{CSM} = 0.5, minCoupling = 0.2$	0.651	0.674	0.134
GanttProject	$w_{CDM} = 0.1$, $w_{SSM} = 0.6$, $w_{CSM} = 0.3, minCoupling = 0.1$	0.765	0.795	0.145
JHotDraw	$w_{CDM} = 0.3$, $w_{SSM} = 0.4$, $w_{CSM} = 0.3, minCoupling = 0.2$	0.745	0.756	0.132
Xerces	$w_{CDM} = 0.1$, $w_{SSM} = 0.6$, $w_{CSM} = 0.3, minCoupling = 0.1$	0.662	0.691	0.161

- *the combination of structural and semantic measures considerably improves the accuracy of our approach.* As expected, the best results are achieved when all the weights of the three cohesion metrics are greater than zero. This means that the combination of structural and semantic measures is worthwhile, thus confirming the findings achieved in (1).

- *the optimal settings of the weights of the three cohesion measures is not stable across the object systems.* The results highlight that the best configuration of weights sensibly changes across the object systems. Unlike in (1), where in general the best performances were achieved giving a high weight (greater than 0.6) to the semantic similarity measure, in this experimental setting it is quite difficult to identify an optimal setting of the weights for the three measures that can be used in general for any system. This means that a different heuristic is required to identify an optimal setting of the weights for different systems.

To better understand how the parameters of the proposed approach affect our results, we statistically analyzed the influence of the factors **Weights** and **Threshold** on the reconstruction accuracy of our approach (**MoJoFM**) through interaction plots[1]. The interaction plots confirmed that generally the best performances can be obtained using as threshold the median, i.e., Q_2, of the non-zero values of the method-by-method matrix and both structural and semantic measures. However, the results also confirmed

[1]The interested reader can find the interaction plots for all systems in our online appendix (118).

Table 4.2: Best results achieved using *variable thresholds*

System	Best Configuration	Merging 2 Classes		
		Mean	Median	Std.dev.
ArgoUML	$w_{CDM} = 0.1$, $w_{SSM} = 0.3$, $w_{CSM} = 0.6, minCoupling = Q_2$	0.868	0.940	0.181
Eclipse	$w_{CDM} = 0.1$, $w_{SSM} = 0.3$, $w_{CSM} = 0.6, minCoupling = Q_2$	0.901	0.913	0.115
GanttProject	$w_{CDM} = 0.3$, $w_{SSM} = 0.3$, $w_{CSM} = 0.4, minCoupling = Q_2$	0.873	0.865	0.138
JHotDraw	$w_{CDM} = 0.3$, $w_{SSM} = 0.2$, $w_{CSM} = 0.5, minCoupling = Q_2$	0.904	0.948	0.152
Xerces	$w_{CDM} = 0.3$, $w_{SSM} = 0.2$, $w_{CSM} = 0.5, minCoupling = Q_2$	0.830	0.831	0.158
System	Best Configuration	Merging 3 Classes		
		Mean	Median	Std.dev.
ArgoUML	$w_{CDM} = 0.2$, $w_{SSM} = 0.4$, $w_{CSM} = 0.4, minCoupling = Q_2$	0.767	0.792	0.200
Eclipse	$w_{CDM} = 0.1$, $w_{SSM} = 0.4$, $w_{CSM} = 0.5, minCoupling = Q_2$	0.749	0.741	0.170
GanttProject	$w_{CDM} = 0.4$, $w_{SSM} = 0.2$, $w_{CSM} = 0.4, minCoupling = Q_2$	0.750	0.718	0.160
JHotDraw	$w_{CDM} = 0.2$, $w_{SSM} = 0.3$, $w_{CSM} = 0.5, minCoupling = Q_2$	0.773	0.730	0.172
Xerces	$w_{CDM} = 0.4$, $w_{SSM} = 0.2$, $w_{CSM} = 0.4, minCoupling = Q_2$	0.683	0.685	0.195

that the weights that produce optimal results are different across the different object systems.

For this reason, we use Principal Component Analysis (PCA) of the method coupling data, in order to identify a heuristic able to set-up different configurations of the weights for different software systems that result in near-optimal performances of the refactoring technique. This allows to identify the different dimensions that describe a phenomenon (e.g., the coupling between pairs of methods) and obtain an indication of the importance of each dimension (captured by one or more coupling measures) in the description of this phenomenon (i.e., the proportion of variance). Table 4.3 shows the results of the PCA on all the object systems. As we can see, the semantic measure is identified by the PCA as the measure that describes most of the coupling between pairs of methods. In particular, the proportion of variance for the semantic similarity measure is higher than 0.6 for all the object systems. Moreover, in general, both structural measures are important, as they describe some of the relationships between pairs of methods. This confirms the finding previously highlighted from the analysis of Tables 4.1 and 4.2.

As expected, the proportion of variance values are rather different across the different systems, so our question was whether using the proportion of variance values to define the weights of the similarity measures provides results of the MoJoFM close to the optimal results shown in Table 4.2. Table 5.5 compares the results obtained using the configuration parameters identified by the PCA proportion of variance (*PCA-based con-*

Table 4.3: Results of PCA: Rotated Components

	PC1	PC2	PC3
Proportion of Variance	0.70	0.27	0.03
Cumulative Proportion	0.70	0.97	1.00
CDM	-0.02	0.00	0.99
SSM	-0.15	-0.98	0.00
CSM	-0.99	0.15	-0.02

(a) ArgoUML

	PC1	PC2	PC3
Proportion of Variance	0.62	0.21	0.17
Cumulative Proportion	0.62	0.83	1.00
CDM	0.02	-0.99	0.14
SSM	0.03	-0.14	-0.99
CSM	0.99	0.02	0.03

(b) GanttProject

	PC1	PC2	PC3
Proportion of Variance	0.80	0.10	0.10
Cumulative Proportion	0.80	0.90	1.00
CDM	0.07	-0.02	0.99
SSM	0.06	-0.99	-0.02
CSM	0.99	0.06	-0.06

(c) JHotDraw

	PC1	PC2	PC3
Proportion of Variance	0.66	0.24	0.10
Cumulative Proportion	0.66	0.90	1.00
CDM	-0.04	0.02	0.99
SSM	-0.26	-0.96	0.00
CSM	-0.96	0.27	-0.05

(d) Xerces

	PC1	PC2	PC3
Proportion of Variance	0.73	0.25	0.02
Cumulative Proportion	0.73	0.98	1.00
CDM	-0.02	0.04	0.99
SSM	-0.58	-0.81	0.02
CSM	-0.81	0.58	-0.04

(e) Eclipse

figuration) with the best results obtained in our experimentation (*best configuration*). As we can see, the difference between the reconstruction accuracy of the *PCA-based configuration* compared with the accuracy obtained using the best configuration is very small. Indeed the difference of MoJoFM is never higher than 0.04. We also executed the Wilcoxon test to compare the accuracy of the two different configurations. The results on all object systems do not highlight any statistically significant difference. This result indicates that the *PCA-based configuration* provides an accuracy similar to the best accuracy obtained by exercising all possible parameter configurations. Given these findings we propose the following heuristics to set the parameters of our approach in a real usage scenario:

- *minCoupling*: use the median of the non-zero values of the method-by-method matrix as threshold to remove spurious relationships between the methods of the class to refactor.

- *weights*: the weights assigned to the structural and semantic measures are established based on the system under analysis by performing the PCA of the values

Table 4.4: Results reconstructing merged classes: PCA based *vs* best configuration

System	#Merged Classes	Best configuration	PCA-based configuration
ArgoUML	2	$w_{CDM} = .1$ $w_{SSM} = .3$ $w_{CSM} = .6$ (**.87**)	$w_{CDM} = .0$ $w_{SSM} = .3$ $w_{CSM} = .7$ (**.84**)
	3	$w_{CDM} = .2$ $w_{SSM} = .4$ $w_{CSM} = .4$ (**.77**)	$w_{CDM} = .0$ $w_{SSM} = .3$ $w_{CSM} = .7$ (**.75**)
Eclipse	2	$w_{CDM} = .1$ $w_{SSM} = .3$ $w_{CSM} = .6$ (**.90**)	$w_{CDM} = .0$ $w_{SSM} = .3$ $w_{CSM} = .7$ (**.88**)
	3	$w_{CDM} = .1$ $w_{SSM} = .4$ $w_{CSM} = .5$ (**.75**)	$w_{CDM} = .0$ $w_{SSM} = .3$ $w_{CSM} = .7$ (**.71**)
GanttProject	2	$w_{CDM} = .3$ $w_{SSM} = .3$ $w_{CSM} = .4$ (**.87**)	$w_{CDM} = .2$ $w_{SSM} = .2$ $w_{CSM} = .6$ (**.84**)
	3	$w_{CDM} = .4$ $w_{SSM} = .2$ $w_{CSM} = .4$ (**.75**)	$w_{CDM} = .2$ $w_{SSM} = .2$ $w_{CSM} = .6$ (**.74**)
JHotDraw	2	$w_{CDM} = .3$ $w_{SSM} = .2$ $w_{CSM} = .5$ (**.90**)	$w_{CDM} = .1$ $w_{SSM} = .1$ $w_{CSM} = .8$ (**.87**)
	3	$w_{CDM} = .2$ $w_{SSM} = .3$ $w_{CSM} = .5$ (**.77**)	$w_{CDM} = .1$ $w_{SSM} = .1$ $w_{CSM} = .8$ (**.74**)
Xerces	2	$w_{CDM} = .3$ $w_{SSM} = .2$ $w_{CSM} = .5$ (**.83**)	$w_{CDM} = .1$ $w_{SSM} = .2$ $w_{CSM} = .7$ (**.79**)
	3	$w_{CDM} = .3$ $w_{SSM} = .3$ $w_{CSM} = .4$ (**.68**)	$w_{CDM} = .1$ $w_{SSM} = .2$ $w_{CSM} = .7$ (**.67**)
In parenthesis the reconstruction accuracy, i.e., the average MoJoFM			

of the similarity measures computed on all the classes of the system. The value of the proportion of variance obtained for each measure will be used as the weight for the corresponding measure.

4.3.3 Threats to Validity

The results achieved show a high reconstruction accuracy of the proposed approach. A threat that could affect the validity of such a result is represented by the fact that our approach was applied on artificial Blobs and thus reconstructing previously merged classes might be trivial. To mitigate such a threat we analyzed the coupling between the classes to be merged in order to understand if there is a correlation with the reconstruction accuracy of our approach. If two merged classes have no coupling between them, then the outcome of splitting might be close to perfect. On the other hand if their coupling is high it might be "translated" as similarity between the members of the merged class, affecting the results. We used the Conceptual Coupling Between Classes (CCBC) (55) and the information-flow-based coupling (ICP) (108) to measure the coupling between the merged classes. Then, we measured the statistical correlation between the coupling of the merged classes and the splitting accuracy by computing the Pearson product-Moment Correlation Coefficient (PMCC) (121). The results revealed no correlation on all the object systems.

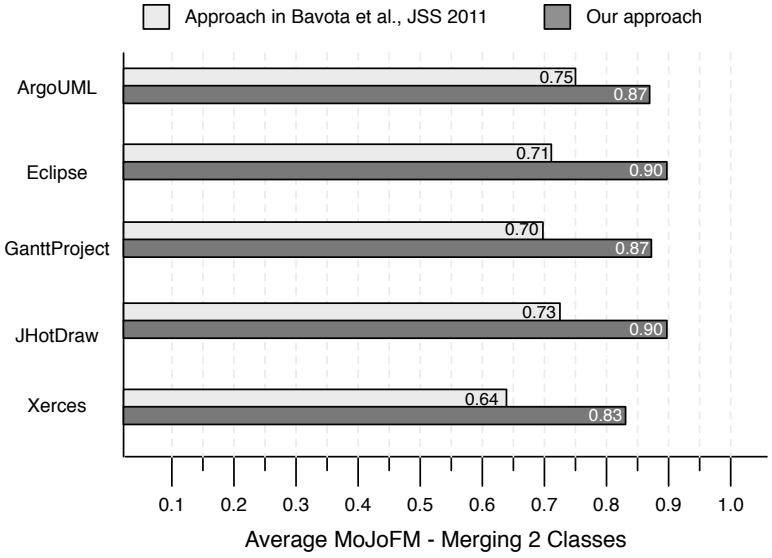

Figure 4.4: Comparison between our approach and the approach presented in (1).

Table 4.5: Mann-Whitney test: our approach *vs* approach presented in (1)

	ArgoUML	Eclipse	GanttProject	JHotDraw	Xerces
Statistically significant difference	Yes (0.03)	Yes (< 0.01)	Yes (< 0.01)	Yes (< 0.01)	Yes (< 0.01)

Moreover, to further mitigate this threat we compared the reconstruction accuracy of the proposed approach with the accuracy achieved by the approach presented in (1), which is based on the same graph-based representation of a class. Since the approach presented in (1) is only able to split a Blob in two classes we performed this comparison only on the artificial Blobs created merging two classes.

Figure 4.4 reports the results achieved using for both the approaches the best configuration of parameters, respectively. As we can see the reconstruction accuracy of our new approach always overcomes the reconstruction accuracy obtained with the approach we proposed in (1). In particular, the average difference of MoJoFM is 16.8%. Note that our new approach is not only able to improve the reconstruction accuracy of the approach we proposed in (1), but it also automatically derives that the artificial Blobs have to be split in two classes, whereas the approach presented in (1) just split the artificial Blobs in two classes by construction.

We also statistically analyzed the performances of the two approaches using the

Mann-Whitney test (122). We chose this test as we cannot assume normality of data and the test does not make normality assumptions. In particular, we used the test to analyze the statistical significance of the difference between the reconstruction accuracy provided by the two approaches. The results were intended as statistically significant at $\alpha = 0.05$. Table 4.5 reports the achieved results. As we can see, the reconstruction accuracy of our new approach is significantly higher than the reconstruction accuracy achieved by the approach presented in (1), for each system.

This result might be surprising since the two approaches use the same structural and semantic measures and the same graph-based representation of the class to split. The only difference is in the algorithm adopted to split the graph into sub-graphs. Thus, to understand the reasons for performance gap between the two approaches, it is important to point out the differences between the two algorithms:

1. Like the extract class refactroing algorithm presented in this thesis, also the Max Flow-Min Cut algorithm in (1) is preceded by a filtering step aiming at removing spurious connections between nodes. Indeed, due to the use of the semantic similarity between methods (that very unlikely is equal to zero) the initial graph representation would be in general a complete graph (i.e., it contains all possible edges). In this case, the Max Flow-Min Cut algorithm would always split a graph containing n nodes in two graphs containing $n-1$ and 1 node, respectively (123). So, filtering and removing some edges is needed in this case to avoid a trivial application of Max Flow-Min Cut algorithm. However, filtering in this case does not disconnect the graph, as splitting the graph is in charge of the Max Flow-Min Cut Algorithm.

   On the other hand, the filtering step of our new approach is much more intensive, as it aims at splitting the graph in subgraphs representing loosely coupled components. So this step is a key for our new extract class refactoring method. The other steps of the new method consists of (i) applying the transitive closure to identify chains of nodes belonging to the same subgraph (and then methods belonging to the same class) and (ii) aggregating the small subgraphs (i.e., the trivial chains composed of less than 3 methods) with the most coupled non-trivial chain previously identified. This merging step is also very important to correct some surplus of the filtering step.

2. The Max Flow-Min Cut algorithm needs as input the source and sink nodes that ideally represent two methods belonging to the two different classes to be extracted from the Blob. In (1) the heuristic used to identify the source and sink nodes consists of selecting the two nodes in the graph connected by the edge with the lowest weight, i.e., they are the two less coupled methods (according to the used structural and semantic similarity measures) in the Blob class. Clearly, in some cases this heuristic might not work properly and select two methods that should instead be in the same class. In this case the splitting performed by the algorithm will be negatively affected, since guided by wrong initial assumptions. Our new technique does not suffer of similar problems, as the splitting is performed by the filtering step.

We also tried to iteratively use the approach presented in (1) to refactor the artificial Blobs created by merging three classes (M_1, M_2, and M_3) together. In this case, in the first iteration the Max Flow-Min Cut algorithm would split the artificial Blob in two classes E_1 and E_2. Therefore, to be useful in an iterative usage, one of the extracted class (suppose E_1) should contain most of the methods of one of the original classes (suppose M_1) while the second extracted class (E_2) should contain most of the methods of the other two original classes (M_2 and M_3). The approach should then be re-applied to E_2 in order to extract M_2 and M_3 thus reconstructing the original classes. However, this rarely happens, and the distribution of methods of the three original classes to the classes E_1 and E_2 achieved after the first iteration is usually more smoothed. To verify this, we applied the approach presented in (1) on the artificial Blob in the first iteration and on both the extracted classes in the second iteration. Then, we selected as refactoring solution the one achieving the highest reconstruction accuracy (i.e., the highest MoJoFM) between the two generated. For example, suppose that E_1 and E_2 are the two classes extracted from the artificial Blob at the first iteration. In the second iteration we apply the Max Flow-Min Cut approach on both E_1 and E_2 obtaining the classes E_3 and E_4 extracted from E_1 and E_5 and E_6 extracted from E_2. We then compute the reconstruction accuracy of the following two set of classes: $S_1 = \{E_1, E_5, E_6\}$ and $S_2 = \{E_2, E_3, E_4\}$. Supposing that the MoJoFM achieved by S_1 is 0.7 while the one achieved by S_2 is 0.6, S_1 is selected as the refactoring solution.

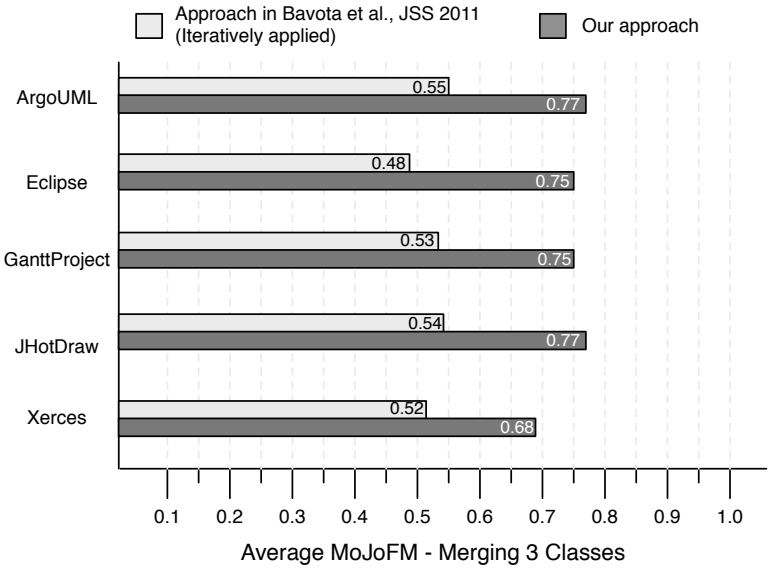

Figure 4.5: Comparison between our approach and the approach presented in (1) when applied iteratively.

Figure 4.5 reports the achieved results. As we can see the iterative application of Max Flow-Min Cut produces results that in the best case are far from the performances achieved by our new technique. The gap of performances with our approach in this scenario is really high. In particular, our approach obtained a reconstruction accuracy in terms of MoJoFM higher than the approach proposed in (1) of about 22% on average.

4.4 Evaluating the Quality of the Refactoring Solutions

The assessment performed on artificial Blobs discussed in Section 6.3 allowed us to configure the parameters of our approach. In this study we evaluate our approach (configured using the PCA-based heuristic defined in Section 6.3) with quality metrics and from a developer's point of view.

In particular, we conducted two experiments with a total of 50 subjects to quantitatively assess how much the refactoring solution suggested by our approach is considered as a good division of the responsibilities implemented in a Blob class. The proposed approach was used to refactor actual Blobs from two open source software systems, namely GanttProject and Xerces. To set the parameters of our approach we used the heuristics presented in Section 6.3. In particular, for GanttProject the configuration is

$w_{CDM} = 0.2$, $w_{SSM} = 0.2$, $w_{CSM} = 0.6$, and $minCoupling = Q_2$, while for Xerces is $w_{CDM} = 0.1$, $w_{SSM} = 0.2$, $w_{CSM} = 0.7$, and $minCoupling = Q_2$. This study was conducted only on GanttProject and Xerces because a set of (manually identified) Blobs for these systems (ten for Xerces and seven for GanttProject) is reported in the literature (124). In particular, in the first user study a subset of 30 subjects performed the experimentation on Blobs from the GanttProject system while in the second user study 20 subjects performed the experimentation on Blobs from the Xerces system.

4.4.1 Research Questions and Planning

In the context of our study, the following research questions were formulated:

- RQ_1: What is the impact of the refactoring suggested by our approach on class cohesion and coupling?
- RQ_2: Does the proposed refactoring results in a better division of responsibilities from a developer's point of view?

To respond to our first research question (RQ_1) we analyzed the changes in terms of cohesion and coupling in the object systems when applying the refactoring operations suggested by our approach. We expect an increase of cohesion (desired effect) due to the split in different classes of the responsibilities implemented in the Blobs. However, we also expect an increase of coupling (side effect) since splitting a class in several classes usually results in an increment of the total dependencies between classes. For these reasons coupling and cohesion should be measured together to make a proper judgment on the complexity and quality of a system, since improvement of cohesion usually happens at the expense of increase in coupling and *vice versa* (125). To measure the cohesion of the analyzed classes we used the LCOM and the C3 metrics while for the coupling we used the MPC metric, since it allows to understand if the interactions due to method calls between classes is increased after the refactoring operations suggested by our approach. The cohesion and coupling metrics were measured before and after refactoring. In particular, we measured the LCOM and C3 method of the Blob classes and of the extracted classes to quantify the increment in terms of cohesion achieved through the extract class refactoring. As for the coupling, also in this case we measured the MPC for the Blob classes as well as for the extracted classes to quantify the new

dependencies existing between the classes extracted from the refactored Blob. However, there might also be an increase of coupling in the system due to new dependencies existing between client classes using methods in the classes extracted from the Blob. To verify this possible side effect, we also measured the changes in coupling as the sum of MPC value for all the classes in the system affected by the refactoring (we will refer to this measured value as MPC_{sum} in the text). To detect the classes affected by each of the 17 refactoring operations we apply them by using the extract class functionality of Eclipse. As benchmark, we compared the results obtained applying our new approach with those obtained using the approach presented in (1).

With regards to the research question RQ_2 we analyzed the refactoring operations proposed by our approach from the developers' point of view. To this aim, we performed two experiments involving a total of 50 Master Computer Science students from the University of Salerno. Before the experiment students attended a two hours seminar about the most common refactoring techniques, their objectives and usefulness during the software lifecycle. During the semester in which the experimentation has been carried out students were attending the courses of *Advanced Software Engineering*, *Advanced Databases*, *Programming Languages and Compilers*, and *Advanced Computer Networks*. As for their background, all students had in their Bachelor curriculum an exam of Object Oriented programming (in Java) and of software engineering. Students voluntarily participated to the study and no selection process was performed (i.e., all the voluntary students were accepted). Finally, during the experiment students were allowed to leave but no one did.

The first experiment involved 30 students who evaluated three different refactoring operations for each of the seven Blobs of the GanttProject system: (i) the refactoring suggested by our approach, (ii) the refactoring suggested by the approach presented in (1), and (iii) a random refactoring. The second option was used to provide the students with an alternative refactoring solution which makes sense but is likely worse than the refactoring solution suggested by our approach (at least according to the results obtained in Section 6.3). The last option does not make sense as a refactoring solution and was only considered to verify whether participants seriously considered this assignment (i.e., a sanity check). For each of the proposed refactoring the students had to express their level of agreement to the claim "*The proposed refactoring results in a better division of responsibilities*" proposing a score using a Likert scale (126): 1:

Strongly disagree; 2: *Disagree*; 3: *Neutral*; 4: *Agree*; 5: *Fully agree*. The students had 140 minutes to perform the assigned task (on average 20 minutes for each Blob). The second experiment was conducted using the same design, but involved 20 subjects and was performed on the ten Blobs of the Xerces system with a time limit of 200 minutes (same 20 minutes average for each Blob).

To answer the research question RQ_2, the results achieved in the two experiments were analyzed through boxplots and statistical tests. As for the statistical analysis, we decided to use the Mann-Whitney test (122) since we cannot assume normality of the data. We collected the ranking for each of the three proposed refactoring solutions. Then, for each pair of considered approaches (e.g., our new approach *vs.* the random refactoring), we used the Mann-Whitney test to analyze the statistical significance of the difference between the scores assigned by the students to the refactoring solutions of the two approaches. The results were intended as statistically significant at $\alpha = 0.05$.

4.4.2 Analysis of the Results

Table 4.6 reports information about the Blobs object of our study before and after the refactoring suggested by our approach, and in particular the LOC and the number of methods[1].

4.4.3 Results of the metrics based evaluation (RQ_1)

Table 7.6 reports the results achieved in our study in terms of cohesion while Table 4.8 reports the data about the coupling. Looking at Table 7.6 for almost all the classes the cohesion is sensibly improved. In particular, the cohesion for the refactored Blobs is on average more than five times better in terms of LCOM (1,310 for the Blobs, 257 for the extracted classes) and more than two times better in terms of C3 (0.13 for the Blobs, 0.27 for the extracted classes). Table 4.9 compares the results in terms of cohesion achieved by our approach with those achieved by the approach in (1). As we can see, on the same set of Blobs, the approach in (1) achieved for the refactored classes

[1]In the number of methods we do not count the constructors (for both pre- and post-refactoring), and eventual getters and setters methods to be add after the refactoring. In this way the sum of methods of the extracted classes is equals to the number of methods of the Blob class.

Table 4.6: Refactoring solutions proposed by our approach on the 17 Blobs object of our study.

System	Class	Split Classes	Pre-refactoring LOC	Pre-refactoring Methods	Post-refactoring LOC	Post-refactoring Methods
Xerces	AbstractDOMParser	2	1,775	45	492	15
					1,259	30
Xerces	AbstractSAXParser	3	1,360	55	110	11
					241	12
					967	32
Xerces	BaseMarkupSerializer	2	1,275	61	98	10
					1,123	51
Xerces	CoreDocumentImpl	3	1,497	119	82	11
					79	14
					1,350	94
Xerces	DeferredDocumentImpl	2	1,612	76	1,061	34
					430	42
Xerces	DOMNormalizer	2	1,291	31	33	13
					1,268	18
Xerces	DOMParserImpl	2	820	17	454	7
					431	10
Xerces	DurationImpl	2	953	44	151	16
					540	28
Xerces	NonValidatingConfiguration	2	403	18	123	3
					284	15
Xerces	XIncludeHandler	4	1,331	111	372	32
					440	26
					169	17
					524	36
GanttProject	GanttOptions	3	513	68	438	51
					72	11
					37	6
GanttProject	GanttProject	3	2,269	90	2,086	71
					124	6
					118	13
GanttProject	GanttGraphicArea	2	2,160	43	2,025	32
					197	11
GanttProject	GanttTree	2	1,730	48	1,382	42
					423	6
GanttProject	GanttTaskPropertiesBean	2	1,685	27	1,164	21
					524	6
GanttProject	ResourceLoadGraphicArea	2	1,060	29	873	21
					227	8
GanttProject	TaskImpl	3	329	46	234	27
					69	10
					44	9

a cohesion on average almost 3 times better in terms of LCOM and two times better

Table 4.7: Cohesion: Results obtained refactoring the 17 Blobs

System	Class	Pre-refactoring		Our approach	
		LCOM	C3	LCOM	C3
Xerces	AbstractDOMParser	83	0.21	0	0.25
				0	0.23
Xerces	AbstractSAXParser	1,126	0.09	49	0.22
				0	0.29
				451	0.19
Xerces	BaseMarkupSerializer	921	0.08	27	0.18
				358	0.15
Xerces	CoreDocumentImpl	6,825	0.05	143	0.19
				190	0.33
				3,322	0.12
Xerces	DeferredDocumentImpl	0	0.14	0	0.18
				41	0.20
Xerces	DOMNormalizer	456	0.08	66	0.33
				150	0.17
Xerces	DOMParserImpl	132	0.24	15	0.38
				54	0.33
Xerces	DurationImpl	701	0.11	211	0.22
				355	0.18
Xerces	NonValidatingConfiguration	147	0.04	4	0.31
				82	0.08
Xerces	XIncludeHandler	4,652	0.08	30	0.42
				602	0.14
				75	0.22
				188	0.27
Average Xerces		**1,504**	**0.11**	**256**	**0.23**
Gantt	GanttOptions	2,100	0.18	1,117	0.27
				295	0.32
				0	0.36
Gantt	GanttProject	2,318	0.08	1,233	0.16
				0	0.36
				0	0.37
Gantt	GanttGraphicArea	845	0.13	511	0.18
				4	0.29
Gantt	GanttTree	649	0.14	493	0.22
				15	0.36
Gantt	GanttTaskPropertiesBean	183	0.13	52	0.18
				4	0.44
Gantt	ResourceLoadGraphicArea	252	0.17	146	0.28
				63	0.35
Gantt	TaskImpl	884	0.27	119	0.31
				58	0.38
				3	0.41
Average Gantt		**1,033**	**0.16**	**242**	**0.31**
Overall Average		**1,310**	**0.13**	**257**	**0.27**

in terms of C3 than the Blob classes[1].

[1]The interested reader can find the results by the approach in (1) for each Blob in our online appendix (118).

Table 4.8: Coupling: Results obtained refactoring the 17 Blobs

System	Class	Pre-refactoring		Our approach	
		MPC	MPC_{sum}	MPC	MPC_{sum}
Xerces	AbstractDOMParser	561	1,483	221 356	1,503
Xerces	AbstractSAXParser	320	1,375	13 77 256	1,409
Xerces	BaseMarkupSerializer	355	655	7 350	661
Xerces	CoreDocumentImpl	341	1,693	15 10 319	1,705
Xerces	DeferredDocumentImpl	392	996	351 82	1,045
Xerces	DOMNormalizer	441	819	1 441	826
Xerces	DOMParserImpl	285	833	163 128	862
Xerces	DurationImpl	352	594	42 312	609
Xerces	NonValidatingConfiguration	91	441	32 60	453
Xerces	XIncludeHandler	573	1,650	58 302 96 132	1,724
Sum Xerces		**3,711**	**10,539**	**3,824**	**10,797**
Gantt	GanttOptions	472	1,047	443 20 19	1,077
Gantt	GanttProject	1,528	3,707	1,492 8 29	3,724
Gantt	GanttGraphicArea	575	1,319	605 4	1,362
Gantt	GanttTree	358	2,654	262 106	2,678
Gantt	GanttTaskPropertiesBean	276	432	179 149	485
Gantt	ResourceLoadGraphicArea	447	853	355 105	877
Gantt	TaskImpl	45	414	42 6 4	428
Sum Gantt		**3,701**	**10,426**	**3,828**	**10,631**
Overall Sum		**7,412**	**20,965**	**7,652**	**21,428**

Concerning the coupling we analyzed both the increase of coupling limited to the extracted classes (MPC column in Table 4.8) as well as the overall increase of coupling for all the classes involved in the refactoring operations (MPC_{sum} column in Table 4.8). As we can see, extracting different classes from the original Blobs resulted in a small

Table 4.9: Average Cohesion: our approach *vs.* approach in (1)

System	Pre-refactoring		Our approach		Approach in (1)	
	LCOM	C3	LCOM	C3	LCOM	C3
Xerces	1,504	0.11	256	0.23	588	0.21
Gantt	1,033	0.16	242	0.31	310	0.27
Overall	**1,310**	**0.13**	**257**	**0.27**	**473**	**0.24**

Table 4.10: Average Coupling: our approach *vs.* approach in (1)

System	Pre-refactoring		Our approach		Approach in (1)	
	MPC	MPC_{sum}	MPC	MPC_{sum}	MPC	MPC_{sum}
Xerces	3,711	10,539	3,824	10,797	3,807	10,757
Gantt	3,701	10,426	3,828	10,631	3,780	10,569
Overall	**7,412**	**20,965**	**7,652**	**21,428**	**7,587**	**21,326**

increment of the MPC value of the extracted classes. For instance, the refactoring of *XIncludeHandler* generated 4 different classes with a cohesion much higher than the cohesion of the original class, e.g., LCOM for the original class is 4,652, while the extracted classes have an average LCOM equals to 224 (more than 20 times better, note that LCOM is an inverse measure of cohesion). On the coupling side the MPC for the original Blob is 573, while the sum of the MPC of the extracted classes is 588. Thus, the percentage increase in terms of MPC is only about 3%. Overall, the average increment of coupling limited to the comparison of the MPC for the original Blobs and of the MPCs for the extracted classes is only +3.2%. This result, as shown in Table 4.10, is just slightly higher than those achieved by the approach in (1) (+2.4%). Note that the (slightly) better performances in terms of coupling ensured by the approach in (1) are an expected results, since it extracts an overall number of classes from the Blobs lower than our approach (i.e., 34 against 41). This clearly results in less dependencies existing between the extracted classes.

As for the coupling measured for all classes involved in the refactoring operations here the increase is very small in terms of percentage. In particular, our approach increases the number of dependencies for these classes from 20,965 to 21,428 (+2.2%) while the approach in (1) from 20,965 to 21,326 (+1.7%).

Summarizing, the application of the refactoring operations suggested by our approach results in a strong increase of cohesion much higher than those achieved by the approach in (1). The price to pay in terms of coupling is quite low and similar for both the approaches.

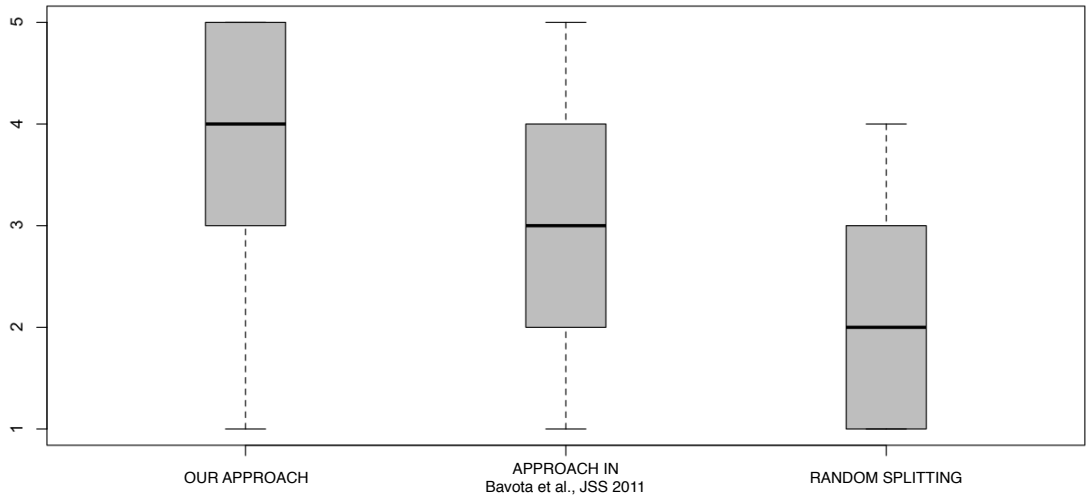

Figure 4.6: GanttProject: Box plots of the ratings provided by students

4.4.3.1 Results of the User Study (RQ_2)

Figures 4.6 and 4.7 show the boxplots summarizing the answers provided by the subjects of our experiments to the questions regarding the division of responsibilities achieved by the refactoring solutions of the different approaches[1]. In particular, Figure 4.6 reports the answers given in our first experiment (30 subjects evaluating the refactoring of GanttProject's Blobs) while Figure 4.7 reports the answers given in our second experiment (20 subjects evaluating the refactoring of Xerces's Blobs). From the analysis of Figures 4.6 and 4.7 it is easy to see that in both experiments subjects gave higher scores on the Likert scale to the refactoring proposed by our new approach. In fact, concerning the Blobs in the GanttProject system (see Figure 4.6), the median of the scores given to our approach is 4 (Agree) against 3 (Neutral) achieved by the approach presented in (1) and 2 (Disagree) of the random splitting. This difference is more evident for the evaluation of the ten Blobs of the Xerces system (see Figure 4.7). In this case the median of the scores given to our new approach is 5 (Fully Agree) against 3 (Neutral) of the approach presented in (1) and 1 (Strongly Disagree) of the random splitting. Thus, in both cases, the refactoring solutions suggested by our approach were considered as a better division of responsibilities than (i) a random splitting (as expected), and (ii) the splitting proposed by the approach proposed in (1). As also

[1]A fine grained analysis of the scores assigned by the students is reported in our online Appendix (118).

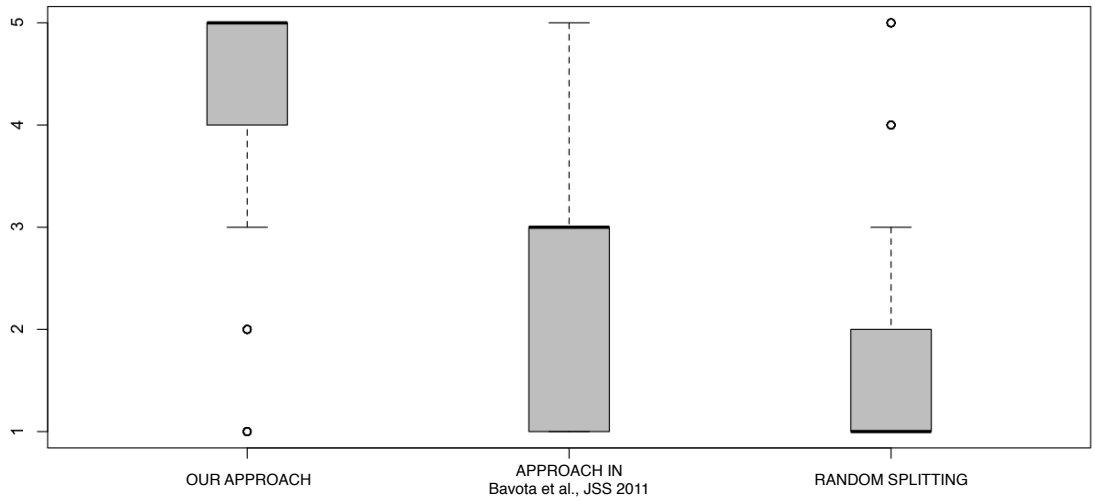

Figure 4.7: Xerces: Box plots of the ratings provided by students

Table 4.11: Results of the Mann-Whitney test.

	α	
	First experiment	Second experiment
our approach *vs* approach in (1)	< 0.01	< 0.01
our approach *vs* random splitting	< 0.01	< 0.01
approach in (1) *vs* random splitting	< 0.01	< 0.01

expected, the refactoring solutions proposed by the approach in (1) obtained in both the experiments a better evaluation than the random splitting, that was just used to understand if subjects treat the experiment seriously.

The above considerations are also supported by statistical analysis. Table 7.8 reports the results of the Mann-Whitney tests used to compare the scores given by the students to the refactoring operations achieved by the different approaches. As we can see, the solutions suggested by our approach always obtain a statistically significant higher score than the other solutions. Moreover, the refactoring solutions suggested by the approach in (1) obtains a statistically significant higher score than the random splitting in both the experiments.

The quantitative data gathered from subjects allow us to positively answer our first research question RQ_2: the division of responsibilities proposed by our approach is meaningful from a developer's point of view. However, to have deeper insights about

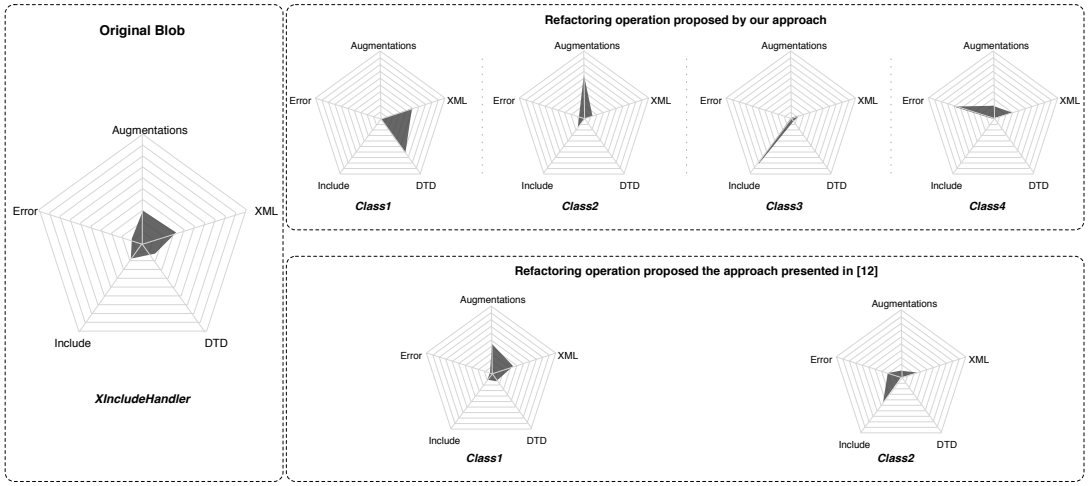

Figure 4.8: Topic Map of XIncludeHandler pre and post refactoring

the scores provided by the students we also analyzed some of the refactoring operations proposed by our approach that have been generally marked with good scores (or not) by the students.

Operations positively evaluated by students

For the Xerces system, two refactoring operations positively evaluated by almost all the students are for the *AbstractSAXParser* and *XIncludeHandler* classes. In particular, we observed that one of the classes extracted from the *AbstractSAXParser* class can be classified as an *Entity* class, since it contains only a set of attributes and the corresponding getter and setter methods. Concerning the refactoring of the *XIncludeHandler* class, it is particular interesting for two main reasons: (i) the refactoring operation suggested by our novel approach in this case achieved the higher average score, i.e., 5, (the refactoring operation suggested by the approach presented in (1) for the same Blob obtained an average score of 2.7), and (ii) this is the case with the higher difference in terms of number of extracted classes with respect to the approach proposed in (1) (4 *vs.* 2). To better analyze this case we report in Figure 4.8 the *topic map* (92) representing the main topics of (i) the original Blob, (ii) the classes extracted by our approach, and (iii) the classes extracted using the approach presented in (1). The topic map for a class C is computed analyzing the term frequency in the methods of C. In particular, we count for each term present in C (excluding the java keywords),

the number of methods that contain it. The five most frequent terms, i.e., the terms present in the highest number of methods, are then used to construct the topic map of C that, for this reason, is represented by a pentagon where each vertex represents one of the main topics. Each vertex is connected to the center of the pentagon by an axis representing the percentage of methods in the class that implements the corresponding topic. The graphical representation of the main topics of C is then obtained by tracing lines between the point on each of the five axes indicating the percentage of methods belonging to C that implement the corresponding topic. The methods in *XIncludeHandler* implement the XInclude handling of XML document according to the W3C recommendations. The XInclude functionality allows to re-use a XML document including it into other XML documents. The main topics in the class are reported in the right side of Figure 4.8. As we can see, the most frequent terms are: XML (the kind of document involved), DTD (Document Type Definition, a set of declarations used to define the document type for markup languages like XML), Include (the main responsibility of the class), Error (the management of the possible errors derived by the XInclude operation), and Augmentations (the infoset augmentation that can be used to modify a XML infoset during schema validation). Thus, even if the main responsibility of this class is the implementation of the XInclude handling, it also implements some auxiliary (and poorly related) responsibilities. The application of our approach to *XIncludeHandler* produced four new classes, each one specialized in one particular responsibility: *Class1* principally deal with the Document Type Definition, *Class2* with the infoset augmentation of XML documents, *Class3* with the implementation of the XInclude operation, and *Class4* with the management of possible errors derived by the XInclude operation. Concerning the refactoring proposed by the approach presented in (1) (bottom part of Figure 4.8), it is possible to observe that the two extracted classes still represent a mixture of different topics, even if also in this case the distribution of responsibilities is better than the original Blob.

Operations negatively evaluated by students

A particularly interesting case is represented by the refactoring of the *NonValidating-Configuration* class. In particular, our approach splits the original Blob containing 19 methods into two classes, one composed of 15 methods and the other having only three methods (see Table 4.6). The extraction of these three methods from the original class

does not achieve a good distribution of responsibilities among the new classes. However, analyzing the original class we observed that it is not easy to find a meaningful splitting from a functional point of view for this class. Even though this class has been marked as Blob in (124), probably the Extract Class refactoring is not the best way to improve the quality of this class. This is also supported by the fact that even if the refactoring proposed by our approach achieved a low average score (3.2) it was the one preferred by the students for this Blob.

With regards to the GanttProject software system, the only case of refactoring negatively rated by the students is represented by the *GanttTaskPropertiesBean* class. This is an expected results, since in our previous work (1) it was observed that for this class it is difficult to achieve a meaningful division of responsibilities using the Extract Class refactoring. Indeed, this Blob can be classified as "Data God Class" or "Lazy Class" (3) because the class holds a lot of the system's data in terms of number of attributes (67). In this case, as suggested in (3), other types of refactoring should be applied to improve the quality of the class, i.e., developers can redistribute the attributes of the Blob to other classes closer to the data.

4.4.4 Threats to validity

In this section we discuss the threats that could affect the validity of our results.

4.4.4.1 Software Metrics Evaluation

In our study we measured the increase in cohesion/coupling derived by the extract class operations suggested by our approach. To measure cohesion and coupling we employed three well-established quality metrics, i.e., $LCOM$ for the structural cohesion, $C3$ for the semantic cohesion, and the MPC for the coupling. As in all the software metrics evaluations, there is a risk that the improvement (in our case of cohesion) achieved by applying the proposed refactoring operations is obtained by construction. In fact (i) $LCOM$ is based on the instance variable shared by the methods implemented in a class, information exploited by our approach and (ii) the $C3$ metric is computed using the Conceptual Similarity between Methods exploited by our approach to capture overlap of semantic concepts between methods in the Blob class. For these reasons, even if a software metric evaluation is needed to verify that a new refactoring approach does not negatively affect the cohesion and coupling of the system, we believe that this

kind of evaluation cannot be the only type of experimentation of a new technique (as done in several previous papers (17, 18, 82, 84, 127)). So, besides achieving an increase of cohesion it is necessary to show that the suggested refactoring operations are consistent with the way developers would perform a refactoring. This is the reason why we also performed user studies. It is worth noting that the results of the metrics-based evaluation is consistent with the results of the user study reported in this section.

4.4.4.2 Subjects and Objects

In this user study 50 master students evaluated the refactoring solutions proposed by our new approach, the approach presented in (1), and a random splitting. The students did not know the goal of our experimentation or the techniques which produced the suggested refactoring solutions, to avoid bias. Moreover, the three refactoring solutions to be evaluated were presented in a random order.

The type of subjects involved in our study, i.e., master students, represents an important threat related to the generalization of the results. The students had good analysis, development, and programming experience and they can be considered as junior industrial analysts. In addition, as highlighted by Arisholm and Sjoberg (128) the difference between students and professionals is not always easy to identify. Since there are several differences between industrial and academic contexts we plan to replicate the experiment with industrial subjects to corroborate the achieved findings. Also, it is possible that subjects did not fully understand the code they judged, since they were not the original developers of the object systems. This threat is in part mitigated by our next studies where (i) we gathered more qualitative answers from students participating in a second user study and (ii) we compared the refactoring solutions proposed by our approach with those identified by original developers of six open source systems.

Another threat to the generalization of our findings is related to the limited number of real Blobs analyzed (7 in the first and 10 in the second experiment, respectively). However, this is the realistic number of tasks that we could possibly evaluate in experiments lasting approximately for two/three hours. It is not easy to perform such experiments using a substantially larger number of Blobs, unless they are conducted in multiple sessions.

Table 4.12: Analysis of the refactoring operations.

System	Class	Response		
		PhD student	Master student I	Master student II
GanttProject	GanttOptions	5	5	5
	GanttProject	5	5	5
	GanttGraphicArea	4	5	5
	GanttTree	4	5	4
	GanttTaskPropertiesBean	3	4	3
	ResourceLoadGraphicArea	4	5	5
	TaskImpl	4	5	5
Xerces	AbstractDOMParser	5	5	5
	AbstractSAXParser	5	5	5
	BaseMarkupSerializer	4	5	5
	CoreDocumentImpl	4	4	4
	DeferredDocumentImpl	1	3	2
	DOMNormalizer	5	5	5
	DOMParserImpl	5	5	5
	DurationImpl	3	4	3
	NonValidatingConfiguration	3	3	3
	XIncludeHandler	5	5	5
	Average	**4.1**	**4.6**	**4.4**

1: *Strongly disagree*; 2: *Disagree*; 3: *Neutral*; 4: *Agree*; 5: *Fully agree*

4.4.4.3 Experimental Design

An important threat is related to the claim rated by the subjects with respect to the different refactoring solutions evaluated (i.e., *The proposed refactoring results in a better division of responsibilities*). In fact, it seems unlikely that when splitting a Blob the extracted classes exhibit a worse division of responsibilities than the Blob. However, in this study we aimed at conducting a massive quantitative study to assess how much the refactoring proposed by our approach was considered as a better division of responsibilities than the selected Blobs. In addition to the refactoring solutions suggested by our approach we also provided the students with the refactoring suggested by the approach in (1) and a random one. It is worth noting that the students considered not meaningful the random refactoring (which means that splitting does not necessarily mean a better division of responsibilities), while they generally considered good (and better than the original Blob) both the refactoring suggested by the new approach

and by the approach presented in (1) and manifested a significantly higher preference for the refactoring of the novel approach. Thus, we are confident that the positive scores provided by the students to the refactoring solutions proposed by our approach represent reliable quantitative data to assess the goodnesses of the splitting operations proposed by our approach. Again, it is worth noting that the results of the user study are consistent with the metrics-based evaluation.

Since students evaluated three different refactoring solutions, another important threat is that they might have rated as meaningful the "least worst" proposed refactoring. We clearly explained to the students that for each analyzed Blob they could rate all the refactoring solutions analyzed with low scores as well as with high scores (there was not necessarily a winner to identify). However, to mitigate such a threat we asked an additional Ph.D. student and two master students[1] to evaluate for each of the 17 experimented Blobs only the refactoring solutions proposed by our approach. As in the two cases before, for each refactored class, the students had to express their level of agreement to the claim "*The proposed refactoring results in a better division of responsibilities*" proposing a score using the same Likert scale used in the experiments. Table 4.12 reports the answers provided for each analyzed class. As we can see, they agreed or strongly agreed with in most cases, with little disagreement between them. Moreover, the cases were the students negatively evaluated the proposed refactoring operations are almost the same as identified in the two experiments, e.g., *DeferredDocumentImpl*, *NonValidatingConfiguration*. Thus, we are quite confident that the results achieved in our experiments reflected well the quality of the refactoring solutions proposed by the experimented approaches.

However, even with all the mitigations described above, the threats related to this experimental design still remains, due to the purely quantitative nature of the study. The user study presented in Section 4.5 will provide a more qualitative evaluation of the proposed approach and will overcome the threats discussed here.

[1]To avoid bias in the experiment none of the authors have been involved in this evaluation.

4.5 Evaluating the Usefulness of the Refactoring Solutions

In our previous user study we performed an online experimental evaluation with software engineers of the quality of the extract class refactoring solutions suggested by our approach. In this Section, we present a second user study, performed with 15 Master students from the University of Salerno[1], aimed at gathering more data from developers about the usefulness of the refactoring solutions suggested by our approach. Also in this case subjects voluntarily took part to the experimentation and had the same academic background as the students involved in the previous user study. The study has been conducted on a set of classes extracted from open source systems that underwent extract class refactoring by the original developers. The subjects had to perform a refactoring on these classes using as initial suggestion the refactoring solutions proposed by our approach. In this way, we also have an oracle (the refactoring of the original developers) to compare the suggested solution and the refactoring performed by the students with. An important difference from the previous user study is that this time we performed the evaluation off-line, by sending all the material needed to perform the experiment via e-mail. We gave subjects two weeks to perform the required tasks. The experimental material as well as the raw data of this study are available online for replication purposes (118).

4.5.1 Research Questions and Planning

In the context of this study, the following research question has been formulated:

- RQ_3: Are the refactoring solutions suggested by our approach useful for developers when performing extract class refactoring?

To obtain the objects needed by our study we mined six open source systems (i.e., Apache HSQLDB, ArgoUML, JEdit, JFreeChart, JHotDraw, Xerces) looking for extract class refactoring operations performed during their history by the original developers. We used Ref-Finder (117) to identify the refactoring operations performed among two subsequent versions of the same system. Ref-Finder is a tool able to identify

[1]None of the 50 students involved in the user study reported in Section 6.4 has been involved in this experiment.

Table 4.13: Extract Class Refactoring Operations Identified in the Six Analyzed Systems

System	Original Class	Extracted Classes
Apache HSQLDB	Database (40)	Database (27) SchemaManager (13)
	Select (14)	Select (7) Result (7)
	UserManager (14)	UserManager (9) GranteeManager (5)
ArgoUML	FileGeneratorAdapter (11)	FileGeneratorAdapter (3) TempFileUtils (8)
	Import (10)	Import (7) ImportCommon (3)
JEdit	JEditTextArea (214)	JEditTextArea (22) SelectionManager (11) TextArea (181)
JFreeChart	JFreeChart (24)	JFreeChart (16) Plot (8)
	NumberAxis (20)	NumberAxis (16) ValueAxis (4)
JHotDraw	DefaultApplicationModel (14)	DefaultApplicationModel (4) AbstractApplicationModel (10)
Xerces	XMLDTDValidator (69)	XMLDTDValidator (38) XMLDTDProcessor (31)
	XMLSerializer (25)	XMLSerializer (12) DOMWriterImpl (13)
In parenthesis the number of methods in each class		

63 different types of refactoring, but unfortunately not the extract class one. However, the latter can be identified by Ref-Finder as a set of *move method* and *move field* operations from the original class to the new extracted classes. We manually validated these sets of move method and move field refactoring retrieved by Ref-Finder to identify extract class refactoring operations performed by the original developers. In total, we identified eleven meaningful extract class refactoring operations performed by the original developers as shown in Table 4.13.

To answer our research question we provided each subject the eleven classes to refactor together with the refactoring solution proposed by our approach. Then, since for each of the eleven identified classes we have the original class as well as the new classes extracted by the developers, we can answer RQ_3 from both a qualitative and a

quantitative point of view.

As for the qualitative analysis, we asked the subjects the following questions:

1. Would you split this class?
   (a) if YES:
      i. Why?
      ii. Would you split the class differently than the provided refactoring solution? Why?
      iii. Did you find the provided refactoring solution a good starting point to perform your refactoring?Why?
   (b) if NO:
      i. Why?

As for the quantitative analysis we measured how much the refactoring produced by the students (i) was different than the solution suggested by our approach and (ii) approximated the refactoring performed by the original developers. We used the MoJoFM (120) to measure the similarity between the refactoring performed by the students, the ones proposed by our approach, and those performed by the original developers. Moreover, we also measured (still through the MoJoFM) how far is the refactoring suggestion proposed by our our approach from the refactoring performed by the original developers on each of the eleven classes object of our study. Given the low number of observations (i.e., eleven) we did not execute any kind of statistical test.

4.5.2 Analysis of the Results

Table 4.14 shows, for each of the eleven classes object of our study, the percentage of subjects answering "YES" to the three YES/NO questions of our survey. For example, 13 out of the 15 students involved (87%) would split the class *Database* and 8 of them (62%) would split the class differently than the solution suggested by our approach. However, all these 13 subjects founded the provided refactoring solution (i.e., the one proposed by our approach) a good starting point to perform the refactoring.

The analysis of Table 4.14 reveals interesting results. First of all, not always subjects would split the provided classes. In particular, there are two of the analyzed eleven classes (i.e., *Select* and *Import*) for which the majority of the students did not feel that

Table 4.14: Answers provided by the subjects

Class	% Students answering YES		
	Would you split this class?	Would you split the class differently?	Was the provided refactoring suggestion useful?
Database	87%	62%	100%
Select	27%	0%	100%
UserManager	100%	87%	67%
FileGeneratorAdapter	67%	40%	100%
Import	40%	0%	100%
JEditTextArea	100%	100%	100%
JFreeChart	100%	0%	100%
NumberAxis	87%	62%	100%
DefaultApplicationModel	80%	8%	100%
XMLDTDValidator	100%	0%	100%
XMLSerializer	87%	67%	100%

extract class refactoring was needed. As explained in the design, we also asked subjects *why* they would/would not split each class. Analyzing these answers we found that subjects judged the complexity of both classes acceptable and were not able to identify different responsibilities implemented in them. Clearly, this result contrasts with the choice made by the original developers. However, analyzing the refactoring performed by the original developers on the classes *Select* and *Import* it is clear that their choice was not driven by the high complexity of those classes but to the willingness to improve their design. In fact, (i) the number of methods implemented in both classes is quite low (i.e., 14 in *Select* and 10 in *Import*) and (ii) in both cases the developers performed the extract class refactoring to split the classes into a Model class, responsible of modeling an entity in the system, and a Controller class working on the Model. For the *Import* class our approach proposes exactly the refactoring performed by the original developers and the six students that would split this class accepted the refactoring suggested as is (and clearly, found the suggestion useful). Moreover, three of them were also able to motivate their choice by explaining that "*the class Import seems a merge between a Model and a Controller*", "*it is possible to extract a model class*", and "*to improve its reusability a model class can be extracted*". As for the *Select* class, also in this case the four subjects that would split it accepted the suggestion of our approach as is, explaining its usefulness with the fact that *"the extracted classes looks strongly cohesive"*.

On the other side there is a set of four classes that all subjects would like to split (i.e., *UserManager*, *JEditTextArea*, *JFreeChart*, and *XMLDTDValidator*). As for the *UserManager* class, the subjects explained that this class performs "*more than just managing the users*" and they can identify "*two different responsibilities implemented in it*". Ten subjects (67%) found the suggestion of our tool useful explaining that "*it eases code comprehension*" by "*highlighting the main responsibilities implemented in the class*". Our approach splits each of these classes in two new classes. It is interesting to note that 87% of the subjects (13 out of 15) modified the suggested refactoring solution and all of them moved just one method from one of the extracted classes to the other obtaining exactly the refactoring performed by the original developers. Concerning the 33% of subjects that did not find useful the suggestion of our approach for this class, most of them motivated this answer by explaining that "*the class was not really complex*" and thus "*its main responsibilities can be identified without any suggestion*". However, none of them complained about the quality of the proposed refactoring.

Interesting is also the case of the *JEditTextArea* class for which all subjects (i) think that a refactoring would be necessary, (ii) would like to change the proposed refactoring, and (iii) think that the suggested refactoring was useful as starting point. As for the need to split this class most of the subjects explain it by highlighting that "*the class is very complex*", "*intricate*", and "*looks low cohesive*". Our technique extracts from *JEditTextArea* three new classes. All subjects suggested that two of these classes can be merged together and, four of them also extracted a new class managing "*the scrolling of a text area*". However, all subjects found the starting refactoring suggestion useful commenting that "*the proposed division of responsibilities makes sense, but perhaps it is a bit excessive*". This motivation explains the fact that all of them merged together two of the thee extracted classes.

For other classes like *JFreeChart*, and *XMLDTDValidator* all students accepted the refactoring suggestion as is, commenting in most cases that "*the extracted responsibilities were meaningful*" and "*cohesive classes were extracted*".

Finally, another interesting case is concerned with the *NumberAxis* class. Most of the students (87%) would like to split this class since "*the management of the axis values should be extracted*". Our approach suggested to split this class in three new classes. While 38% of students appreciated this suggestion, the other 62% applied a change to it by merging two of the three suggested classes. The students who applied

Table 4.15: MoJoFM between (i) the refactoring suggested by our approach and that performed by the original developers (ii) the refactoring performed by subjects and the refactoring proposed by our approach, and (iii) the refactoring performed by subjects and that performed by the original developers

Class	Our approach to Original Dev.		Subjects to our approach		Subjects to Original Dev.	
	MoJoFM	#Move/Join	Avg. MoJoFM	Avg. #Move/Join	Avg. MoJoFM	Avg. #Move/Join
Database (40)	0.97	1	0.98	0.6	0.99	0.5
Select (14)	0.83	2	1.0	0	0.83	2
UserManager (14)	0.93	1	0.94	0.9	0.98	0.3
FileGeneratorAdapter (11)	0.86	1	0.92	0.6	0.94	0.4
Import (10)	1.00	0	1.00	0	1.00	0
JEditTextArea (214)	0.84	34	0.97	6	0.87	27
JFreeChart (24)	0.95	1	1.00	0	0.95	1
NumberAxis (20)	0.94	1	0.96	0.6	0.98	0.4
DefaultApplicationModel (14)	0.92	1	0.99	0.2	0.92	1
XMLDTDValidator (69)	0.88	8	1.00	0	0.88	8
XMLSerializer (25)	0.91	2	0.97	0.7	0.94	1.4
Average	**0.91**	**4.7**	**0.98**	**0.9**	**0.93**	**3.8**
In parenthesis the number of methods of the class						

the change to the refactoring proposed our approach (just a join operation) were able to replicate the refactoring performed by the original developers. Overall, all subjects appreciated the refactoring suggestion highlighting as it "*eases the comprehension of the main responsibilities implemented in a class*".

Summarizing, except for the case of the *UserManager* class discussed above, subjects always found useful the solutions suggested by our approach when performing refactoring. Among the most frequent explanations we found:

1. it eases code comprehension;
2. it highlights the main responsibilities implemented in a class;
3. the extracted classes are cohesive.

Moreover, subjects stated that in some cases "*without the refactoring suggestion it would be too difficult to identify the main responsibilities of the classes*".

Concerning the quantitative data, Table 4.15 reports (i) the MoJoFM between the refactoring solution suggested by our approach and that performed by the original developers, (ii) the average MoJoFM between the refactoring performed by the subjects and the refactoring solution suggested by our approach and (iii) the average MoJoFM

between the refactoring performed by the subjects (starting from the suggestions of our approach) and the refactoring performed by the original developers. Moreover, Table 4.15 also reports the number of Move/Join operations needed to convert one refactoring into the other.

The first thing that leaps to the eyes is that our approach is able to well approximating the refactoring performed by the original developers, achieving on average 0.91 of MoJoFM. For example, for the *Database* class from the Apache HSQLDB system our approach achieve 0.97 of MoJoFM. This class was split in 2 new classes by the original developers (see Table 4.6), one containing 27 and one containing 13 methods. Our approach splits the *Database* class into three classes. The first is composed of the same 27 methods included in one of the two classes extracted by the developers. The other two extracted classes contains the remaining 13 methods, 8 in one class and 5 in another one. Thus, just performing one *Join* operation (i.e., merging the two smallest classes extracted) it is possible to obtain the refactoring performed by the developers.

As for the average number of Move/Join operations required to convert the refactoring suggested by our approach into the refactoring performed by the original developers, on average 4.7 operations are required. The only case in which a quite high number of Move/Join operations is required to convert the refactoring solution proposed by our approach to the one performed by the developers is related to the class *JEditTextArea*. In this case, 34 Move/Join operations are required. However, it is worth noting that the original class was composed by 214 methods. Thus, in this case 34 Move/Join operations required to convert the refactoring proposed by our approach into the one performed by the developers represent a good results, as also demonstrated by the high MoJoFM achieved (0.84). Note that, excluding the case of the *JEditTextArea* class, the average number of required Move/Join operations required to convert the refactoring solution proposed by our approach into the refactoring performed by the original developers is only 1.8.

The second important result of our study is that the refactoring suggested by our approach is only slightly modified by the 15 subjects. In fact, the average MoJoFM is 0.98 and the average number of required Move/Join operations to convert the suggestion by our approach in the refactoring performed by subjects is less than one (0.9). Moreover, starting from the suggestions by our approach, subjects were able to further approximate the refactoring performed by the original developers achieving an average

MoJoFM of 0.93. On average, only 3.8 Move/Join operations are needed to convert their refactoring in the refactoring performed by the original developers.

This result highlights how the refactoring solutions suggested by our approach objectively represent a very good starting point for developers interested in performing extract class refactoring operations. In fact, students having zero knowledge of the object systems were able to comprehend the classes and perform refactoring operations very close to those performed by the original developers having a deep knowledge of these open source systems.

4.5.3 Threats to Validity

Here we discuss the main threats that could affect the validity of our results.

4.5.3.1 Subjects and Design of the User Study

As in our previous user study (see Section 6.4) also in this case our subjects were Master students. This clearly results in the same threats related to their knowledge of the source code under analysis. However, using the refactoring solutions suggested by our approach, they were able to produce refactoring very close to those performed by the original developers. Thus, we are confident that this threat did not affect the validity of our study.

Another threat is related to the fact that we did not perform the experimentation in a controlled setting, as we invited subjects to participate via e-mail. However, unlike the user study presented in Section 6.4, which simply required to score refactoring solutions, performing this type of study in a controlled setting is quite unfeasible since the time needed to refactor 11 classes and provide all the required qualitative feedback is much higher. Indeed, subjects invested between 6 and 10 hours to perform the required tasks[1].

4.5.3.2 Reliability of the Considered Oracle

In this study we considered the refactoring performed by the original developers as the golden standard to which test the refactoring proposed by our approach. This is, to the best of our knowledge, the first refactoring approach evaluated against real refactoring

[1]These data were provided by subjects when sending their results to us.

Table 4.16: Our approach *vs* the approach in (1): MoJoFM achieved in reconstructing the refactoring performed by the original developers

System	Original Class	Our approach		Approach in (1)	
		MoJoFM	#Move/Join	MoJoFM	#Move/Join
Apache HSQLDB	Database (40)	0.97	1	0.66	13
Apache HSQLDB	Select (14)	0.83	2	0.42	7
Apache HSQLDB	UserManager (14)	0.93	1	0.58	5
ArgoUML	FileGeneratorAdapter (11)	0.86	1	0.67	3
ArgoUML	Import (10)	1.00	0	0.63	3
JEdit	JEditTextArea (214)	0.84	34	0.74	51
JFreeChart	JFreeChart (24)	0.95	1	0.64	8
JFreeChart	NumberAxis (20)	0.94	1	0.78	4
JHotDraw	DefaultApplicationModel (14)	0.92	1	0.67	4
Xerces	XMLDTDValidator (69)	0.88	8	0.54	31
Xerces	XMLSerializer (25)	0.91	2	0.49	12
Average	-	0.91	4.7	0.62	12.8
In parenthesis the number of methods of the class					

operations performed by original developers of open source systems. However, we do not know (i) the experience of the developer who actually performed the refactoring and (ii) how the developer performed the refactoring (e.g., totally manually, using a tool suggestion, etc.).

Concerning the first point, the open source community working on the considered projects accepted the performed refactoring operations without modifying it in future. This makes us at least confident that the performed refactoring operations were correct and meaningful.

Concerning the second point we analyzed the commit messages wrote by the developers when uploading the changes resulting from the refactoring operation in the software repository. We did not find any claim about the usage of tools to perform the refactoring, although we cannot be 100% sure that this was not the case.

4.5.3.3 On the Performance of Our Approach when approximating a Manually performed Refactoring

In the performed study our approach was able to approximate the refactoring performed by the original developers with a very high accuracy (i.e., 0.91 of MoJoFM). To have a benchmark, we compared the performance of our approach with that achieved by the approach in (1). Table 4.16 reports the MoJoFM between the refactoring performed by

the original developers and the refactoring suggested (i) by our approach and (ii) by the approach in (1). Moreover, Table 4.16 also reports for both approaches the number of Move/Join operations needed to convert the refactoring they suggest in the refactoring performed by the original developers. As we can see the gap of performances between the two approaches is very sharp, 0.91 for our approach against the 0.62 achieved by the approach in (1). An interesting case is represented by the *JEditTextArea* class, since it is the only one split by the original developers into three new classes. In this case our approach achieves 0.84 of MojoFM, against the 0.74 achieved by the approach in (1). However, we also iteratively applied the approach in (1) as explained in Section 4.3.3. The MoJoFM even decreases to 0.71, thus again demonstrating the unsuitability of iteratively applying the approach proposed in (1).

4.5.3.4 On the low Number of Refactoring Operations Analyzed

To identify refactoring operations performed by the original developers we mined the history of six software systems. However, we only identified 11 extract class refactoring operations. This result is quite surprising but can be explained by analyzing the process we used to identify the extract class operations. First the Ref-Finder tool we used to identify refactoring operations does not identify extract class refactorings, but only sets of move method and move field operations. We manually validated these sets to recognize them as extract class refactoring operations, so it is possible that we miss the identification of some extract class refactoring operations. Second and most important, we found several operations which include a mixture of different refactoring operations. For example, often some methods of a Blob class are deleted, others are moved to an existing class (i.e., move method) and the remaining are split in new classes (i.e., extract class). We decided to ignore these cases since they do not represent pure extract class operations and testing an extract class refactoring approach in these cases would be unreliable.

4.6 Final Remarks

This Chapter describes an approach to automate Extract Class refactoring. Given a class to be refactored, the approach computes a measure of similarity between all possible pairs of methods in the class. Such a measure captures relationships between

methods that impact class cohesion (e.g., attribute references, method calls, and semantic content) and gives the likelihood that two methods should be members of the same class. The approach identifies chains of strongly related methods, i.e., highly coupled methods. The set of extracted method chains is exploited to build new classes—one for each chain—having higher cohesion than the original class.

An empirical assessment and evaluation of the proposed approach was performed through three studies. The goal of the first study was to identify a heuristic for the definition of the parameters of the proposed approach. Then, it has been experimented on real Blobs of open source systems to evaluate (i) how good the proposed solution is from a cohesion and coupling metrics point of view, (ii) how good the proposed refactoring solution is considered by software engineers as it is. Finally, in a third study we evaluated on eleven classes (i) how much the proposed solution is useful as starting point to perform a refactoring, and (ii) how well the proposed refactoring solution approximate a refactoring made by the original developers. The results show that the refactoring solutions proposed by our approach strongly increases the cohesion of the refactored classes without leading to significant increases in terms of coupling. Moreover, the refactoring solutions proposed by our approach are considered useful to developers performing extract class refactoring and are able to approximate a manually performed refactoring at 91% on average.

5

Extract Package Refactoring

The material in this Chapter has been presented in (129, 130).

5.1 Introduction

In this Chapter we instantiate the approach aimed at decomposing complex object presented in Section 3.4 to the Extract Package refactoring. As explained in Chapter 2, the aim of this refactoring is to decompose a package with poor cohesion (i.e., a promiscuous package) into smaller and meaningful packages having higher cohesion. The approach exploits two class-level coupling metrics (i.e., Information-Flow-based Coupling (ICP) (108) and Conceptual Coupling Between Classes (CCBC) (55)) to capture structural and semantic relationships between classes, respectively. The use of the ICP measure allows the technique to capture the amount of information flowing between the classes of the system via parameters through method invocations. In other words the ICP measure provides information about the structural dependencies between classes. On the other side, the CCBC measure captures the lexical information embedded in the comments and identifiers of the classes, allowing to identify semantically (i.e., domain semantics) related classes, i.e., classes containing similar terms in their comments and identifiers. These two sources of information are combined and used to determine classes that should belong together in a package. The technique is automated and suggests to developers how to split existing packages, when needed.

The evaluation of the approach has been performed on five systems. In particular, we merged several packages of the object systems and used the proposed approach

Table 5.1: Systems used in the case study.

System	Version	KLOC	#Classes	#Packages	Cohesion		
					Mean	Median	St. Dev
eTour	1.0	30	134	17	0.348	0.311	0.067
GESA	2.0	46	297	22	0.332	0.289	0.120
JHotDraw	6.0 b1	29	275	12	0.364	0.379	0.052
SESA	1.2	11	128	14	0.318	0.292	0.073
SMOS	1.0	23	121	12	0.400	0.424	0.039

to split the merged package aiming at reconstructing the original packages. Our assumption is that the higher the reconstruction accuracy of our approach, the higher the meaningfulness of the proposed re-modularization. This assumption is supported in part by our choice of systems, which have high quality. However, to further verify this assumption in the cases where the re-modularization proposed by our approach is significantly different from the original decomposition of the system, we asked some of the original developers of the object systems to analyze the proposed package decompositions and evaluate them from a functional point of view.

Section 5.2 presents the proposed approach, while Section 5.3 presents the design and the results of the empirical study.

5.2 The Approach

The proposed approach takes as input a package identified by the software engineer as a candidate for re-modularization. Then, a measure reflecting a relationship between pairs of classes from the package is computed. The measured values between classes are stored in a $n \times n$ matrix, called *class-by-class* matrix, where n is the number of classes in the package under analysis. A generic entry $m_{i,j}$ in the class-by-class matrix represents the likelihood that class c_i and class c_j should be in the same package.

Using the information in the class-by-class matrix the approach extracts chains of strongly related classes. The classes of the original package are distributed in different packages according to the extracted chains. If the number of extracted chains is one, no re-modularization is suggested by the tool (this generally happens when the cohesion of the analyzed package is high). Otherwise, based on the extracted class chains the

approach suggests new packages with higher cohesion than the original package. Note that the structure of individual classes in the package is not changed.

While the proposed approach is automated, it is actually supposed to serve as an assistant to the developer. Design decisions are often more complex and subtle than just trying to maximize package cohesion. In consequence, the extracted packages are analyzed by the software engineer who can accept the proposed re-modularization as is, or change it by moving classes from one package to another.

5.2.1 Class-by-Class Matrix Construction

The likelihood that class c_i and class c_j should be in the same package is estimated by capturing different types of relationships between classes that can affect package cohesion. In particular, we define the likelihood that two classes should be in the same package by combining two different (structural and semantic) measures, i.e., information-flow-based coupling (ICP) (108) and Conceptual Coupling Between Classes (CCBC) (55). The definition of these metrics can be found in Sections 3.2 (ICP) and 3.3 (CCBC).

The choice of metrics to use is not random, as it is based on previous research (55) that analyzed the combination of structural and semantic coupling metrics to predict changes in OO software. ICP and CCBC fared better than other structural and conceptual metrics, respectively. Moreover, the empirical analysis conducted in (55) have shown that structural and semantic coupling measures do not correlate, which indicates that they capture different aspects of coupling. In light of these results, we expect that (i) a package composition based on these metrics will group together classes that tend to change together, which is a desirable property in order to localize change, and (ii) the use of a combination of orthogonal quality metrics to guide the re-modularization activity can provide better results than any one of its constituents (89, 131).

The likelihood that classes c_i and c_j should be in the same package as:

$$coupling_{i,j} = w_{ICP} \cdot \widetilde{ICP}(c_i, c_j) + w_{CCBC} \cdot CCBC(c_i, c_j)$$

where $w_{ICP} + w_{CCBC} = 1$ and their values express the confidence (i.e., weight) in each measure. The weights assigned to these measures are empirically defined and the methodology for this step is presented in Section 5.3.

5.2.2 Class Chains Extraction

Once computer the relationships between the classes of the package to be refactored, the extraction of the class chains is performed in two steps. Firstly, the class-by-class matrix is filtered in order to remove spurious structural and/or semantic relationships between classes using a threshold, *minCoupling*. Also in this case, we experiment in our evaluation (Section 5.3) both constant and variable thresholds (see Section 4.2.2) as done for the Extract Class refactoring approach.

Once the class-by-class matrix is filtered, its transitive closure is computed to identify the chains of strongly related classes (i.e., packages) that should be extracted from the promiscuous package. Clearly, also for this approach it is possible that the set of computed chains (i.e., suggested packages) include chains with a very short length due to classes having poor relationships with other classes. To avoid suggesting very small packages (i.e., packages with a very low number of classes), we use a chain length threshold, *minLength*, to identify trivial chains, i.e., chains with a length less than *minLength*. Similar to (132), in our approach we set $minLength = 4$ since a good re-modularization approach should avoid the creation of packages with too few classes. This minimum length can be easily changed if needed. Then, we compute the coupling between trivial and non-trivial chains and merge each trivial chain with the strongest coupled non-trivial chain. The coupling between chains is calculated using the same measures used to calculate the coupling between classes. Specifically, the coupling between chains Ch_i and Ch_j is computed as the average coupling between all the possible pairs of classes from Ch_i and Ch_j.

5.3 Empirical Evaluation

In this section we report the design and the results of the study conducted to evaluate the proposed Extract Package approach.

5.3.1 Planning

The study follows the Goal-Question-Metrics paradigm (133).

5.3.1.1 Definition and Context

The goal of the empirical study is (i) to assess the parameters of the proposed approach, i.e., the weights of the coupling metrics (w_{ICP}, and w_{CCBC}) and the *class-by-class matrix* filtering threshold (*minCoupling*), and (ii) to determine whether the proposed approach generates meaningful re-modularization of packages in object-oriented software systems.

The objects of our study are: an open source system, JHotDraw[1] and four software systems (eTour, GESA, SESA, and SMOS) developed by university students during a Software Engineering course. Among these, GESA has been developed in an industrial traineeship and it is operational at University of Molise since 2009. JHotDraw is a Java GUI framework for structured drawing editors. eTour is an electronic touristic guide, while GESA is a web-based application used in the management of university courses. SESA is also a web-based application used to manage relevant information of the Software Engineering Lab of the University of Salerno, e.g., people, projects, publications. Finally, SMOS is a software developed for high schools, which offers a set of functionalities aimed at simplifying the communications between the school and the students' parents. Table 7.1 reports the statistics (i.e., KLOC, number of classes, and number of packages) as well as the versions of the systems used in the study. The table also reports the descriptive statistics of the packages' cohesion from the systems, measured using a cohesion metric, namely *CohesionQ*, defined in (82). Our evaluation strategy requires that the object systems have high package cohesion. The measures support our choice, as the average cohesion values for the five systems are higher than that of the systems analyzed in (82): JEdit (with average cohesion 0.288), ArgoUML (0.172), Jboss (0.125), and Azureus (0.117).

5.3.1.2 Research Questions and Planning

By construction, the approach will extract from a package, class chains having higher cohesion than the original package. However, as we mentioned before a good re-modularization cannot be based only on the higher cohesion of the new packages. An evaluation involving developers is required in order to assess the overall quality and meaningfulness of the proposed re-modularization. For this reason, our study aims at

[1] http://www.jhotdraw.org

(i) assessing the parameters of our approach and (ii) analyzing if the proposed approach is able to identify meaningful re-modularization operations from a developer point of view. Thus, two research questions are formulated:

- **RQ**$_1$: How do the parameters of the proposed approach affect the results?
- **RQ**$_2$: Is the proposed approach able to find meaningful re-modularizations?

To respond to our research questions we mutated the original version of the object systems using a tool that randomly selects $m \geq 2$ packages of the system and merges them in a single package $\widehat{P}_m$. The merging operation was recorded in a log file to allow us to know the packages merged by the tool. At the end of the merging operation we obtained a mutated system with a worse package decomposition compared to the original system. The proposed approach is then applied to the $\widehat{P}_m$ package in order to reconstruct (or improve) the original packages. At the end of the re-modularization operation we obtained a new version of the mutated system. Specifically, given the merged package $\widehat{P}_m$, the proposed approach is expected to generate m packages. To evaluate the proposed approach, the new packages were compared with the originally merged packages aiming at identifying the total number of classes correctly and incorrectly placed in the new packages. The ideal behavior is that the split packages are the same (i.e., contain the same classes) as the original packages. In essence, we consider them as a "golden standard". This choice is supported by the fact that the systems used in the study have a generally good package quality that is reflected in terms of package cohesion (see Table 7.1). In particular, one of the object systems, JHotDraw, has been developed as a "design exercise" and its design relies heavily on using well-known design patterns. The other four systems were chosen among the best projects developed during the software engineering course. As shown in Table 7.1 the cohesion of the packages of the other four systems is close to that of JHotDraw. With that in mind, recovering the original packages likely means that the approach is able to identify meaningful re-modularizations.

In order to respond to our first research question, we identified different re-modularization solutions on the same merged packages, i.e., mutated systems, using different settings of **weights** for the selected metrics, i.e., w_{ICP}, and w_{CCBC}, and different values for the **threshold** used to remove spurious relationships from the class-by-class matrix, i.e., the parameter *minCoupling*. In particular, for each metric weight we varied this

parameter starting at 0 and increasing it until 1 by a step of 0.1. We exercised all the possible combinations of such values assuring that $w_{ICP}+w_{CCBC} = 1$, i.e., 11 different combinations. Concerning the parameter *minCohesion* we experimented both constant and variable thresholds. In particular, we used five different constant thresholds and three different variable thresholds. Concerning the constant thresholds we used 0.1, 0.2, 0.3, 0.4, and 0.5, while as variable thresholds we considered the first (Q_1), the second (Q_2), and the third (Q_3) quartile, respectively, of the non-zero values in the class-by-class matrix. Note that the use of quartiles allows to define a threshold that is less affected—as compared to the other descriptive statistics (e.g., mean)—by problems caused by skewed distributions of the values in the class-by-class matrix. We selected different values for the number of packages to be merged, i.e., $m \in \{2, 3, 5\}$, aiming at obtaining merged packages with a low cohesion and varied set of responsibilities. For each value of m we performed 10 different trials, i.e., $n = 10$, randomly selecting each time different combinations of the merged packages. In total, we did 30 merging and re-modularizations operations for each system (varying on the 11 combinations of weights, i.e., w_{ICP} and w_{CCBS}, and on 8 different values for *minCoupling*). Thus, the total number of trials performed on each object system is $11 \times 8 \times 30 = 2,640$, for a total of 13,200 re-modularizations for the five systems.

To evaluate the results produced by the configurations experimented for our approach we used the two Information Retrieval metrics recall and precision (106). In our study recall measures the percentage of classes correctly placed in the split packages, while precision measures the percentage of classes that are correctly placed. Since the two metrics measure two different concepts, we decided to use the **F-measure** (106) as the dependent variable to assess the performances of the proposed approach and to guide the selection of the best values for the weights and parameters described above. Note that the F-measure is computed analyzing only the reconstruction of the merged packages and not the entire system decomposition.

The reconstruction accuracy (F-measure) achieved using the best parameters setting is also used to respond to our second research question. Our assumption is that the higher the reconstruction accuracy of our approach, the higher the meaningfulness of the proposed re-modularization. As explained before, this assumption is supported in part by our choice of systems which have a high quality in terms of package cohesion. However, it is possible that even in software systems having a good remodularization

Table 5.2: Subjects involved in the functional evaluation.

System	#Subjects	Original Developers?
eTour	2	Yes
GESA	5	Yes
JHotDraw	2	No
SESA	2	Yes
SMOS	5	Yes

quality, some classes are misplaced in some packages. Thus, in our second research question, we analyzed the proposed re-modularization operations from a functional point of view. In particular, in the cases where the re-modularization proposed by our approach is considerably different from the original decomposition of the system, we asked some of the original developers of eTour, GESA, SESA, and SMOS to analyze the proposed package decomposition and evaluate the performed re-modularizations. For JHotDraw, the same evaluation was made by two graduate students who are very familiar with the system. We involved a total of 16 subjects in the functional evaluation distributed among the object systems as reported in Table 5.2. To identify the cases to analyze, we set an F-measure threshold ϵ. All the cases for which our approach was not able to reconstruct the original packages with an F-measure higher than ϵ were analyzed by the students. For each of the selected cases the students responded to the following question:

Is the proposed package decomposition meaningful?

with a score using a 5-point Likert scale (126): 1: *Strongly agree*; 2: *Weakly agree*; 3: *Neutral*; 4: *Weakly disagree*; 5: *Strongly disagree*.

5.3.2 Analysis of the Results

In this section we present the results of the empirical study. We discuss two aspects of the results. First, we use the results to determine the best values for the weights and parameters. Second, we analyze the results of the students' evaluation of the proposed re-modularizations.

Table 5.3: Descriptive statistics of results achieved reconstructing merged packages.

System	m	Best Configuration			F-Measure (After step 1)			F-Measure (After step 2)		
		w_{CCBC}	w_{ICP}	Threshold	Mean	Median	Std.dev.	Mean	Median	Std.dev.
eTour	2	0.9	0.1	Q_3	0.804	0.852	0.094	0.891	0.936	0.086
	3	0.9	0.1	Q_3	0.688	0.703	0.102	0.760	0.774	0.082
	5	0.9	0.1	Q_3	0.559	0.562	0.084	0.668	0.659	0.057
GESA	2	0.9	0.1	Q_3	0.917	0.936	0.039	0.967	0.981	0.044
	3	0.9	0.1	Q_3	0.763	0.795	0.114	0.822	0.897	0.131
	5	0.9	0.1	Q_3	0.603	0.578	0.104	0.720	0.706	0.073
JHotDraw	2	0.7	0.3	Q_3	0.749	0.788	0.142	0.785	0.807	0.180
	3	0.9	0.1	Q_3	0.672	0.709	0.100	0.724	0.760	0.085
	5	0.8	0.2	Q_3	0.593	0.635	0.056	0.688	0.675	0.027
SESA	2	0.8	0.2	Q_3	0.897	1.000	0.095	0.930	1.000	0.108
	3	0.8	0.2	Q_3	0.647	0.602	0.125	0.700	0.643	0.109
	5	0.8	0.2	Q_3	0.548	0.522	0.138	0.660	0.600	0.104
SMOS	2	0.8	0.2	Q_3	0.769	0.846	0.206	0.804	0.920	0.263
	3	0.8	0.2	Q_3	0.705	0.720	0.108	0.770	0.790	0.101
	5	0.9	0.1	Q_3	0.558	0.604	0.126	0.668	0.700	0.138

The number of merged packages is m.

Table 5.3 reports the results produced—in terms of **F-measure**—by the *best* configuration of parameters identified on each of the object systems[1]. The results in Table 5.3 highlight:

- *the benefits of the second step of our approach.* After merging each trivial chain (i.e., a chain composed of less than 4 classes), with the most similar non-trivial chain (see Section 5.2.2), we obtained an average increment of the F-measure by about 7% (with respect to the previous step of the approach);
- *the decrease of the reconstruction accuracy when increasing the number of merged packages.* Indeed, the average F-measure decreases from 88% when merging 2 packages, to 75% when merging 3 packages, until 68% when merging 5 packages;
- *a general rule for setting the parameters of our approach.* The results reveals that the best performances can be obtained using as threshold the third quartile, i.e.,

[1]The complete results achieved with all the possible combination of parameters can be found in (134).

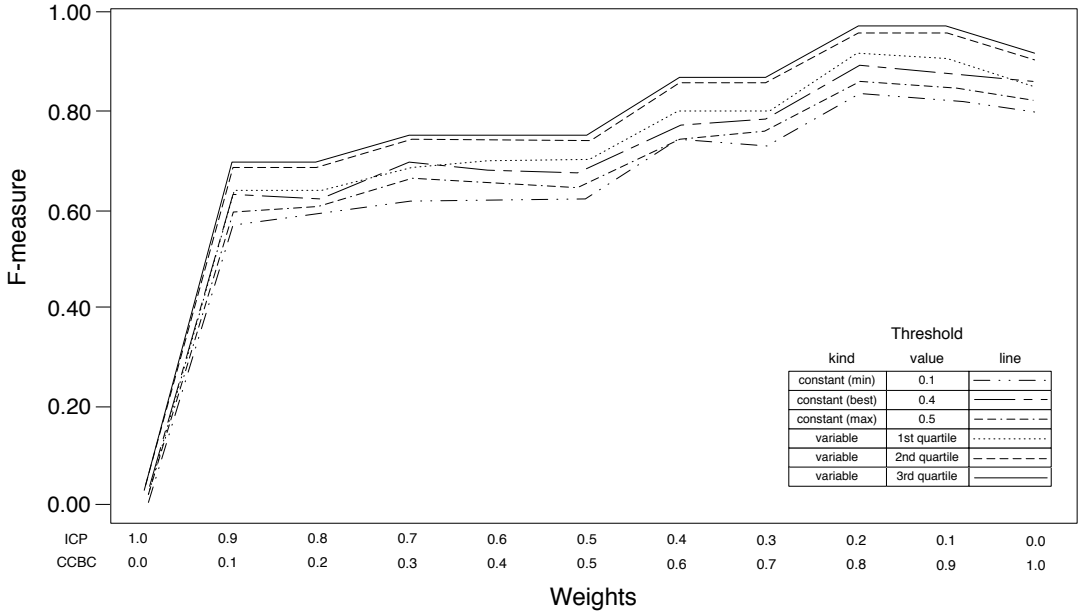

Figure 5.1: Interaction between Weight and Threshold on GESA merging 2 packages.

Q_3, of the non-zero values of the class-by-class matrix and setting $w_{CCBC} \geq 0.7$. However, the best configuration of weights is slightly different among the object systems. Thus we need to investigate deeper the influence of the configuration parameters on the performances of the proposed approach.

5.3.2.1 Influence of the parameters

To better analyze the influence of the configuration parameters Figs. 5.1, 5.2, and 5.3 show the interaction plots between **Weights** and **Threshold** on GESA, merging 2, 3, and 5 packages, respectively. We report the results obtained using all the three variable thresholds but for sake of readability we only report the results achieved using the lower, the higher, and the best constant threshold (see (134) for the complete interaction plots of all the systems).

The analysis indicates that the variable thresholds ensure better filtering performances than the constant thresholds across the different inputs, i.e., the different artificial packages to be re-modularized. In particular, the best performances are achieved using Q_3 as threshold to remove spurious relationships in the class-by-class matrix.

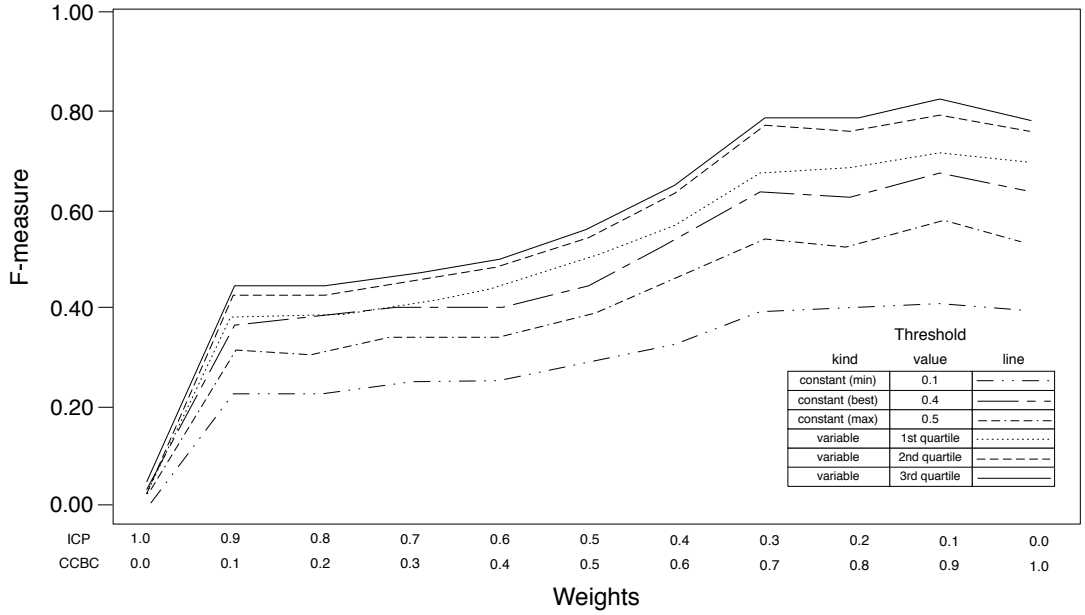

Figure 5.2: Interaction between Weight and Threshold on GESA merging 3 packages.

Regarding the weights, the results reveal that the weight for the semantic metric, i.e., w_{CCBC}, should be higher than 0.6. In fact, all combination of weights having $w_{CCBC} \geq 0.7$ has a reconstruction accuracy almost equal to the best (see Figs. 5.1, 5.2, and 5.3). This trend (as well as the trend concerning the variable threshold Q_3) is confirmed across all the experimented systems (see (134)). The high importance (i.e., weight) of the semantic metric probably derives from the fact that even for packages with good structural cohesion there might be pairs of classes with no structural interaction, e.g., two classes with no method calls between each other. Note that the method calls that we capture are just a subset of all possible ways in which two classes can be structurally related. We extract method calls statically with a rather simple and conservative analysis of the code using Eclipse's AST parser. A more sophisticated analysis would likely yield additional structural relations, which may increase the weight of the structural component of the combined measure. In the cases where there are no structural relationships between classes, only the semantic metric can help to cluster together these pairs of classes, when needed. This fact is also highlighted by the strong performances decrease affecting our approach when the weight for the semantic metric is equal to zero. It is worth noting that even if our approach is really

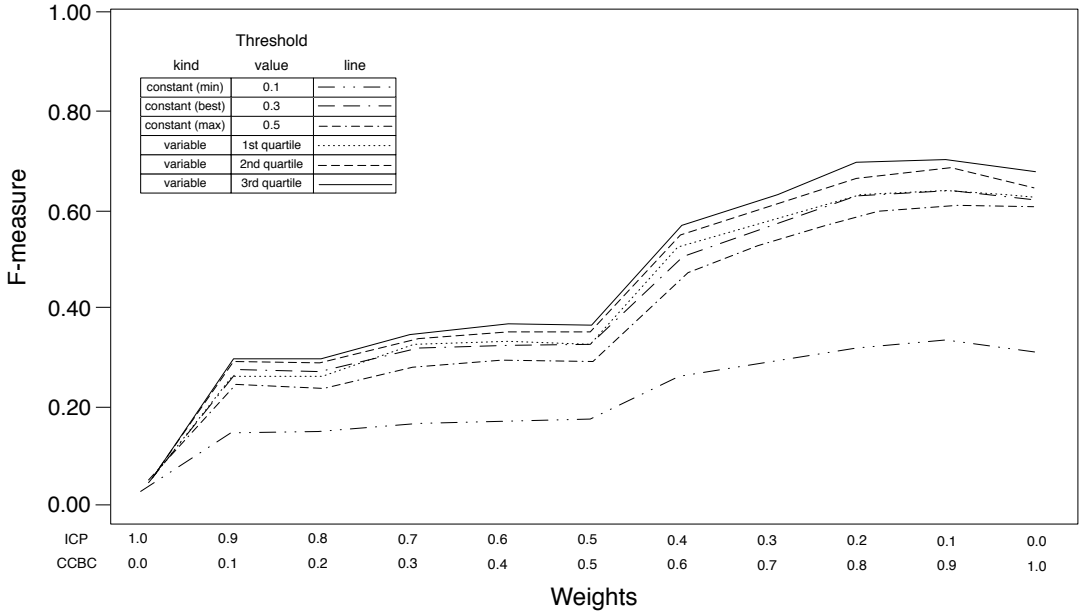

Figure 5.3: Interaction between Weight and Threshold on GESA merging 5 packages.

stable across all the configurations of weights having $w_{CCBC} \geq 0.7$, it shows slight decrease of performances when the structural metric, i.e., ICP, is set to zero (see Figs. 5.1, 5.2, and 5.3). All these observations suggest that the semantic metric captures most of the coupling (relevant to our task) between the classes of the object systems and consequently helps to better cluster together classes of the same original package, i.e., classes with high coupling. To verify such a conjecture we apply PCA to the coupling measures. This allows identifying the different dimensions that describe a phenomenon, e.g., the coupling between pairs of classes, and obtain an indication of the importance of each dimension (captured by one or more coupling metrics) in the description of this phenomenon, i.e., the proportion of variance. It is worth noting that as it is defined, the structural metric ICP gives a value different than zero only in case two classes are related at least by one method call. The problem is that the pair of classes that are related through a method call represents only a small percentage of the possible pairs of classes in a software system, e.g., in JHotDraw "only" 2% out of about 38,000 possible pairs of classes have method calls between them (about 650). In such a situation, the output of the PCA is trivial since it assigns almost all the description of the observed phenomenon, i.e., coupling between classes, to the semantic metric. To

avoid this problem, we executed the PCA only on the pairs of classes related trough at least a method call. Table 5.4 shows the achieved results. As we can see, the semantic metric is identified by the PCA as the metric that describes most of the coupling between pairs of classes (its proportion of variance is always higher than 0.6). Moreover, the proportion of variance values provided by the PCA are close to the weights used for the coupling metrics in the configurations with the best results of the F-measure. Thus, as observed for the Extract Class refactoring approach (see Chapter 4) the PCA could be used to weight the exploited coupling metrics taking into account the portion of coupling captured by each metric (proportion of variance). In particular, the higher the proportion of variance captured by one metric, the higher its weight. To verify the usefulness of the PCA in setting the weights for the metrics exploited by our approach we formulated an additional research question:

- **RQ**$_3$: Can the proportion of variance obtained by PCA be used to weight the coupling metrics exploited in our approach?

To respond to this research question, we compared the results obtained using as configuration parameters the one identified by the proportion of variance of the PCA, i.e., *PCA-based configuration*, with the best results obtained in our experimentation, i.e., *best configuration*. Table 5.5 reports the achieved results. As we can see the difference between the reconstruction accuracy of the *PCA-based configuration* compared with the accuracy obtained using the best configuration is very small, i.e., the difference of F-measure is never higher than 0.04. This result indicates that the *PCA-based configuration* provides an accuracy similar to the best accuracy obtained exploiting all possible configurations. Such results indicate that the proportion of variance provided by the PCA can be used to weight the corresponding metric also in this approach.

Given these findings we propose the following heuristics to set the parameters of our approach in a real usage scenario:

- *minCoupling*: use the third quartile of the non-zero values of the class-by-class matrix as threshold to remove spurious relationships between the classes of the package to re-modularize.

- *weights*: the weights assigned to the structural and semantic metrics are established for the system under analysis by performing the PCA of the values of the

Table 5.4: Results of PCA: Rotated Components

	PC1	PC2
Proportion of Variance	0.62	0.38
Cumulative Proportion	0.62	1.00
CCBC	0.96	0.26
ICP	0.26	-0.96

(a) eTour

	PC1	PC2
Proportion of Variance	0.90	0.10
Cumulative Proportion	0.90	1.00
CCBC	0.99	-0.05
ICP	0.05	0.99

(b) GESA

	PC1	PC2
Proportion of Variance	0.82	0.18
Cumulative Proportion	0.82	1.00
CCBC	-0.99	0.07
ICP	-0.07	-0.99

(c) JHotDraw

	PC1	PC2
Proportion of Variance	0.77	0.23
Cumulative Proportion	0.77	1.00
CCBC	-0.98	0.21
ICP	-0.21	-0.98

(d) SESA

	PC1	PC2
Proportion of Variance	0.94	0.06
Cumulative Proportion	0.94	1.00
CCBC	-0.99	-0.01
ICP	-0.01	0.99

(e) SMOS

coupling metrics computed on the pair of classes of the system having $ICP > 0$. The value of the proportion of variance obtained for each metric will be used as the weight for the corresponding metric.

5.3.2.2 Qualitative evaluation

Even if our approach is able to reconstruct merged packages with very high accuracy, in a minority of cases it does not reconstruct the original packages and proposes an alternative decomposition of the system. In order to understand if the proposed decomposition is still meaningful, even when different from the original, the developers analyzed the proposed re-modularizations. To select the cases to analyze, we set an F-measure threshold $\epsilon = 0.7$; all the cases under this threshold, i.e., 35 of 150, were analyzed by the developers. Table 5.6 reports the answers to the question *"Is the proposed package decomposition meaningful?"* given by the developers for each analyzed case.

Table 5.5: Results reconstructing merged classes: PCA based *vs* best configuration

System	# Merged Classes	Best configuration	PCA-based configuration
eTour	2	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.89**)	$w_{CCBC} = 0.6$, $w_{ICP} = 0.4$ (**0.85**)
	3	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.76**)	$w_{CCBC} = 0.6$, $w_{ICP} = 0.4$ (**0.74**)
	5	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.67**)	$w_{CCBC} = 0.6$, $w_{ICP} = 0.4$ (**0.63**)
GESA	2	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.97**)	*same as the best configuration*
	3	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.82**)	*same as the best configuration*
	5	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.72**)	*same as the best configuration*
JHotDraw	2	$w_{CCBC} = 0.7$, $w_{ICP} = 0.3$ (**0.79**)	$w_{CCBC} = 0.8$, $w_{ICP} = 0.2$ (**0.77**)
	3	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.72**)	$w_{CCBC} = 0.8$, $w_{ICP} = 0.2$ (**0.72**)
	5	$w_{CCBC} = 0.8$, $w_{ICP} = 0.2$ (**0.69**)	*same as the best configuration*
SESA	2	$w_{CCBC} = 0.8$, $w_{ICP} = 0.2$ (**0.93**)	*same as the best configuration*
	3	$w_{CCBC} = 0.8$, $w_{ICP} = 0.2$ (**0.70**)	*same as the best configuration*
	5	$w_{CCBC} = 0.8$, $w_{ICP} = 0.2$ (**0.66**)	*same as the best configuration*
SMOS	2	$w_{CCBC} = 0.8$, $w_{ICP} = 0.2$ (**0.80**)	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.80**)
	3	$w_{CCBC} = 0.8$, $w_{ICP} = 0.2$ (**0.77**)	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.77**)
	5	$w_{CCBC} = 0.9$, $w_{ICP} = 0.1$ (**0.67**)	*same as the best configuration*

In parenthesis the reconstruction accuracy, i.e., the average F-Measure

In particular, the table reports the number of students that have answered one of the possible options. Moreover, Table 5.6 assigns a unique ID to each re-modularization operation evaluated by the students. This is done to easily reference the operations in the discussion of the results. As we can see in Table 5.6, the developers marked as meaningful most of the re-modularization operations suggested by the tool. It is worth noting that the answers given by the students for each of the analyzed cases never differ by more than one point on the Likert scale, which indicates high agreement among them.

Three of the cases for which the developers gave a positive evaluation will be the object of discussion in the following. In particular, $id = 1$ for eTour, $id = 14$ from GESA, and $id = 21$ from SMOS. In this discussion we will use *topic maps* (92) to represent the main topics in a package. There are many ways to determine topics in source code. In particular, given a generic set of classes S, e.g., a package or a group of packages, it is possible to derive the main topics in S by analyzing the term frequency in the classes it contains. We count, for each term present in S, the number of classes

Table 5.6: Analysis of the failure cases. Answers to the question: *"Is the proposed package decomposition meaningful?"*

System	#Subjects	ID operation	1: Fully agree; 5: Strongly disagree.				
			1	2	3	4	5
eTour	2	1	2	-	-	-	-
		2	-	1	1	-	-
		3	-	1	1	-	-
		4	1	1	-	-	-
		5	-	-	2	-	-
		6	-	1	1	-	-
		7	-	-	-	-	2
JHotDraw	2	8	1	1	-	-	-
		9	-	2	-	-	-
		10	-	-	2	-	-
		11	-	1	1	-	-
		12	-	-	2	-	-
		13	-	-	2	-	-
GESA	5	14	4	1	-	-	-
		15	-	-	-	3	2
		16	-	3	2	-	-
		17	-	3	2	-	-
		18	2	3	-	-	-
		19	-	1	4	-	-
		20	-	-	2	3	-
SMOS	5	21	5	-	-	-	-
		22	-	-	-	-	5
		23	-	-	3	2	-
		24	-	-	4	1	-
		25	-	-	3	2	-
		26	-	1	4	-	-
SESA	2	27	-	-	-	-	2
		28	1	1	-	-	-
		29	-	1	1	-	-
		30	-	-	2	-	-
		31	-	-	2	-	-
		32	-	1	1	-	-
		33	-	-	-	2	-
		34	1	1	-	-	-
		35	-	2	-	-	-

that contain it with a frequency higher than 3. The five most frequent terms, i.e., the terms present in the highest number of classes, are then used to construct the topic map

of S that for this reason is represented by a pentagon, where each vertex represents one of the main topics. Each vertex is connected to the center of the pentagon by an axis representing the percentage of classes in S that implements the corresponding topic. The graphical representation of the main topics of S is then obtained by tracing lines between the point on each of the five axes indicating the percentage of classes belonging to S that implement the corresponding topic.

First, we present some of the cases for which the developers gave negative evaluations (i.e., high values on the Likert scale; 4 or 5).

GESA – merging two packages

In a first case negatively evaluated by the developers ($id = 15$ in Table 5.6) the tool merged the following three packages: *examSessionManagement*, *timetableManagement*, and *classroomManagement*. The last two packages were reconstructed by our approach as a single package. The re-modularization is most likely due to the semantic similarity of the two packages. In fact, the *timetableManagement* package contains, among others, classes for the management of the classroom timetable, which use identifiers similar to those used in *classroomManagement*. Another case where the developers did not agree with the suggested re-modularization is represented by $id = 22$ in Table 5.6. In this case the tool merged five packages and only four packages were reconstructed by our approach. In particular, the *connectionManagement* package, which contains the set of classes responsible of the connection to the DBMS, was merged with the *classRegisterManagement* package. This is due to the strong structural relationships (i.e., method calls) between the classes of the two packages. In fact, the *classRegisterManagement* groups all the classes assigned to operations related to the class register, e.g., absence, delay, disciplinary note, and most of these classes access the persistent data in the DBMS using the classes in the *connectionManagement* package.

SESA – code clones

Two other interesting cases negatively evaluated by the developers regard the SESA software system ($id = 27$ and $id = 33$ in Table 5.6). Both these cases have a common reason that caused the failure of our approach. In particular, several classes in SESA contain "source code clones", e.g., pieces of codes copied and pasted among different classes. Moreover, the comments used to describe the responsibilities assigned to the classes follow a standard template containing a set of words shared between almost all the classes of the system. This clearly results in a high semantic similarity even between

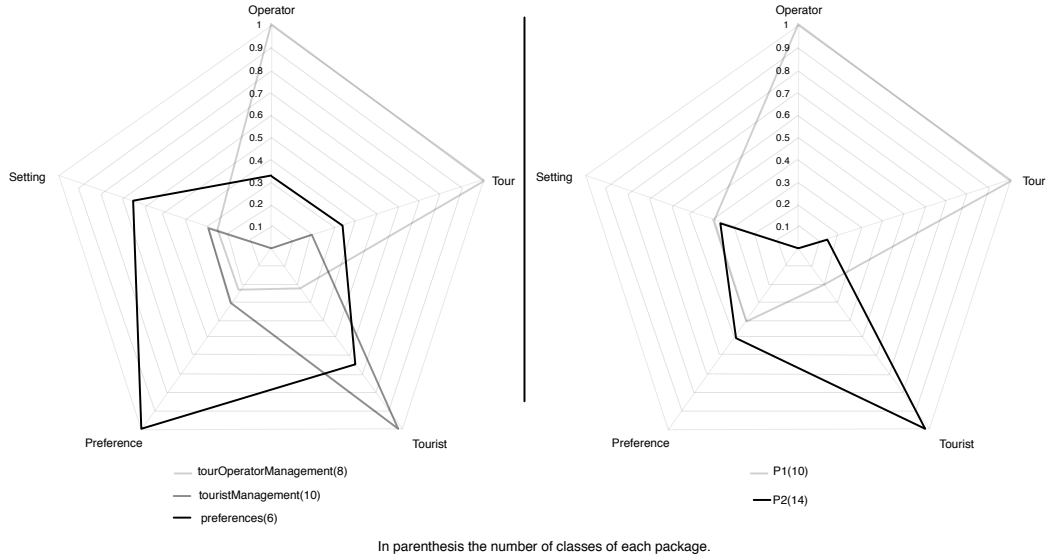

Figure 5.4: Topic Map eTour: original packages *vs* new packages.

classes having different responsibilities. Thus, our approach fails to reconstruct the original packages when it attempts to decompose a package created by merging packages containing classes with source code clones and/or very similar comments. A possible solution to this limitation can be found in (135), where the authors propose the use of smoothing filters to improve the performances of the IR-based traceability recovery techniques. In particular, these filters reduce the weight of terms that frequently occur among different artifacts (in our case classes), improving the precision of the IR method. The application of this kind of filters to our approach is outside the scope of this paper. However, we think that our approach could only benefit from the use of the smoothing filters.

eTour – removing the preferences package

A first "failure" case positively evaluated by the developers regards the eTour software system. The following three packages were merged together in a single artificial package:

- *tourOperatorManagement*: this package contains all the classes responsible with the management of the users registered to the system as tour operators.
- *touristManagement*: similar to the previous package, touristManagement groups together all the classes resposible with the management of the users registered as

tourists.

- *preferences*: package containing classes used to manage the preferences set by the users of the system, e.g., favored cities for the tourist.

When applied to the resulting package, our technique suggested only two packages, with the six classes from the *preferences* package distributed among the two suggested packages. In particular, two classes were moved to the *tourOperatorManagement* package while four classes were moved to the *touristManagement* package. We deeply analyzed the original package in order to find some explanations. We observed that in eTour three kind of roles are defined for the users: administrator, tour operator, and tourist. However, only the last two have the possibility to customize the system using the preference panel. In fact, two out of the six classes present in *preferences* are responsible for the management of the tour operator's preferences, while the remaining four deal with the tourist's preferences. The high method interactions and semantic consistency between the classes present in the *tourOperatorManagement* (*touristManagement*) package and the two (four) classes responsible of the management of the tour operator's (tourist's) preferences explain the output of our approach. Fig. 5.4 show the topic map of the packages pre and post the re-modularization. It is worth noting that the topics assigned to the new packages are almost the same as the topics assigned to the original *touristManagement* and *tourOperatorManagement* packages.

GESA – two packages instead of three

In this case the tool merged in a single package the following three packages, all belonging to the application layer:

- *lessonNegotiationManagement*: GESA has a feature that allows teachers to negotiate the teaching schedule. This package contains all the classes responsible for the schedule negotiation.

- *reportManagement*: this package contains a set of classes that provide different kinds of reports to the administrator of the system. The reports contain a schematic representation of a portion of the persistent data in the system.

- *userManagement*: this package contains all the classes assigned to the management of the system's users.

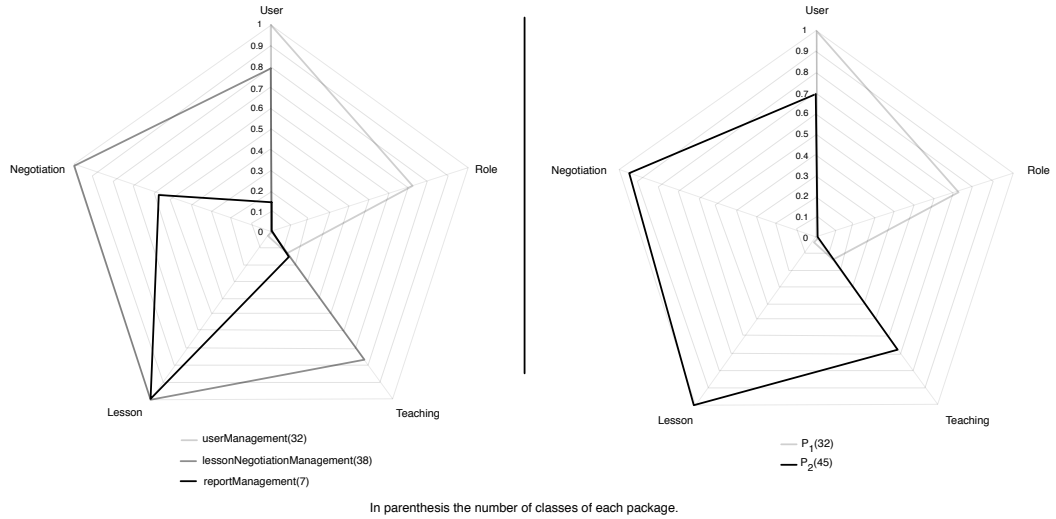

Figure 5.5: Topic Map GESA: original packages *vs* new packages.

We applied our approach to reconstruct the original packages. However, the output of the tool was significantly different from the original package decomposition. In fact, instead of three packages the approach reconstructed only two packages; we call them P_1 and P_2. In particular, we noticed that P_1 is equal to the *userManagement* package, i.e., it contains the same set of classes, while P_2 is the result of merging the *lessonNegotiationManagement* package with the *reportManagement* package. To understand if the proposed decomposition is still meaningful we compared the topic map of the original packages with the topic map of the new packages (see Fig. 5.5). Besides confirming the semantic equivalence of the packages *userManagement* and P_1, from the analysis of Fig. 5.5 we can observe that in the original decomposition the main topics of *reportManagement* were completely subsumed by the main topics of *lessonNegotiationManagement*. Analyzing the package *reportManagement* we noticed that six out of the seven classes from this package provide the administrator with reports directly or indirectly concerned with the entity lesson, e.g., course timetable report, classroom timetable report, teacher's lesson report, etc. The high conceptual coupling between *lessonNegotiationManagement* and *reportManagement* is also confirmed by the similarity of the topic map representing package *lessonNegotiationManagement* and the topic map representing package P_2 (see Fig. 5.5). So, while the original decomposition is probably meaningful from a functional point of view, one can argue that the proposed

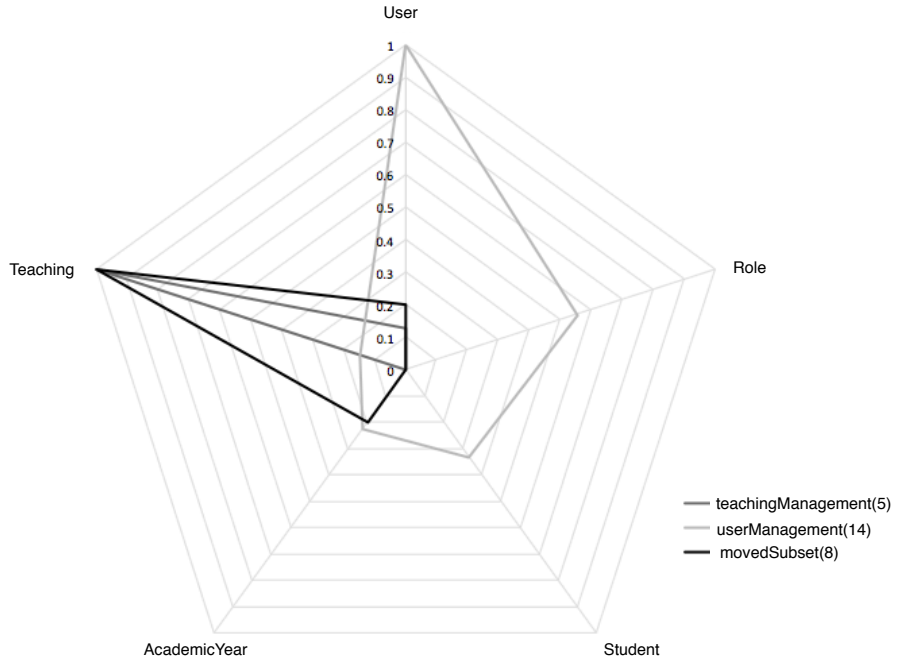

Figure 5.6: Topic Map of the moved classes in the SMOS re-modularization.

decomposition is still semantically meaningful. This observation is useful in planning our future research, as we could adapt our approach not only to split packages, but also to recommend existing packages to be merged, when needed. As mentioned earlier, design decisions regarding the structure of a system involve more considerations than just high cohesion and low coupling. This is clearly such a case, where the user may or may not agree with our tool's suggestion in the end. It is important to note also that the semantic analysis based on IR techniques, such as, LSI, is independent of grammar and domain models, while it is dependent on consistent use of terms in the analyzed code. Inconsistencies in the use of terms usually affect such an analysis negatively.

SMOS – moving classes between packages

The last case is from the SMOS software system. In particular, the following packages were merged from the application layer:

- *teachingManagement*: this package contains all the classes assigned to the management of the lectures.

- *userManagement*: this package contains all the classes assigned to the management of the users.

In this case, the number of packages reconstructed is equal to the number of original packages (two), yet our approach applied an interesting operation. Specifically, a subset of eight classes from the *userManagement* package was moved in the *teachingManagement* package. Analyzing this set we noticed that the eight classes implement operations regarding the management of the association between users, i.e. teachers, and teaching assignments (e.g., assign a new teaching assignment to a teacher, show the teacher's assignments, etc.). The topic map shown in Fig. 5.6 underlines that the moved set of classes is semantically closer to the *teachingManagement* package than to the *userManagement* package. This indicates that probably the set of moved classes is more suited in the teachingManagement package than in the userManagement package. Indeed, although it is different from the original design, the proposed modularization has been evaluated as meaningful by the developers.

5.3.3 Threats to Validity

All the findings of our study might be affected by several threats to validity (136) discussed in the following.

5.3.3.1 System Mutation

We decided to use the proposed approach to split previously merged packages and then evaluated the re-modularization accuracy comparing the split packages with the original packages. While the object systems were chosen because they are generally well designed, there is the risk that the original packages are not a good oracle. To mitigate such a threat we analyzed the package decomposition of the subject systems in order to ensure its meaningfulness. Moreover, as we can see in Table 7.1, the cohesion of the packages in the object systems is very high on average, which is also an indicator of good modularization. In fact, the same metrics were used in (82) to evaluate the decomposition quality of several open source systems, e.g., ArgoUML, JEdit. The metric values obtained by the object systems are much better than the values obtained by the systems in (82). Moreover, JHotDraw is generally considered a well-designed system and it was developed using several design patterns. eTour, GESA, SESA, and

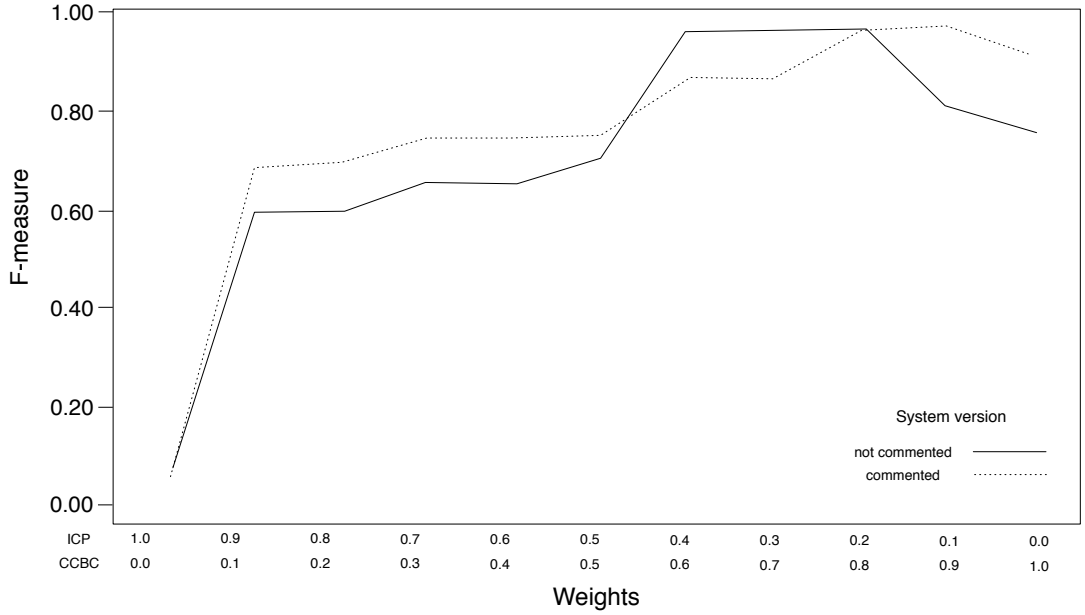

Figure 5.7: Performances on $GESA_{Comments}$ and on $GESA_{NoComments}$ merging 2 packages.

SMOS were selected among the best systems developed during a Software Engineering course and, as we can see in Table 7.1, have an average package cohesion comparable to JHotDraw.

Another aspect of the evaluation is represented by the choice to split previously merged packages. Indeed, given the high quality of the subject systems, the coupling between classes from different packages is generally low. Thus, the splitting operation seems to be trivial. However, we observed that in several cases, classes from different packages have many structural dependencies between them, e.g., such a situation is typical of classes from the subsystems responsible with the management of the system's users. In such cases, the semantic measures avoid the creation of class chains with different responsibilities and help in the reconstruction of the original packages.

5.3.3.2 Experiment Design and Results Analysis

The meaningfulness of the proposed re-modularization operations was evaluated using the F-measure, based on the precision and recall that reflect the reconstruction accuracy

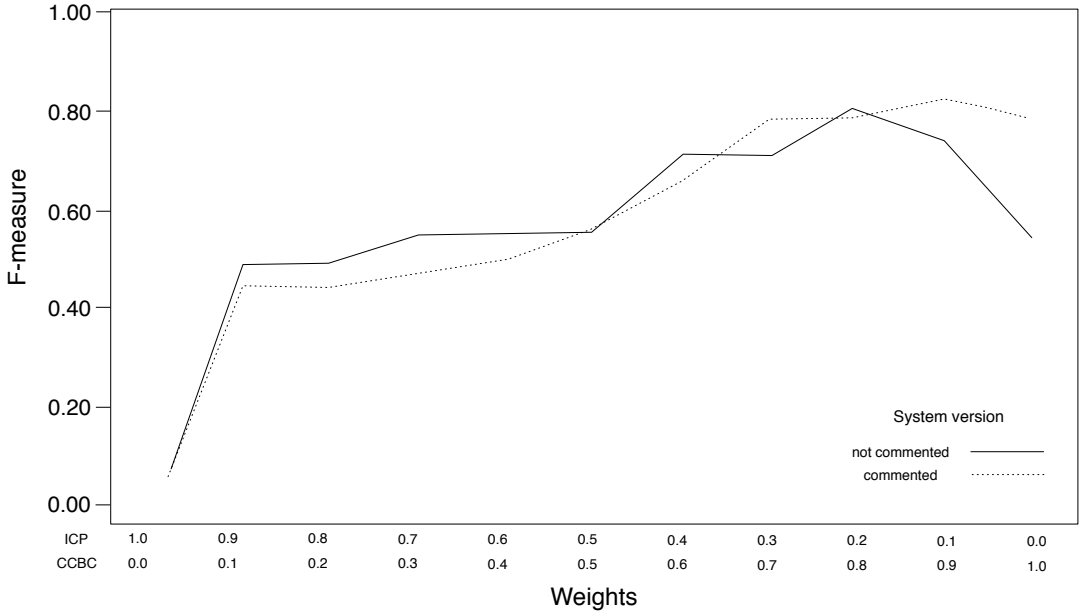

Figure 5.8: Performances on $GESA_{Comments}$ and on $GESA_{NoComments}$ merging 3 packages.

of the proposed approach. The same approach was also used in previous work on class refactoring (1, 17, 18, 115, 137).

To better evaluate the suggested re-modularizations, we also analyze the cases where our approach does not reconstruct the original package decomposition. In particular, several students, familiar with the systems, analyzed the proposed alternative re-modularization and evaluated its meaningfulness. The students did not know the goal of our experimentation to avoid bias, however as in any such situation, user subjectivity is part of the evaluation.

5.3.3.3 The Role of CCBC in Software Re-modularization

In order to find the optimal setting of parameters for our approach, we analyzed several different configurations (see Section 7.3.1). The results showed that the weight w_{CCBC} for the semantic metric should be really high, generally higher than 0.7, to obtain good performances. CCBC is highly dependent on the quality of the identifiers and comments in the code, so, given its high influence on the approach, we expect the approach also to be sensitive to the quality of the comments and identifiers. All

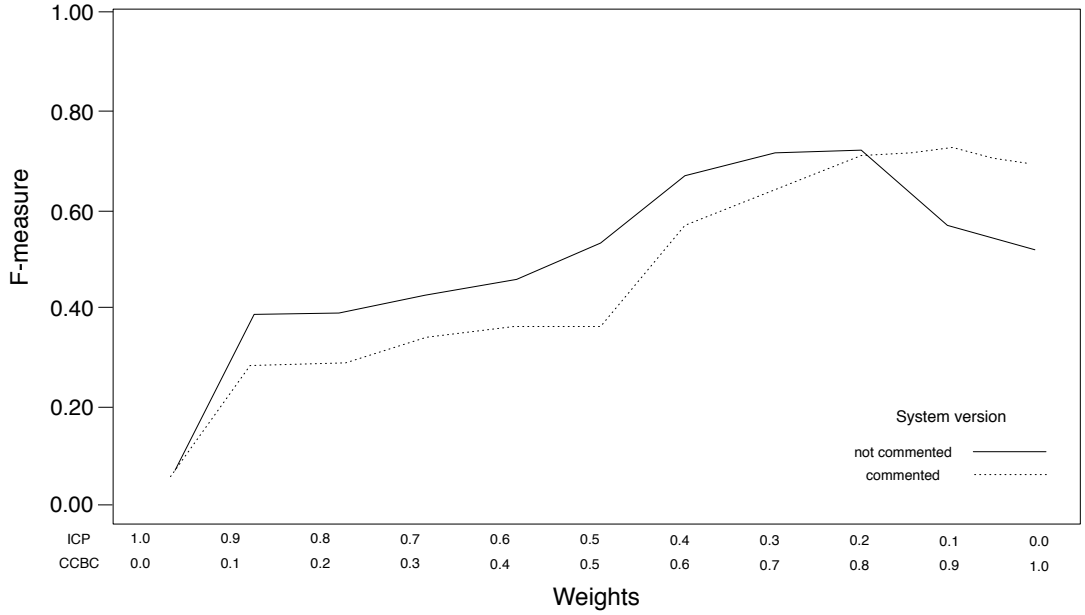

Figure 5.9: Performances on $GESA_{Comments}$ and on $GESA_{NoComments}$ merging 5 packages.

the experimented systems have well commented classes besides exhibiting a generally good package decomposition. Thus, to investigate the performances of our approach (and the influence of the configuration parameters) in a completely different scenario, we removed all the comments present in the source code from the object systems and re-executed our experimentation. The goal was to assess the impact of the comments on the approach. The results show again that the best performances of our approach are achieved using as threshold the third quartile of the class-by-class matrix, as in the cases where comments are included. Regarding the weights, Figs. 5.7, 5.8, and 5.9 compare the performances of our approach on GESA with comments ($GESA_{Comments}$) and GESA without comments ($GESA_{NoComments}$) using all the possible combinations of weights and merging 2, 3, and 5 packages, respectively[1]. It is worth noting that the reconstruction accuracy achieved by our approach is almost identical on the two "versions" of the system. However, our approach applied on $GESA_{NoComments}$ shows a strong decrease in performance when the weights assigned to the semantic metric is higher than 0.8. Moreover, while the best performances on $GESA_{Comments}$ are

[1]The complete results achieved with the other systems can be found in (134)

achieved setting $w_{CCBC} = 0.9$, on $GESA_{NoComments}$ the best results are obtained setting $w_{CCBC} = 0.8$. Thus, even if the weight for the semantic metric slightly decreases, its contribution in the software re-modularization remains essential. This is most likely due to the semantic information present in the identifiers used by the developers. Note that, the need to reduce the weight of the semantic metric in order to achieve good results in the *no comment scenario* is confirmed on all the experimented systems (see (134)).

Table 5.7: Results of PCA on the "NoComments" systems

	PC1	PC2
Proportion of Variance	0.55	0.45
Cumulative Proportion	0.55	1.00
CCBC	-0.84	0.45
ICP	-0.45	-0.84

(a) eTour

	PC1	PC2
Proportion of Variance	0.80	0.20
Cumulative Proportion	0.80	1.00
CCBC	0.99	-0.07
ICP	0.07	0.99

(b) GESA

	PC1	PC2
Proportion of Variance	0.68	0.32
Cumulative Proportion	0.68	1.00
CCBC	-0.99	0.07
ICP	-0.07	-0.99

(c) JHotDraw

	PC1	PC2
Proportion of Variance	0.71	0.29
Cumulative Proportion	0.71	1.00
CCBC	0.97	0.25
ICP	0.25	-0.97

(d) SESA

	PC1	PC2
Proportion of Variance	0.84	0.16
Cumulative Proportion	0.84	1.00
CCBC	-0.99	0.00
ICP	0.00	0.99

(e) SMOS

5.3.3.4 On the Use of PCA as Heuristic to Set the Metric Weights

The experimentation performed in the *no comment scenario* allows also to further investigate the validity of the proposed PCA-based heuristic to set the metric weights. In fact, we expected that applying the PCA on the object systems without comments, we obtain a decrement of the proportion of variance assigned to the semantic metric,

i.e., CCBC, with a consequent decrement of its weight, i.e., w_{CCBC}. To verify such a conjecture, we re-executed the PCA on the "NoComment" versions of the object systems. Table 5.7 reports the achieved results. Analyzing Figs. 5.7, 5.8, and 5.9, and Table 5.7, it is easy to see that in this scenario also, the weights suggested by the PCA on the GESA system, i.e., $w_{ICP} = 0.2$ and $w_{CCBC} = 0.8$, result in a configuration with performances very close to the best. This trend is confirmed on all the experimented systems (see (134)) and further supports the possibility to use PCA as a heuristic to set the metric weights.

5.4 Final Remarks

In this Chapter is presented and evaluated a technique that suggests decompositions of promiscuous packages to improve their cohesion. Central to proposed approach, and a departure from previous work, is the combined use of structural and semantic relationships between classes in this context. The evaluation revealed that the technique produces meaningful decompositions from structural and functional point of view.

Although the approach has been applied on five realistic systems, as with all empirical studies, the generalization of our findings cannot be ensured. Thus, replicating the study on other systems is the only way to corroborate the results achieved and mitigate the external threat to validity related to the generalization of our findings.

6

Move Method Refactoring

The material in this Chapter has been presented in (138, 139).

6.1 Introduction

In this Chapter is presented Methodbook, an approach to support the software engineer in identifying move method refactoring opportunities. Methodbook represents the instantiation of the approach presented in Section 3.5 for moving misplaced code components to the move method refactoring. 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. 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 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) (62) 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 (i.e., group of methods), the class that contains the highest number of "friends" of the method under analysis.

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

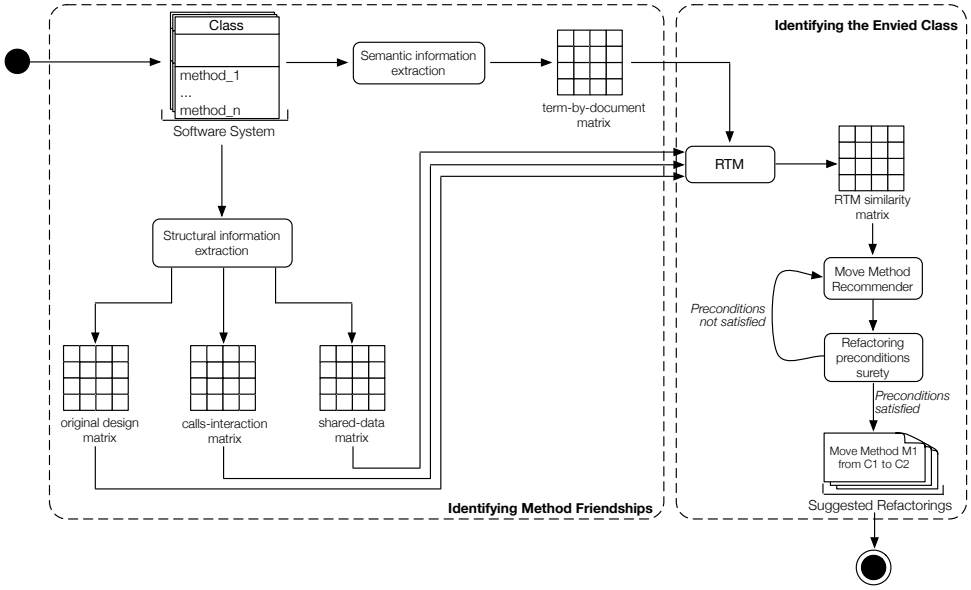

Figure 6.1: Methodbook: the process.

In this paper we evaluate the usefulness of Methodbook in two case studies. In the first study we evaluated Methodbook on five software systems through well-established metrics that capture 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 thirty academic software developers on an open source software system.

Methodbook is overviewed in Section 6.2. Section 6.3 reports the first case study where Methodbook has been evaluated via quality metrics, while Section 6.4 reports the results of the study with users.

6.2 Methodbook

In a nutshell, Methodbook works as depicted in Figure 6.1. The process used by Methodbook to identify move method refactoring operations is composed of two main steps: (i) identification of methods' friendships[1], and (ii) identification of the envied

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

class. In the first step, semantic and structural information is extracted from the source code. The semantic information is represented by words in comments and identifiers of in source code and is stored in the *term-by-document* matrix. This matrix is used by RTM to derive semantic relationships between methods and define a probability distribution of topics (topic distribution model) among methods. Besides semantic 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 methods, besides semantic 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 6.1). 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"* (3). 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 as an envied class. For this reason, in the second step of Methodbook, the envied class is identified by analyzing the classes containing the 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 suggests any refactoring operation. Otherwise,

Methodbook verifies if a set of preconditions ensuring the preservation of the system behavior post-refactoring are satisfied for the suggested envied class; if the preconditions are satisfied, the refactoring is suggested, otherwise Methodbook will look for a new envied class for m_i. 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). The interested reader can find more information about RTM in Section 3.5, while in the next subsections we further detail about the two steps behind the Methodbook's process.

6.2.1 Identifying Method Friendships

The *method friendships* are identified using RTM through the analysis of structural and semantic relationships among methods as well as the original structure of the classes (see Figure 6.1).

As the very first step, methods are analyzed to extract words contained in comments and identifiers. In order to extract words from compound identifiers and comments, advanced algorithms for splitting identifiers are employed (140). 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 6.1). 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 (106).

A light-weight static analysis[1] 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.

[1]The static analysis is performed using the Eclipse AST parser.

Concerning the structural dependencies among the methods of the system, Methodbook exploits two structural measures, namely, Structural Similarity between Methods (SSM) (110) and Call-based Dependence between Methods (CDM) (1), previously used to compute similarities between methods for identifying Extract Class refactoring opportunities (1, 115). These measures do not correlate and capture two orthogonal aspects of method relationships (1). In particular, SSM captures attribute references in methods and it is used to build the *shared-data matrix* while CDM (1) takes into account the calls performed by the methods and it is used to build the *calls-interaction matrix*. The interested reader can find the definition of these two measures in Section 3.2.

The set of friendships derived by analyzing structural similarity between methods are supplied as existing links to RTM. The basic idea behind RTM is that textual documents (that is, methods represented by the *term-by-document matrix*) are modeled 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 (62). The strength of RTM as compared to other topic modeling techniques is in its ability 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 semantic 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 6.1).

6.2.2 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 dimension. 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 by 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 the class C_j contains a 75% of m_k' friends (3/4). To avoid this issue, the envied class is identified as the one containing the higher percentage of m_k' best friends 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 *et. al* (20) to ensure that the program behavior does not change after the application of the suggested refactoring. These preconditions are classified in three different categories (20): (i) compiling preconditions, e.g., the envied class does not contain a method having the same signature with the moved method, (ii) behavior-preservation preconditions, e.g., the envied class should not inherit a method having the same signature with 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 can be found in (20). If refactoring involving identified envied class satisfies all the preconditions, the move method operation is suggested by Methodbook. 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 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 cases where identification of an envied class is trivial, i.e., there is a class containing a sensibly higher percentage of friend methods as compared to 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 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 (141). 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., it is more difficult 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 is the set of classes containing identified method friends for the method to be moved m, while $l(c)$ represents the likelyhood 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 of [0, 1]. The higher the value, the higher might be the goodness of provided recommendation.

Figure 6.2 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 as compared to 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).

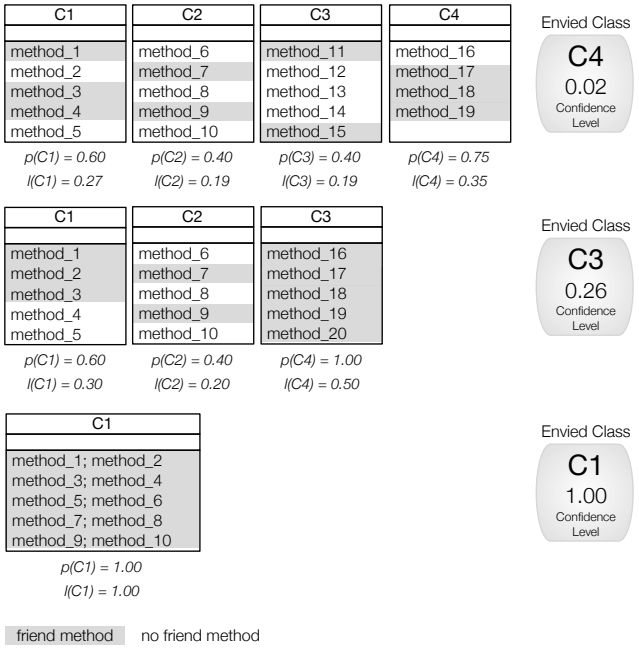

Figure 6.2: Three examples of envied class identification with different confidence levels.

6.3 Evaluation Based on Quality Metrics

The *goal* of this case study is to evaluate the goodness of the move method refactoring operations recommended by Methodbook. Good move method recommendations should help improve the design quality of a given software system in terms of class cohesion and coupling. In fact, these are the two main quality aspects that can be improved while performing this kind of refactoring.

The experimentation was carried out on an open source software system, JHotDraw[1], on three industrial projects, namely AgilePlanner[2], eXVantage[3], and GESA[4], and on a software system, SMOS, developed by a team of Master students at the University of Salerno (Italy) during their industrial traineeship. JHotDraw is a Java GUI framework for technical and structured graphics. 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, focuses on providing code

[1]http://www.jhotdraw.org/ verified on 01/12/2013
[2]http://ase.cpsc.ucalgary.ca/
[3]http://www.research.avayalabs.com/
[4]http://www.distat.unimol.it/gesa/

coverage information to software developers and testers. GESA automates the most important 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 communications between the school and the students' parents.

Table 6.1 reports the size, in terms of KLOC, number of classes, and number of methods, and the versions of the object systems. The table also reports the average value for four quality metrics aimed at measuring structural and semantic class cohesion and coupling, namely Connectivity (142), Conceptual Cohesion of Classes (C3) (60), Message Passing Coupling (MPC) (109), and Conceptual Coupling Between Classes (CCBC) (55), calculated 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.* (142), they can artificially increase the class cohesion. C3 is a conceptual cohesion metric, complementary to structural cohesion, which exploits LSI (Latent Semantic Indexing) (91) to compute the overlap of semantic 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 semantic information (i.e., lexical information) captured in the code by comments and identifiers. Two classes are conceptually related if their (domain) semantics are similar, i.e., the terms present in their comments and identifiers are similar.

We evaluate the impact on these four quality metrics of the move method operations recommended by Methodbook. 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 and/or high coupling have been shown to correlate with high defect rates (60).

Table 6.1: Software systems used in the case study

System	KLOC	Classes	Methods	Connectivity$_{avg}$	C3$_{avg}$	MPC$_{avg}$	CCBC$_{avg}$
AgilePlanner 2.5.0	24	299	2,731	0.195	0.220	3.460	0.070
eXVantage 2.01	36	352	2,172	0.240	0.283	2.707	0.077
GESA 2.2	46	295	1,643	0.144	0.082	10.955	0.349
JHotDraw 6.0 b1	29	275	2,976	0.159	0.291	3.084	0.053
SMOS 1.0	23	121	599	0.197	0.082	5.984	0.389
Total	122	990	7,949	-	-	-	-

2. MPC has been shown to directly correlates with maintenance effort (109). 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 subject systems while applying move method operations suggested by Methodbook, this certainly represents a first indication of the goodnesses of the Methodbook recommendations.

6.3.1 Research questions and planning

In the context of our study, the following research questions were formulated:

- **RQ**$_1$: Is Methodbook able to improve the design quality of a software system in terms of class cohesion and coupling?

- **RQ**$_2$: Does the confidence level serve as a good indicator for the goodness of Methodbook recommendations?

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 10,121 methods)[1].

[1]We applied Methodbook in isolation on each object system.

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 ($\mathbf{RQ}_1$), we applied them incrementally starting from those having the higher confidence level (see Section 6.2). 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 subject 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 (20) (using the JDeodorant Eclipse plug-in) on the same five 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 the approach described by Tsantalis and Chatzigeorgiou (20). Note that the two structural quality metrics used in our evaluation, i.e., Connectivity and MPC, were also used in the experimentation of the approach presented by Tsantalis and Chatzigeorgiou (20).

Concerning our second research question ($\mathbf{RQ}_2$), the order in which these refactoring operations are applied (refactoring from those having the higher confidence level to those having low confidence level) allows to analyze a possible correlation between the confidence level and the goodnesses of the suggested refactoring operations. In particular, if the confidence level is a good indicator for the goodness of Methodbook 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 6.2) to ten. Our choice is not random since we tried several different values for this parameter (1, 3, 5, 7, and 10) and, after having manually analyzed the suggestions proposed by Methodbook[1], we believe that the best refactoring recommendations are usually obtained using $\mu = 10$.

The move method recommendations formulated by both Methodbook and JDeodorant are available online[2].

[1]The training of Methodbook has been performed on a system not used in its empirical evaluation.
[2]http://distat.unimol.it/reports/methodbook

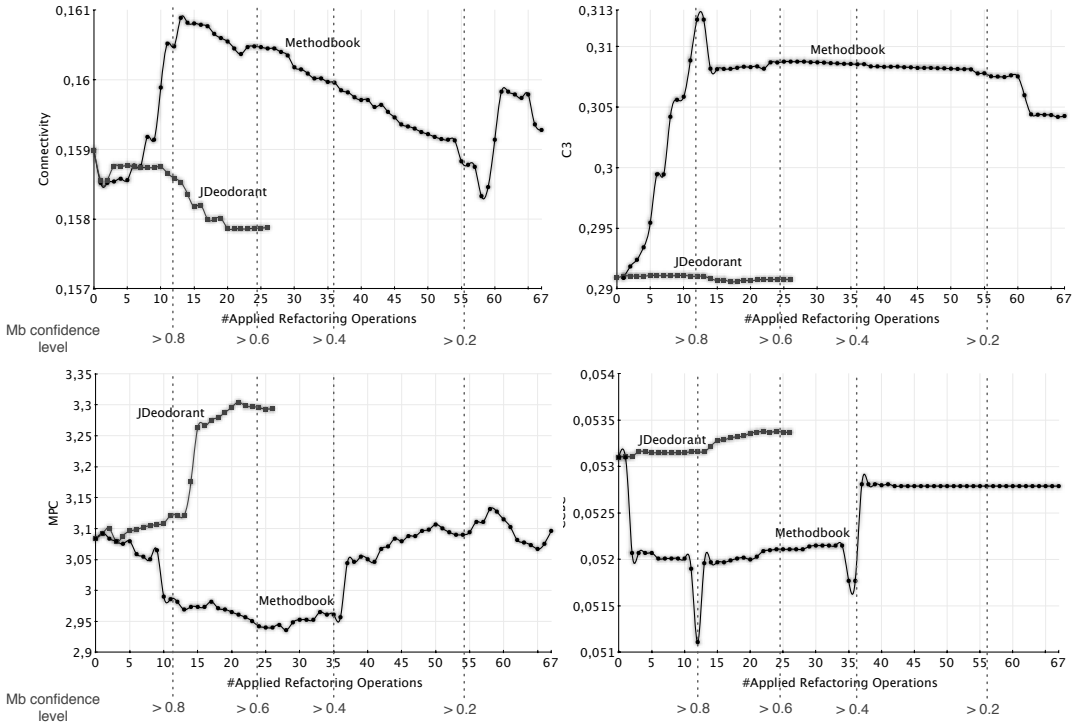

Figure 6.3: Evolution of the four quality metrics on JHotDraw by applying the refactoring operations suggested by Methodbook (67) and JDeodorant (26)

6.3.2 Experiment results

Figures 6.3, 6.4, 6.5, 6.6, and 6.7 show the evolution of four employed quality metrics by applying the refactoring operations suggested by Methodbook and JDeodorant on JHotDraw, SMOS, AgilePlanner, GESA, and eXVantage, respectively. As explained before, the suggestions by Methodbook are applied starting from those having the higher confidence level to those having low confidence level.

The results achieved on JHotDraw (Figure 6.3) show that Methodbook is able to sensibly improve the four cohesion and coupling quality metrics. In particular, the Connectivity cohesion metric shows a strong increase during the application of the first 12 suggestions by Methodbook. Note that these 12 suggestions are those having a confidence level higher than 0.8. Applying these 12 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,

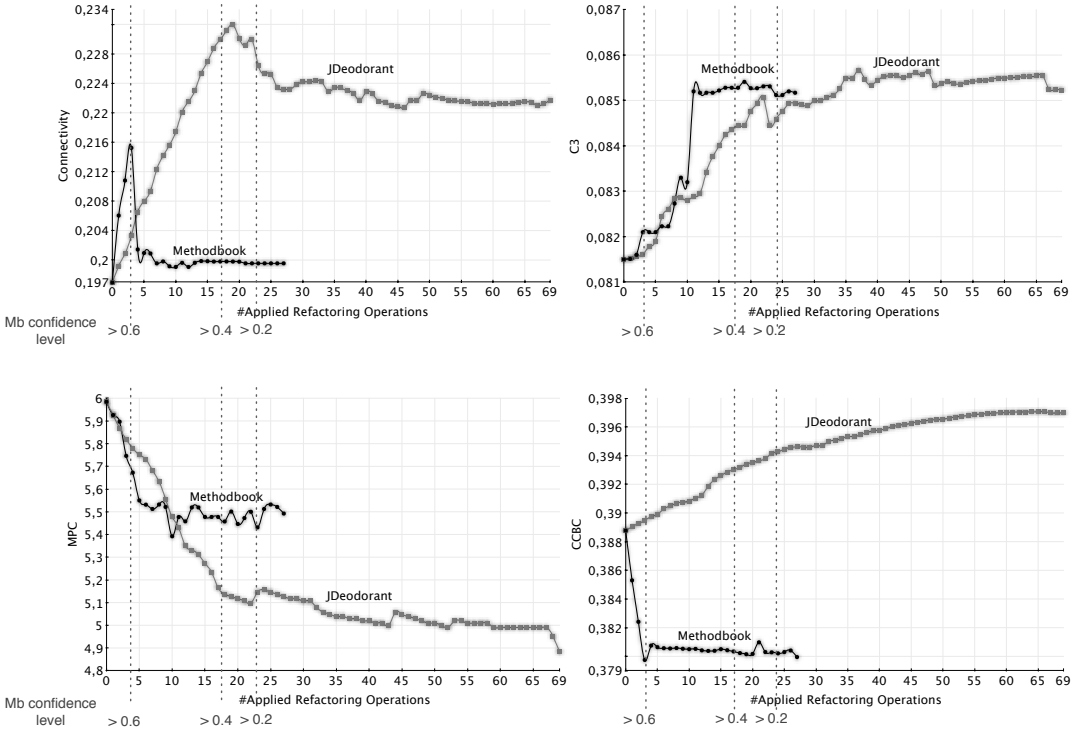

Figure 6.4: Evolution of the four quality metrics on SMOS by applying the refactoring operations suggested by Methodbook (27) and JDeodorant (69)

applying Methodbook's recommendations having a confidence level lower than 0.8 is 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 classes' quality with respect to the quality level reached after the application of the only recommendations having high confidence level.

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 6.3). This means that applying JDeodorant suggestions on this system deteriorates the design quality in terms of class cohesion and coupling.

The situation is totally different on SMOS (see Figure 6.4). 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

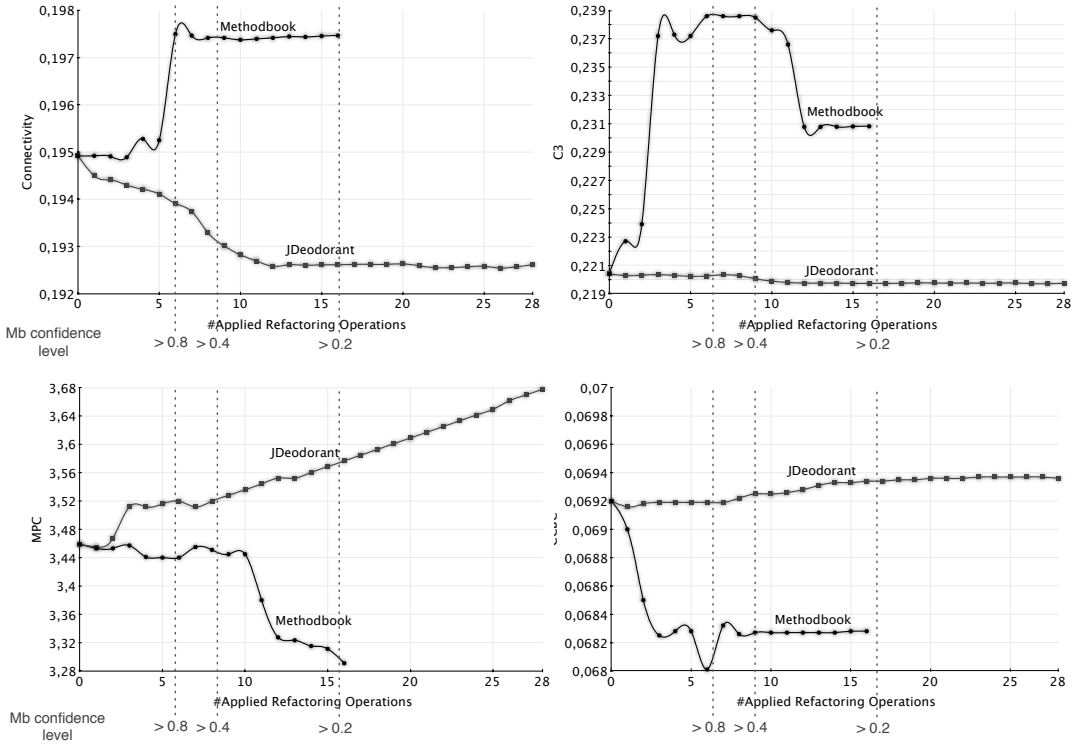

Figure 6.5: Evolution of the four quality metrics on AgilePlanner by applying the refactoring operations suggested by Methodbook (17) and JDeodorant (28)

(the semantic coupling, i.e., CCBC, increases). This result is likely due to the fact that JDeodorant does not take into account semantic information during the generation of the move method recommendations. As for Methodbook, it suggests a lower number of operations compared to JDeodorant (27 *vs* 69) on SMOS. 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 6.4). Concerning the structural metrics, the Methodbook recommendations having confidence level higher than 0.6 are able to strongly improve all the quality metrics, while, as observed on JHotDraw, low confidence level operations are not always able to improve the quality metrics. On the semantic side, as expected, Methodbook performs better than JDeodorant, ensuring increase of semantic cohesion and decrease of semantic coupling.

As for the others software systems, on AgilePlanner and eXVantage the trend is almost the same as JHotDraw (see Figures 6.5, 6.7). In fact, on these systems (i) the

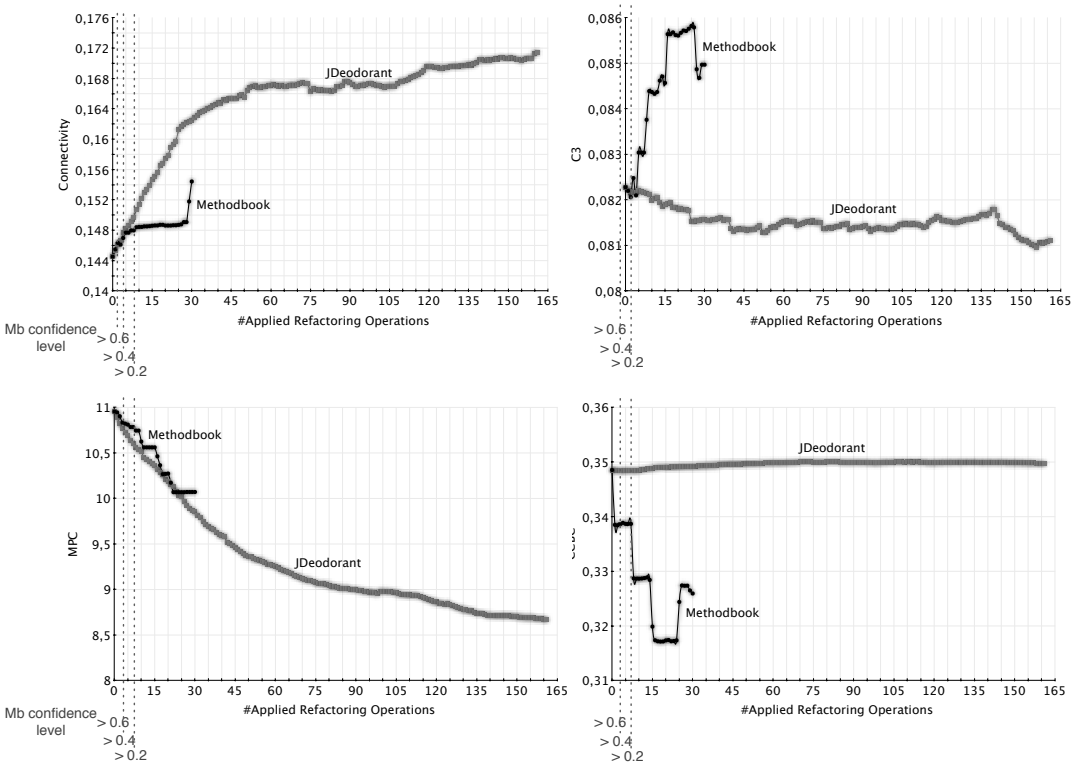

Figure 6.6: Evolution of the four quality metrics on GESA by applying the refactoring operations suggested by Methodbook (30) and JDeodorant (165)

Methodbook's suggestions having high confidence level (i.e., higher than 0.6) strongly increase the structural and semantic cohesion while decrease the structural and semantic coupling, and (ii) JDeodorant generally decreases the design quality of the systems in terms of the exploited quality metrics. Concerning GESA, the results are almost inline with SMOS: JDeodorant is able to strongly 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 6.6). Note that, as for SMOS, the suggestions by Methodbook generally have a low confidence level also in GESA.

Summarizing, our results highlight the importance of the confidence level as indicator of goodnesses of the Methobook's suggestions ($\mathbf{RQ}_2$). In particular, when the confidence level is high (generally higher than 0.6) the Methodbook's recommendations are able to improve the design quality of a software system in terms of class cohesion and coupling ($\mathbf{RQ}_1$). Moreover, on three out of five subject systems (AgilePlanner,

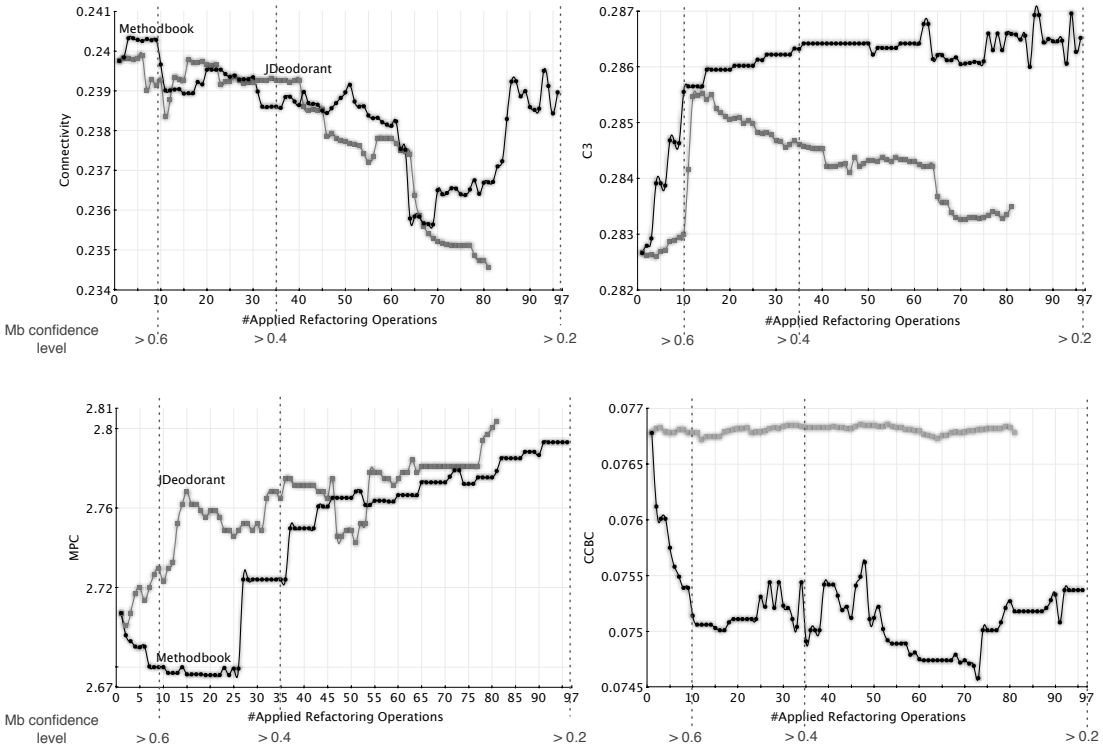

Figure 6.7: Evolution of the four quality metrics on eXVantage by applying the refactoring operations suggested by Methodbook (95) and JDeodorant (80)

eXVantage, and JHotDraw) Methodbook outperforms JDeodorant while on the remaining 2 systems (GESA and SMOS) Methodbook achieves a better quality metrics improvement only on the semantic side.

6.3.3 Threats to validity

This section describes some threats to validity (136) that may affect the results of our first case study.

6.3.3.1 Choice of the 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 employed 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 suggested move method operations. Among the structural metrics, Connectivity and MPC were preferred for several reasons: (i) Connectivity, on the contrary of other structural cohesion metrics, e.g., Lack of Cohesion of Methods (LCOM) (9), 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) (9), and (iii) the same structural metrics were also used in the evaluation of JDeodorant (20). To the best of our knowledge, on the semantic side the C3 metric is the only semantic cohesion metric available in literature while the CCBC was preferred to the semantic coupling metric presented in (143), since the latter is based on RTM, which represents the foundation of Methodbook.

6.3.3.2 Software metrics evaluation

The achieved results indicate that Methodbook is able to improve the quality of subject systems in terms of class cohesion and coupling. Moreover, on three out of the five employed systems it is able to outperforms the state-of-the-art tool JDedorant (20). 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 a developer' point of view. For this reason, we do not only base our evaluation on metrics measurement, but also performed two user studies reported in Section 6.4.

6.4 Evaluating Methodbook with Software Developers

In our previous case study (Section 6.3) we evaluated the Methodbook's recommendations by measuring the difference between pre- and post-refactoring design quality in terms of class cohesion and coupling. However, the refactoring operations should not only improve the quality of a software system in terms of metrics but should also be meaningful from a developer's point of view. For this reason, we performed two studies involving software developers in the evaluation of refactoring operations proposed by Methodbook. The first study was conducted on JHotDraw and involved 30 developers.

Since these subjects have not participated in the development of the subject system, i.e., JHotDraw, 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 of these studies to evaluate Methodbook from all possible perspectives. Indeed, the only study with external developers is not enough since they do not have a 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 suggested by Methodbook 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 from different points of view. Also in these two studies we compare suggestions performed by Methodbook with those performed by the competitive approach proposed by Tsantalis *et al* (20).

In the context of the two studies, the following research question was formulated:

- **RQ**$_3$: Are the refactoring recommendations produced by Methodbook meaningful from a functional point of view?

6.4.1 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.

6.4.1.1 Planning

The study with external developers was executed at the University of Salerno (Italy) and replicated at the University of Molise (Italy) with different software developers. We involved a total of 30 bachelor students, 14 at the University of Salerno and 16 at the University of Molise. All the subjects were third year bachelor students and had good knowledge of Java and object oriented programming. In the context of this study we compared the meaningfulness of refactoring recommendations proposed by

Methodbook with those proposed by JDeodorant on JHotDraw in order to respond to our research question ($\mathbf{RQ}_3$). To this aim, we selected 20 methods from JHotDraw and, for each of them, we asked participants to identify appropriate class(es) where the method could be implemented. The participants evaluated the meaningfulness of the Methobook's refactoring suggestions through a questionnaire. The 20 methods were selected among four different groups. In particular, we selected:

- 5 methods for which only Methodbook suggests to move them from their original class to a new envied class (*onlyMethodbook* group - OM). For each method in this group the participants had three possible options in the questionnaire (three possible classes from JHotDraw): (i) the original class, i.e., the class where the method was originally implemented (implicitly preferred by JDeodorant), (ii) the class suggested by Methodbook, and (iii) a randomly selected class.

- 5 methods for which only JDeodorant suggests to move them from their original class to a new envied class (*onlyJDeodorant* group - OJ). For each method in this group the participants had three possible options in the questionnaire: (i) the original class (implicitly preferred by Methodbook), (ii) the class suggested by JDeodorant, and (iii) a randomly selected class.

- 5 methods for which both Methodbook and JDeodorant suggest to move them from their original class to the same new envied class (*bothSameClass* group - BS). For each method in this group the participants had three possible options in the questionnaire: (i) the original class, (ii) the class suggested by Methodbook and JDeodorant, and (iii) a randomly selected class.

- 5 methods for which both Methodbook and JDeodorant suggest to move them from their original class to two different new envied classes (*bothDifferentClasses* group - BD). For each method in this group the participants had four possible options in the questionnaire: (i) the original class, (ii) the class suggested by Methodbook, (iii) the class suggested by JDeodorant, and (iii) a randomly selected class.

A randomly selected class was considered only to verify whether participants seriously considered the given assignment (that is a sanity check). For each of the proposed class the subjects had to assign a score on a five-point Likert scale (126) from 1 (the class is

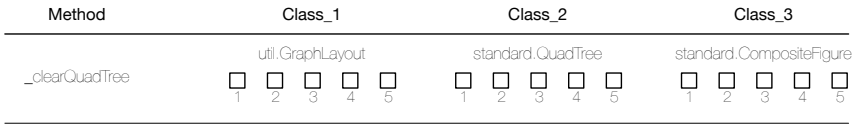

Figure 6.8: An example of question belonging to the *bothSameClass* group

not at all suitable to implement the analyzed method) to 5 (the class is very suitable to implement the analyzed method). Figure 7.3 shows an example of a question belonging to the *bothSameClass* group: *_clearQuadTree* is the method to place, *util.GraphLayout* is the random class, *standard.QuadTree* is the class suggested by Methodbook and JDeodorant, and *standard.CompositeFigure* is the original class.

Given the results of our first case study (see Section 6.3), for the *onlyMethodbook* group we selected 5 methods for which Methodbook suggests move method operations with the maximum confidence level, i.e., 1.0. On the contrary, for the groups *bothSameClass* and *bothDifferentClasses* we were not able to select Methodbook's suggestions having high confidence level. In fact, as for the *bothDifferentClasses* group there were only 5 methods for which Methodbook and JDeodorant suggest to move them from the original class in different envied classes (we selected all of them) and for no one of them Methodbook exhibits high confidence level (the average confidence level for these 5 methods is 0.4). Concerning the *bothSameClass* group, 6 were the methods for which Methodbook and JDeodorant suggest to move them from the original class in the same envied class (we randomly selected 5 out of 6 of them) and also in this case for no one of them the Methodbook's suggestions have high confidence level (average 0.5). The differences in the confidence level of the Methodbook's operations present in the questionnaire will allow us to further investigate about the goodnesses of the confidence level as indicator of the quality of the suggested refactoring operations.

Note that the subjects were not aware of the experimented techniques, i.e., Methodbook and JDeodorant, of the questionnaire structure nor of the different groups of questions. The 20 methods were removed from the JHotDraw system together with all the references to them in the source code. The removed methods were copied in 20 text documents and provided to the subjects, together with the questionnaire and the mutated version of JHotDraw (i.e., the one without the 20 removed methods), at the beginning of the experimentation.

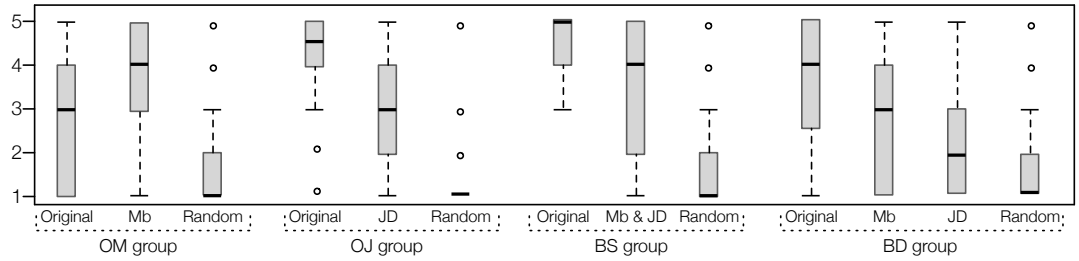

Figure 6.9: Box plots of the ratings provided by the 30 subjects

We analyzed the answers provided by the subjects through boxplots and statistical tests. As for the statistical tests, for each of the four groups of methods, we collected the ranking of classes in each of the different sets of proposed classes, i.e., original, suggested by Methodbook, suggested by JDeodorant, and random. Then, considering two particular sets, e.g., original *vs.* suggested packages, we used the t-test (122) to analyze the statistical significance of the difference between the ranking of classes in the two sets. The results were intended as statistically significant at $\alpha = 0.05$.

6.4.1.2 Analysis of the Results

Figure 6.9 shows the boxplots summarizing the answers provided by 30 subjects involved in our study while Table 6.2 reports the results of the t-test. All the data are presented by group of methods, i.e., OM, OG, BS, and BD.

Concerning the OM group, i.e., only Methodbook suggests to move the method from its original class to a new envied class, the boxplots in Figure 6.9 highlights that the subjects generally preferred the class suggested by Methodbook (median 4) to the original class (median 3). Moreover, the result of the t-test shows that the difference of preferences in favor of the class suggested by Methodbook is statistically significant (see Table 6.2). Note that, as explained before, in this scenario JDeodorant implicitly prefers the original class for the five methods under analysis.

In the OJ group, i.e., only JDeodorant suggests to move the method from its original class to a new envied class, the students tend to reward the original class (median 4.5) compared to the envied class suggested by JDeodorant (median 3). Also in this case, this difference in the subjects' preferences is classified as statistically significant by the

Table 6.2: Results of the T-test

Group	Test	α	Mean of differences
OM	original *vs* suggested	< 0.01	-0.89
	original *vs* random	< 0.01	+1.33
	suggested *vs* random	< 0.01	+2.21
OJ	original *vs* suggested	< 0.01	+1.09
	original *vs* random	< 0.01	+2.83
	suggested *vs* random	< 0.01	+1.74
BS	original *vs* suggested	< 0.01	+0.53
	original *vs* random	< 0.01	+2.46
	suggested *vs* random	< 0.01	+1.93
BD	original *vs* suggested Methodbook	< 0.01	+1.21
	original *vs* suggested JDeodorant	< 0.01	+1.57
	original *vs* random	< 0.01	+2.30
	suggested Methodbook *vs* suggested JDeodorant	0.02	+0.36
	suggested Methodbook *vs* random	< 0.01	+1.09
	suggested JDeodorant *vs* random	< 0.01	+0.73

t-test (see Table 6.2). It is worth noting that for this group of methods, Methodbook implicitly preferred the original class to the other classes of the system.

When both the approaches suggest to move a method from its original class to the same envied class, i.e., BS group, the original class is still preferred by the subjects (median 5) compared to the suggested class (median 4). Even for this comparison the t-test shows a statistically significant difference.

Finally, for the BD group, i.e., both the approaches suggest to move a method from its original class to two different envied classes, the students' preferences are also this time targeted to the original class (median 4) followed by the class suggested by Methodbook (median 3) and the class suggested by JDeodorant (median 2). Moreover, the t-test shows that (i) the scores assigned to the original class are statistically significant higher than those assigned to the suggestions by Methodbook as well as to

those by JDeodorant is statistically significant and (ii) also the scores assigned to the Methodbook's suggestions are statistically significant higher than those assigned to the JDeodorant's suggestions (see Table 6.2).

Note also that in all the analyzed groups of questions the suggested random classes achieved as median lo lowest possible (i.e., 1), confirming that the participants seriously considered the given assignment.

Summarizing, the results of this study highlight that:

- When Methodbook does not suggest any refactoring operation (OJ group) the subjects confirm that the original class is the best place for the method under analysis.

- The suggestions produced by Methodbook are generally preferred to those produced by JDeodorant. In particular: (i) in the OM group, JDeodorant implicitly prefer the original class, while the students rewarded the class suggested by Methodbook, (ii) in the OJ group, the original class (implicitly preferred by Methodbook) achieved the highest scores, and (iii) in the BD group, the class suggested by Methodbook is generally preferred to that suggested by JDeodorant (median 3 *vs* 2).

- The confidence level is confirmed as an important indicator of the goodnesses of the Methodbook's suggestions. In fact, when the suggestions proposed by Methodbook have high confidence level (OM group, with average confidence level equals to 1), the subjects generally prefer the class suggested by Methodbook to the original class. On the contrary, when Methodbook suggests a move method operation with low confidence level (groups BS with average confidence level equals to 0.5 and BD with average confidence level equals to 0.4), the subjects prefer the original class to the class suggested by Methodbook.

The performed analysis allow us to positively answer to our research question ($\mathbf{RQ}_3$): 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. In fact, even if Methodbook achieved good scores (median 4) in the BS group where the average confidence level is 0.6, the students in this scenario still preferred the original class (median 5).

Table 6.3: 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

6.4.2 Evaluation with Original Developers

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

6.4.2.1 Planning

In this study we executed both Methodbook and JDeodorant on the two subject systems, i.e., GESA and SMOS. Then, in order to answer our research question ($\mathbf{RQ}_3$), 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 5 people) that developed the system while for SMOS 5 out of 7 of the programmers that have worked to its development.

The subjects evaluated all the refactoring operations suggested by the two approaches 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). The number of refactoring operations suggested by two approaches (and thus, evaluated by the subjects) is reported in Table 6.3. As we can see the number is rather high for both the systems, and thus we gave four days to the subjects to evaluate all the refactoring operations. Each subject filled-in her two questionnaires, i.e., one with the Methodbook's suggestions and one with the JDeodorant's suggestions, independently. After that, all the subjects involved in the development of each system performed a review meeting to discuss their scores and reach a consensus. At the end of the meeting the subjects provided only two filled-in questionnaires (and thus, one for each system) reporting their comprehensive evaluation. We also asked the developers to comment on some particular cases.

Table 6.4: Subjects' answers to the question "*Would you apply the proposed refactoring?*"

		definitely not	*no*	*maybe*	*yes*	*absolutely*
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%**

6.4.2.2 Analysis of the Results

Table 6.4 summarizes the subjects' answers to the question "*Would you apply the proposed refactoring?*". Concerning the GESA software system the developers gave a positive answers to the 12% of operations suggested by JDeodorant (11% *yes* + 1% *absolutely*) against a 44% achieved by Methodbook (37% *yes* + 7% *absolutely*).

On the "negative answers side" the two approaches almost reached a tie with the 31% of JDeodorant's suggestions (24% *definitely not* + 7% *no*) and the 32% of Methodbook's suggestions (12% *definitely not* + 30% *no*) discarded by the subjects. Finally, there is a huge percentage (57%) of the 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 (0.22 with only one suggestion having confidence level higher than 0.6), the original developers generally preferred them to those generated by JDeodorant. In fact, while the percentage of move method suggestions discarded (through a *definitely not* or a *no* answer) is almost the same, the developers accepted (through a *yes* or a *absolutely* answer) a much higher percentage of Methodbook's suggestions than of the JDeodorant's ones.

The 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 operation. 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, in GESA a class is also present implementing all the operations needed to exchange data with the database, that is the class *ControlConnection*. Thus, the developers feel that the envied class identified by Methodbook is a better place to implement the method *executeOperation*.

On the other hand, ad example of Methodbook's suggestion that the subjects 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). 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 a quality metric point of view, but that is not meaningful from a developer point of view.

Concerning the SMOS software system, the developers accepted the 23% of operations suggested by JDeodorant (22% *yes* + 1% *absolutely*) against a 37% achieved by Methodbook (30% *yes* + 7% *absolutely*). As for the "rejected" refactoring operations, the 18% of the JDeodorant suggestions (6% *definetly not* + 12% *no*) and the 26% of the Methodbook suggestions (11% *definetly not* + 15% *no*) were classified as bad. Finally, also in this case a high percentage of JDeodorant suggestions were classified with *maybe* (59%) against a 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.

As for GESA, we asked the SMOS developers to comment for us some of their decisions. Figure 6.10 shows the method *classroomOnDeleteCascade* moved by Methodbook from the class *ManagerClassroom* to the class *ManagerRegister*. All the 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 from the database all the information related to it. As we can see, the information related to a classroom are mostly concerned with the class register, e.g., students absences. Thus,

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

  ...

  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);
  }
}
```

Figure 6.10: The method *classroomOnDeleteCascade* was moved by Methodbook from its class *ManagerClassroom* to the envied class *ManagerRegister*

the method *classroomOnDeleteCascade* invokes several times the class *ManagerRegister* that, for this reason, is identified by Methodbook as envied class.

On the other side, there is a group of 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 suggestions are generally not recommendable to apply.

Finally, we also asked both the 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 each control class to the corresponding entity object (e.g., move the method *getUserList()* from *ManagerUser* to *User*). 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 clearly separate entity and control objects in the systems. This set of move method operations suggested by JDeodorant also explain (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 6.3.

In conclusion, Methodbook suggestions were generally preferred to the JDeodorant ones (41% of accepted suggestions for Methodbook, 15% for JDeodorant) on both of the subject systems. However, the Methodbook suggestions also achieved a higher number of "rejected" suggestions compared to the JDeodorant ones (33% *vs* 27%). This is an expected result given the average low confidence level of the Methodbook suggestions on the two systems.

6.4.3 Threats to validity

This section describes some threats to validity (136) that may affect the results of our evaluations conducted with the users.

6.4.3.1 Evaluation with External Developers

As for the generalization of the results achieved in our experiment conducted with external developers, the population of subjects involved in the experimentation, i.e., bachelor students, represents the main threat. However, the subjects had acceptable analysis, development, and programming experience. In particular, in the context of the Software Engineering courses, bachelor students from both Universities had participated in software projects, where they practiced software development and documentation production. Moreover, as highlighted by Arisholm and Sjoberg (128) the difference between students and professionals is not always easy to identify.

Another threat is represented by the system domain knowledge of subjects. Before the controlled experiment execution meetings were organised with the aim of giving an acceptable system domain knowledge to the students and mitigating such a threat.

Moreover, we also executed the experiment with original developers where this threat is not present.

Also the number of refactoring operations (20) evaluated by the subjects is a possible threat to the generalization of the achieved results. We were not able to perform a complete evaluation of all the refactoring operations suggested by both approaches, i.e., Methodbook and JDeodorant, on JHotDraw for time constraints. In fact, for each of the analyzed methods, the subjects had to evaluate at least 3 (4 in the BD group) classes in which the method could be placed. Thus, 20 is the realistic number of refactoring operations that we could possibly evaluate in a user study lasting approximately for three hours (on average 9 minutes available to analyze each refactoring operation).

Finally, it is worth noting that in our evaluation we did not use any computer tool but preferred to provide a printed copy to the subjects, i.e., the questionnaires, of the move method refactoring operations identified by each approach since in this way we avoid confounding the results with how well subjects could use the tools interfaces.

6.4.3.2 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. Six of the subjects involved in this experimentation (three for each system) work in industry while among the remaining, one is a Ph.D. student, and three are master students. Thus, the kind of subjects involved in our two user studies is quite different, allowing a good generalization of the achieved results.

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 subjects 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 subjects provided an objective evaluation of the analyzed move method operations.

Finally, also in this experiment we avoided the use of tools preferring printed questionnaires to exclude biases derived by the subjects ability with the tools interfaces.

6.5 Final Remarks

In this Chapter is presented and evaluated Methodbook, an approach to automate Move Method refactoring. Methodbook uses RTM to analyze both structural and conceptual information gleaned from software to suggerst move method refactoring operations. Methodbook has been evaluated 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 quality of five software systems from a quality metrics point of view. The results indicate that the Methodbook's suggestions having high confidence level are able to significantly improve cohesion and coupling of the subject systems. Moreover, on three out of five experimented systems Methodbook uniformly outperforms JDeodorant, while on the remaining two systems the two approaches almost reached a tie.

In a 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 30 students on an open source software system. The results indicate that Methodbook provides meaningful recommendations for move method refactoring from a developer's point of view. In addition, the developers generally prefer Methodbook's recommendations compared to those produced by JDeodorant. The achieved results strongly support the potential usefulness of Methodbook in integrated development environments.

7

Move Class Refactoring

The material in this Chapter has been presented in (144).

7.1 Introduction

During maintenance, the structural design of the software system evolves and changes are not always performed following OO guidelines (3, 145). Indeed, software evolution is often driven by market forces that put pressure on stake-holders to reduce the time to market, which may lead to suboptimal design choices. One of the main reasons for such an architectural erosion is inconsistent placement of source code classes in software packages (82, 99). Such a scenario, on one hand negatively impacts the package cohesion and on the other hand increases the number of dependencies (coupling) between packages (100).

In such cases, re-modularization of the system is necessary (3, 101). Most of the existing approaches focus on proposing a whole new re-modularizations to the developer, i.e., they produce a completely new decomposition of classes in packages (e.g., (77, 79, 83)). The results of a totally new re-modularization might be difficult to interpret by software developers unless they provide explicit mapping (and explanation) to the original design. For this reason, this kind of re-modularization is preferable only when the structure of the system is too degraded and prevents the possibility of adopting focused and fine-grained refactoring operations (3), e.g., move a class between the existing packages. Focused refactoring operations have to be preferred when refactoring is systematically applied during software evolution. To this aim, we propose an

automated approach to support re-modularization through move class refactoring that takes into account the existing package structure and the content. The approach is an instantiation of the approach presented in Section 3.5 for moving misplaced code components to the move class refactoring.

The proposed approach analyzes underlying latent topics in classes and packages and uses structural dependencies to recommend refactoring operations aiming at moving classes to more suitable packages. In addition, the topics extracted from the classes and packages are used to identify their responsibilities and provide some rationale behind the proposed refactoring recommendation, e.g., the class *ActionExportProfileXMI* is very relevant to the topic *[profile, model, url]* and should be moved into package *org.argouml.profile*, which is described by the topic *[profile, ocl, model]*. As for our move method approach described in Chapter 6, the topics are acquired via Relational Topic Models (RTM) (62) (see Section 3.5 for details). RTM is used as an underlying solution to analyze conceptual (that is, topics in classes and packages) and structural (that is, dependencies) information to recommend refactoring solutions. The resulting tool, coined as *R*3 (Rational Refactoring via RTM), has been evaluated in two empirical studies. In the first study we analyzed the ability of *R*3 to propose refactoring operations that lead to reduced coupling among software modules in nine software systems. However, refactoring operations should not only improve the quality of a software system in terms of metrics, but, most importantly, should be meaningful from a developer's point of view. This observation calls for our second study, where we evaluated *R*3 refactoring recommendations with developers in two case studies, one conducted with 14 original developers of four software systems and one with 44 students and academics plus 4 professional software developers on another open source software system.

Section 7.2 presents the details behind *R*3. Section 7.3 reports the first case study where *R*3 has been evaluated via quality measures, while Section 7.4 reports the results of the study with users.

7.2 *R3*: Rational Refactoring via RTM

We propose an approach, namely *R*3, that automatically analyzes the underlying latent topics inferred from identifiers, comments, and string literals in the source code classes as well as structural dependencies among these classes. Using the results of the analysis

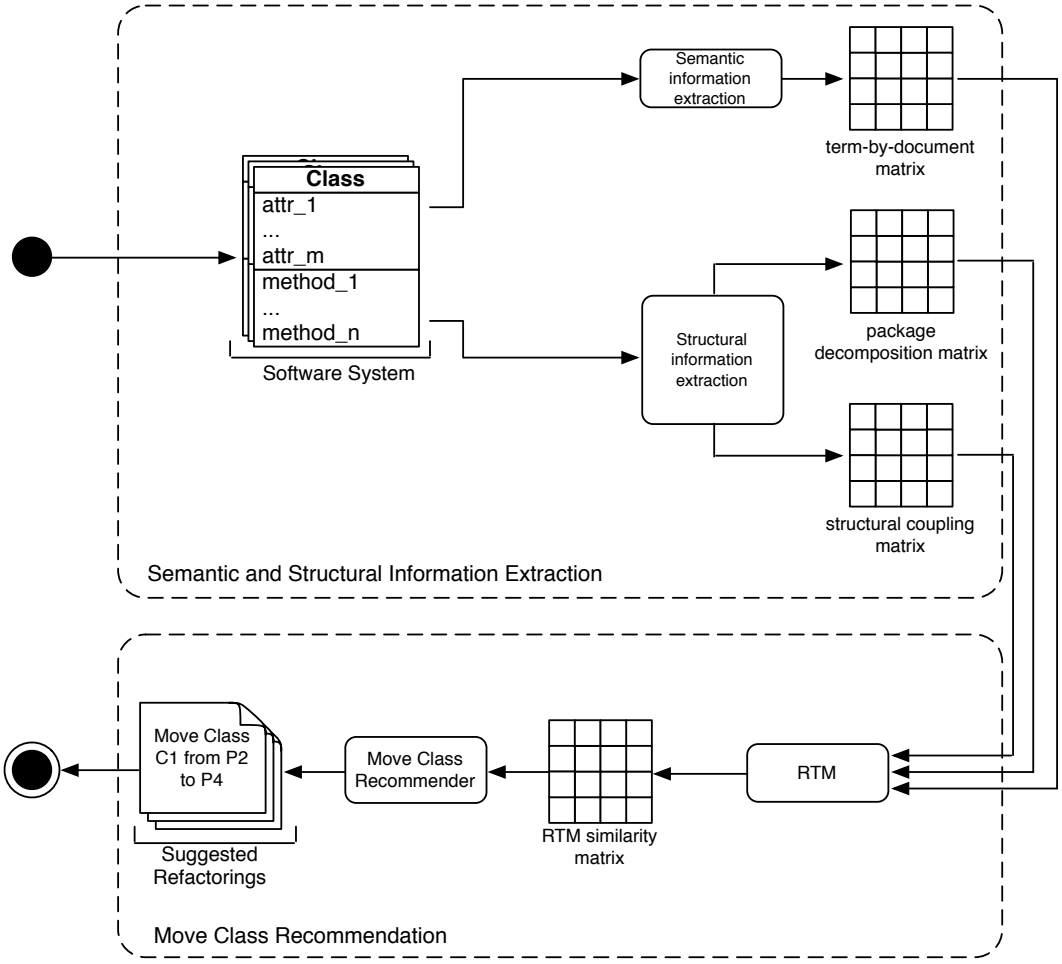

Figure 7.1: Identifying move class refactoring with $R3$.

we are able to identify possible move class refactoring opportunities (i.e., identify more suitable packages for relocating a class under analysis). The integrated analysis of structural and semantic information, as modeled by $R3$ allows us to analyze the quality of software packages both from a conceptual (that is, responsibilities implemented in classes in different packages) and structural (that is, dependencies among classes in a package and among other packages) points of view.

In a nutshell, $R3$ works as depicted in Figure 7.1. Semantic information (identifiers, comments, and string literals) is extracted from source code classes and stored in a *term-by-document* matrix. The *term-by-document* matrix is required by RTM to derive semantic relationships between classes and define a probability distribution of

topics (topic distribution model) among classes. Besides semantic information, *R*3 also exploits static analysis to (i) derive dependencies among classes (stored in the *structural coupling matrix*) and (ii) the existing package composition (stored in the *package decomposition matrix*). These two matrices are used to adjust the probability distribution taking into account structural relationships between classes, besides semantic information. In particular, the *structural coupling matrix* is employed to provide RTM with information concerning the dependencies (i.e., calls) between classes (that is the main information used for software modularization). The *package decomposition matrix* is used in the context of a fine-grained re-modularization to take into account the design decisions made by the developers. Providing RTM with information on the original design induces the technique to suggest a move class refactoring operation only if it results in a clear improvement of the design quality.

The model derived by RTM is then used to compute similarities among classes based on both probabilistic distributions of latent topics and underlying dependencies. After obtaining similarities among all the classes for a given system (*RTM similarity matrix* in Figure 7.1), for each class the approach identifies a set of highly similar classes (that is, classes sharing similar topics and/or having structural relationships). The set of identified classes is then used to determine refactoring operations aiming at moving the class into a package that contains the higher number of similar classes. Clearly, if the identified package coincides with the original package, no refactoring is required.

As it can be seen, the approach is completely automated; once the refactoring operations are identified, they can be applied to the software system obtaining a new modularization. The new modularization should have a better quality in terms of cohesion and coupling. However, design decisions are oftentimes more intricate and delicate than just trying to minimize coupling and maximize cohesion. As a result, the proposed recommendations should be analyzed by developers who can accept or reject proposed move class refactoring operations or make alternative decisions based on underlying recommendations and analysis information. Unfortunately, without a deep knowledge of the complete system, it may be difficult to reach an agreement on which refactoring should (not) be applied. The proposed approach aims at mitigating such a problem. Indeed, one unique characteristic that distinguishes *R*3 from all the other refactoring approaches is its ability to generate an evaluation (based on quantitative analysis) and explanation (based on qualitative analysis) for the refactoring recommendations.

7.2.1 Semantic and Structural Information Extraction

One key prerequisite for generating refactoring recommendations using *R3* is the semantic and structural information that should be extracted and analyzed. As the very first step, classes are analyzed to extract words contained in comments, identifiers, and string literals. In order to extract the single words advanced algorithms for splitting identifiers are employed (140). 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 classes, and n is the number of classes in the system (see Figure 7.1). 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 (106).

A light-weight static analysis is also applied to the current release of the software system to detect (i) dependencies between classes (i.e., method calls) and (ii) existing package decomposition. The latter is a simple boolean $n \times n$ matrix (called *package decomposition matrix*), where n is the number of classes composing the software system to re-modularize. A generic entry $o_{i,j}$ of this matrix equals to 1 if the class C_i and the class C_j are grouped in the same package in the original modularization, otherwise it is equal to 0. Concerning the dependencies among the classes of the system, we capture them using the Information-Flow-based Coupling (ICP) (108) and store this information in another $n \times n$ matrix (called *structural coupling matrix*). ICP measures the amount of information flowing into and out of a class via parameters through method invocation, i.e., the measure sums the number of parameters passed at each method invocation. The interested reader can find its definition in Section 3.2.

7.2.2 Computing the RTM Similarity Matrix

The three computed matrices, i.e., *term-by-document matrix*, *package decomposition matrix*, and *structural coupling matrix*, are supplied to RTM to generate a topic distribution model (see Figure 7.1). As explained in Section 3.5, the peculiarity of RTM as compared to other topic modeling techniques is in its ability to adjust the probability distribution of each topic taking into account explicit relationships among the documents. In our approach, explicit relationships among the documents (classes) are

modeled through dependencies among classes and original design (stored in the *calls interaction matrix* and *original design matrix*, respectively).

The enriched topic distribution model (based on both semantic and structural information) obtained by RTM is used to compute similarities among all the classes of the system. Such similarities are stored in a $n \times n$ matrix, namely *RTM similarity matrix*, that is employed to identify move class refactoring operations (see Figure 7.1).

7.2.3 Identifying Move Class Refactoring Opportunities

*R*3 uses the *RTM similarity matrix* to determine the degree of similarity among classes in the system and identify classes similar to a given class candidate for move class refactoring. A cut point then is used to detect the μ most similar classes. We tried several values for μ and the best results were achieved setting $\mu = 5$. *R*3 then analyzes these classes and the packages containing them to identify the best target package for a given class. In our current implementation, target package is the one that contains the highest number of most similar classes. Note that more sophisticated criteria can be used to select the best target package for a class under analysis, given the list of similar classes. However, we experimented with different possible solutions by manually analyzing resulting refactoring suggestions and found no significant differences between the simple, adopted, solution and more sophisticated heuristics. Moreover, the adopted solution is justified by the observation that the higher the number of related classes in the package, the higher the quality of the package in terms of cohesion and coupling metrics (3). Finally, in those cases where two or more packages contain the same number of similar classes, the target package is the one that contains the highest ranked similar class.

The following example illustrates the process of identifying the target package for the class *org.argouml.ui.explorer.ActionExportProfileXMI* that represents a well-know design problem in ArgoUML 0.16[1]. Given the textual information extracted from this class as well as a list of other classes, which are structurally connected to *ActionExportProfileXMI*, *R*3 recommends a more appropriate package where the class should be moved. RTM-based analysis reveals that the topic *"profiles"* is the dominant topic in *ActionExportProfileXMI*. Additionally, the package, which *ActionExportProfileXMI* is most structurally dependent on is *org.argouml.profile*. That is, strong structural

[1]http://argouml.tigris.org/

dependencies exist between the class being considered and the classes *Profile* and *ProfileException*, which are implemented in *org.argouml.profile* package. After supplying these dependencies into RTM, *R*3 discovers that the top five similar classes include all the classes belonging to the package *org.argouml.profile*, i.e., *StreamModelLoader.java, ProfileManager.java, CoreProfileReference.java, ResourceModelLoader.java,* and *FileModeLoader.java*. This means that for *R*3, the class *ActionExportProfileXMI* should be placed in the package *org.argouml.profile*.

Although the version 0.16 of ArgoUML implements it in the package *org.argouml.ui.explorer*, evidence suggests that it should actually be moved to the package *org.argouml.profile*. After moving the package we observe a noticeable decrement in coupling. The descriptions of the class and packages, which appear in the Javadocs[1], also support the recommendation by *R*3. The external documentation summarizes the package *org.argouml.ui.explorer* as follows, *"contains classes for the explorer tree view of argouml."* The package *org.argouml.profile* is said to *"Contains support for UML profiles"* while the class *ActionExportProfileXMI "Exports the model of a selected profile as XMI"*. The Javadocs also suggest that the package *org.argouml.profile* may be a more appropriate place to implement the class. This example illustrates the strength of *R*3 to make suggestions that both improve software quality from the perspective of structural and conceptual metrics.

7.2.4 Putting Software Developers into the Loop

While *R*3 is a completely automated approach, it is designed to serve as a refactoring assistant for software developers. The approach can take as an input a class or a set of classes that may be candidates for move class refactoring. A specific class may be supplied as an input to *R*3 to identify if there are any other more suitable packages for this class. Alternatively, the whole system can be used as an input to *R*3 resulting in a set of recommendations about possible move class refactoring opportunities.

To facilitate software developer's task of accepting or rejecting a suggested move class refactoring operation, *R*3 provides an evaluation and an explanation behind the recommended refactoring operation (see Figure 7.2). This evaluation is provided in the form of a confidence level, while the explanation is based on qualitative data extracted via topic analysis.

[1]http://argouml-stats.tigris.org/nonav/javadocs/javadocs-0.32/

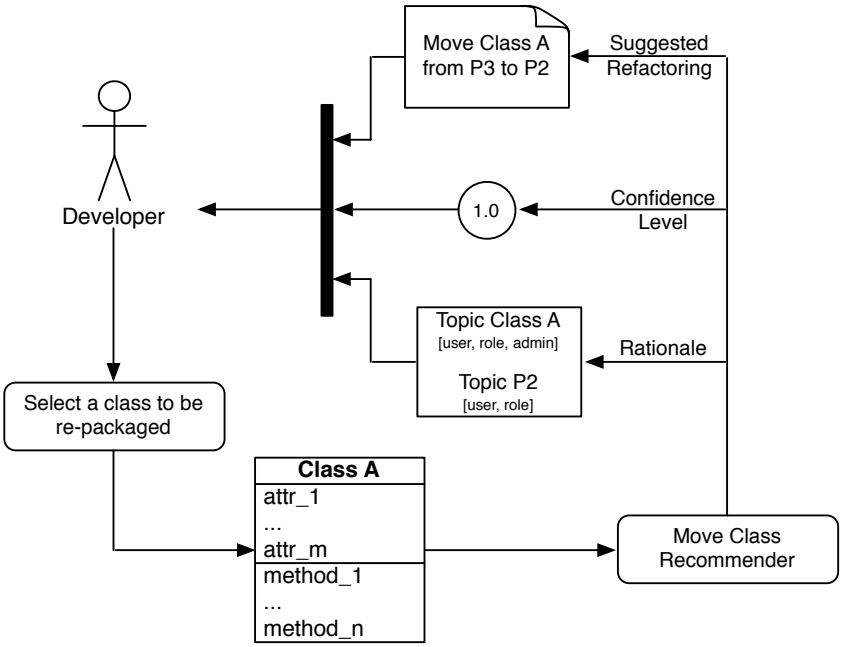

Figure 7.2: Interaction between $R3$ and the software engineer.

For the computation of the confidence level, we employ information entropy to analyze distributions of μ similar classes across different packages and quantify the confidence of the proposed refactoring recommendation. We consider the most similar classes as an outcome of a random variable X. For a random variable X with μ outcomes $\{x_i : i = 1, \ldots, \mu\}$ the Shannon information entropy, a measure of uncertainty, is defined as:

$$H(X) = \sum_{i=1}^{\mu} p(x_i) \frac{1}{log_{\mu}(x_i)}$$

where $p(x_i)$ is the probability value of outcome x_i. Note that, as defined, $H(X)$ can assume values in [0,1]. Thus, the confidence level for the suggested package is defined as follows:

$$\text{confidenceLevel} = 1 - H(X)$$

That is, the more scattered similar classes among the packages, the higher the entropy of the suggestion of the target package (the confidence is low, since we have many candidate packages). On the other hand, if all the similar classes are implemented in a single package, the entropy of this suggestion is low (the confidence is high, since we have

one or a few target packages). Consider the example where we want to move the class *ActionExportProfileXMI*. In this case the top five similar classes include all classes belonging from the same package, *org.argouml.profile*. Thus, the suggestion has the lowet uncertainty ($H(X) = 0$) and, consequently, the highest confidence ($confidenceLevel = 1$).

As for the explanation of the suggested refactoring, $R3$ analyzes and presents the topics for a given class as well as topics for packages suggested as target packages for refactoring operation. Conceptual overlap between a class candidate to be moved and a suggested target package in terms of underlying latent topics (generated by RTM) serves as a good indication for the rationale behind the proposed refactoring. Starting from the extracted topics, the explanations provided by $R3$ are in the following form:

MOVE class C implementing the topics $[T_1, \ldots, T_n]$
FROM its package P_i grouping the topics $[T_1, \ldots, T_m]$
TO the package P_j grouping the topics $[T_1, \ldots, T_k]$

where C is the class to be moved, P_i is the original package, P_j is the target package, and T_i is a topic composed by a set of words.

We use the same example from ArgoUML to illustrate how this feature of $R3$ works in a real scenario. Our running example focuses on identifying the appropriate package to implement the class *ActionExportProfileXMI*. For each class and package within a software system $R3$ identifies relevant topics based on the analysis of textual and structural information, which was provided as an input. As previously mentioned, each class has a probability of being associated with every topic extracted. We use the key words from the topic with the highest probability to provide additional insight into the suggestions. *ActionExportProfileXMI*'s most significant topic is *[profile, model, url]*. Likewise, for each package in a software system, our approach also identifies the most prevalent topics. The packages *org.argouml.profile* and *org.argouml.ui.explorer*, which were discussed in Section 7.2.3, are best described by the topics *[profile, ocl, model]* and *[tree, node, explor]*, respectively. Thus, in this case the $R3$'s explanation will be:

MOVE class *ActionExportProfileXMI* implementing the topics *[profile, model, url]*
FROM its package *org.argouml.ui.explorer* grouping the topics *[tree, node, explor]*

TO the package *org.argouml.profile* grouping the topics *[profile, ocl, model]*

Based on the topic analysis, implementing the class *ActionExportProfileXMI* in the package *org.argouml.profile* appears to be a better option than implementing it in the package *org.argouml.ui.explorer*. These findings support the recommendation made by *R*3.

7.3 Software Metrics Evaluation

One widely accepted rule to increase the maintainability of software systems is to pursue low coupling among the software modules (146, 147, 148). The *goal* of our first case study is to (i) verify whether the move class operations suggested by *R*3 are able to reduce the coupling among the packages of an OO software system and (ii) analyze the relationship between the confidence level and the changes in terms of coupling.

The subjects of our study are nine software systems. Four of them, namely GanttProject[1], jEdit[2], JHotDraw[3], and jVLT[4], are open-source projects, two are industrial projects, namely eXVantage[5], GESA[6], and three, eTour, SESA, and SMOS, have been developed by a team of Master students of the University of Salerno during the Software Engineering course. Table 7.1 reports the size, in terms of KLOC, number of classes, and number of packages, and the versions of the systems. Moreover, Table 7.1 reports the average (structural and semantic) coupling between the packages of each system. We measured the structural coupling between two packages P_i and P_j as:

$$StructuralCoupling(P_i, P_j) = \frac{\sum_{l=1}^{|P_i|} \sum_{s=1}^{|P_j|} MPC(C_l, C_s)}{|P_i| \times |P_j|}$$

where $C_l \in P_i$, $C_s \in P_j$, and $MPC(C_l, C_s)$ is the Message Passing Coupling (MPC) (109) between C_l and C_s. MPC is a 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

[1]http://www.ganttproject.biz/
[2]http://www.jedit.org/
[3]http://www.jhotdraw.org/
[4]http://jvlt.sourceforge.net/
[5]http://www.research.avayalabs.com/
[6]http://www.distat.unimol.it/gesa/

Table 7.1: Software systems used in the case study

System	KLOC	Classes	Packages	*StructuralCoupling*			*SemanticCoupling*		
				Mean	Median	St. Dev.	Mean	Median	St. Dev.
eTour 1.0.1	30	134	17	0.105	0.02	0.155	0.261	0.227	0.105
eXVantage 2.01	36	352	85	0.045	0.008	0.363	0.202	0.141	0.204
GanttProject 1.10.2	28	273	27	0.036	0.009	0.113	0.136	0.105	0.098
GESA 2.2	46	295	22	0.097	0.002	0.108	0.364	0.332	0.087
jEdit 4.4	101	537	29	0.011	0.006	0.040	0.177	0.191	0.106
JHotDraw 6.0 b1	29	275	12	0.096	0.001	0.279	0.089	0.075	0.068
jVLT 1.3.2	24	214	23	0.067	0.012	0.221	0.127	0.142	0.041
SESA 1.4	11	128	14	0.019	0.003	0.092	0.463	0.429	0.215
SMOS 1.0	23	121	12	0.082	0.010	0.119	0.273	0.301	0.128
Total	328	2,329	241	-	-	-	-	-	-

implemented by other classes. It has been demonstrated that the MPC directly correlates with the maintenance effort (109). Thus, higher MPC values (higher coupling) indicate higher effort in maintaining a software system.

As for the semantic coupling, we measure it between two packages P_i and P_j as:

$$SemanticCoupling(P_i, P_j) = \frac{\sum_{l=1}^{|P_i|} \sum_{s=1}^{|P_j|} CCBC(C_l, C_s)}{|P_i| \times |P_j|}$$

where $C_l \in P_i$, $C_s \in P_j$, and $CCBC(C_l, C_s)$ is the Conceptual Coupling Between Classes (CCBC) (55) C_l and C_s. CCBC is based on the semantic information (i.e., domain semantics) captured in the code by comments and identifiers. Two classes are conceptually related if their (domain) semantics are similar, i.e. they have similar responsibilities. Higher CCBC values indicate higher coupling. Note that the 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 managable maintenance activities.

7.3.1 Study Design

We used $R3$ to suggest a package for all the classes in the subject software systems. Thus, we applied $R3$ on a total of 2,329 classes. Then, we identified the move class refactoring operations suggested by $R3$ comparing the suggested package of each class with its original package. If the suggested package is different from the original package

Table 7.2: Possible values for the $R3$ confidence level.

Value	Five most similar classes ($C_1 \dots C_5$) distribution among packages	Probability distribution
0.00	$C_1 \in P_1$ and $C_2 \in P_2$ and $C_3 \in P_3$ and $C_4 \in P_4$ and $C_5 \in P_5$	$\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}$
0.17	$C_1, C_2 \in P_1$ and $C_3 \in P_3$ and $C_4 \in P_4$ and $C_5 \in P_5$	$\frac{2}{5}, \frac{0}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}$
0.34	$C_1, C_2 \in P_1$ and $C_3, C_4 \in P_3$ and $C_5 \in P_5$	$\frac{2}{5}, \frac{0}{5}, \frac{2}{5}, \frac{0}{5}, \frac{1}{5}$
0.41	$C_1, C_2, C_3 \in P_1$ and $C_4 \in P_4$ and $C_5 \in P_5$	$\frac{3}{5}, \frac{0}{5}, \frac{0}{5}, \frac{1}{5}, \frac{1}{5}$
0.58	$C_1, C_2, C_3 \in P_1$ and $C_4, C_5 \in P_4$	$\frac{3}{5}, \frac{0}{5}, \frac{0}{5}, \frac{2}{5}, \frac{0}{5}$
0.69	$C_1, C_2, C_3, C_4 \in P_1$ and $C_5 \in P_5$	$\frac{4}{5}, \frac{0}{5}, \frac{0}{5}, \frac{0}{5}, \frac{1}{5}$
1.00	$C_1, C_2, C_3, C_4, C_5 \in P_1$	$\frac{5}{5}, \frac{0}{5}, \frac{0}{5}, \frac{0}{5}, \frac{0}{5}$

this means that $R3$ suggests a move class refactoring. To evaluate the coupling changes achieved by instantiating the recommended refactoring operations we applied them incrementally starting from those having the higher confidence level (see Section 7.2). After each performed refactoring operation we measured the average (structural and semantic) coupling between the packages of the system as defined above. In this way we were able to observe if performed refactoring operations were able to reduce the average package coupling for a given system. Moreover, the order of the refactoring operations (refactoring from those having the higher confidence level to those having low confidence level) allows to easily analyze if there is a correlation between the confidence level of the suggested refactoring operations and increase/decrease in coupling in the system. In particular, if the confidence level is a good indicator for the goodness of $R3$ recommendations, we expect to observe higher decrease in average package coupling for higher confidence levels of a refactoring operation (and *viceversa*). Note that since $R3$ considers the 5 most similar classes of a class C to identify the best package for C, we can obtain as confidence level one of the 7 possible values reported in Table 7.2. For example if all the top 5 most similar classes belonging to different packages, the entropy will be 1 and thus, the confidence level will be 0. On the contrary, if all the top 5 most similar classes belonging to the same package, the entropy will be 0 and thus, the confidence level will be 1.

Table 7.3: Percentage agreement between packages suggested by $R3$ and original design.

System	% Agreement	Confidence level distribution						
		1.00	0.69	0.58	0.41	0.34	0.17	0.00
eTour	62%	63%	1%	18%	12%	5%	0%	1%
eXVantage	55%	75%	9%	5%	4%	4%	3%	0%
GanttProject	70%	62%	24%	6%	4%	2%	2%	0%
GESA	55%	92%	4%	4%	0%	0%	0%	0%
jEdit	51%	71%	14%	4%	4%	4%	2%	1%
JHotDraw	52%	46%	15%	15%	11%	8%	5%	0%
jVLT	30%	49%	13%	10%	9%	8%	8%	3%
SESA	26%	53%	10%	20%	7%	7%	0%	3%
SMOS	68%	72%	8%	8%	7%	5%	0%	0%
Average	52%	65%	11%	10%	6%	5%	2%	1%

7.3.2 Experiment results

In this section we analyze the results obtained in the case study.

Table 7.3 reports the percentage of agreement between the original design of each subject system and the suggested package provided by $R3$ as well as the distribution of the confidence level in these cases. As we can see, $R3$ suggests the original package on average for 52% of the classes. Moreover, it is worth noting that generally when there is an agreement between $R3$ and the original design, the $R3$'s suggestions are generally provided with a high confidence level (86% have a confidence level ≥ 0.58).

The remaining 48% of classes that are placed in different packages than the original ones represent our disagreement scenario, i.e., the suggested move class refactoring operations. Table 7.4 shows the changes in terms of structural and semantic coupling achieved while applying move class operations suggested by $R3$. Analyzing the JHotDraw system it is possible to observe that by applying only the 9 move class operations having confidence level of 1 it is possible to achieve a reduction in the average structural coupling in the system by 86% and of the average semantic coupling by 41%. A reduction of coupling is still achieved when applying move class refactoring operations with confidence levels of 0.69 and 0.58 (globally, -3% for the $StructuralCoupling_{avg}$ and -7% for the $SemanticCoupling_{avg}$), while when applying the move class operations having confidence level lower than 0.58 we achieve an increase of the average coupling

Table 7.4: Coupling improvement while applying move class refactoring operations suggested by $R3$

System	Confidence level													
	1.00		0.69		0.58		0.41		0.34		0.17		0.00	
	StC	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*
eTour	-6%	-3%	-11%	-7%	-5%	-3%	-40%	-44%	-41%	22%	0%	0%	n.a.	n.a.
eXVantage	-50%	-48%	-6%	-7%	-4%	-24%	+1%	0%	-3%	-15%	+39%	+16%	+44%	+9%
GanttProject	-36%	-10%	-9%	-6%	+4%	-2%	-12%	-8%	+1%	-3%	-5%	+9%	+7%	+1%
GESA	-25%	-27%	-14%	-33%	-37%	-50%	+8%	+18%	0%	-7%	-4%	0%	n.a.	n.a.
jEdit	-21%	-4%	-9%	-8%	-15%	0%	+73%	+15%	+2%	-15%	+8%	-1%	+2%	0%
JHotDraw	-86%	-41%	0%	-7%	-3%	0%	+16%	+18%	+9%	0%	+10%	+4%	+1%	+1%
jVLT	-22%	-5%	-3%	-2%	-2%	-4%	-5%	-3%	+48%	+18%	+60%	+16%	+1%	+5%
SESA	-3%	-1%	-14%	-2%	-6%	-12%	+67%	-6%	+22%	-2%	+6%	+1%	0%	0%
SMOS	n.a.	n.a.	-16%	-6%	0%	-3%	+69%	-3%	+3%	+1%	-1%	0%	+5%	0%
Average	-31%	-17%	-9%	-9%	-8%	-11%	+20%	+5%	+5%	0%	+13%	+5%	+7%	+2%

$StC = \delta StructuralCoupling_{avg}$, $SeC = \delta SemanticCoupling_{avg}$

On eTour and GESA no move class refactoring operations have been proposed with confidence level equal to 0.0

On SMOS no move class refactoring operations have been proposed with confidence level equal to 1.0

of the system. Note that this trend is confirmed for all the object systems (see Table 7.4).

We also analyzed the average improvement provided by the single refactoring operations at different confidence levels to further investigate different effects of move class operations having different confidence levels. Table 7.5 reports the achieved results. As we can see on average each move class operation having the highest confidence level reduces the $StructuralCoupling_{avg}$ by 2.7% and the $SemanticCoupling_{avg}$ by 1.2%. From the data in Table 7.5 it is also possible to observe that applying move class operations having confidence level higher or equal to 0.58 we are generally able to reduce the coupling between the packages, while move class operations having confidence level lower than 0.58 generally results in an increase of coupling.

The obtained results demonstrate that $R3$ is able to reduce the coupling between software modules for a given software system by recommending useful move class refactoring operations. However, this empirical observation holds only when the confidence level for suggested operations is higher than 0.58, thus highlighting the goodness of the confidence level as an indicator of the quality of $R3$ recommendations.

7.3.3 Threats to validity

In this section we analyze the main threats that could affect the findings of our first case study.

Table 7.5: Average coupling improvement for move class refactoring operations at different confidence levels.

System	Confidence level													
	1.00		0.69		0.58		0.41		0.34		0.17		0.00	
	StC	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*
eTour	-1.9%	-1.1%	-2.6%	-1.8%	-0.4%	-0.3%	-1.2%	-1.3%	-6.7%	-3.6%	0%	0%	n.a.	n.a.
eXVantage	-1.8%	-1.7%	-0.1%	-0.1%	-0.2%	-1.2%	+0.1%	0.0%	-0.2%	-1.1%	+4.3%	+0.9%	+4.4%	+0.9%
GanttProject	-4.0%	-1.2%	-0.3%	-0.2%	+0.6%	-0.1%	-0.7%	-0.5%	+0.3%	-0.7%	-0.5%	+1.7%	-0.6%	+0.7%
GESA	-0.3%	-0.3%	-1.9%	-4.7%	-3.1%	-4.1%	+8.2%	+18.2%	-0.4%	-0.7%	-0.9%	0%	n.a.	n.a.
jEdit	-0.5%	-0.1%	-0.2%	-0.2%	-0.4%	0.0%	+2.3%	+0.2%	+0.1%	-1.0%	+1.3%	-0.1%	+2.2%	+0.3%
JHotDraw	-9.6%	-4.6%	0.0%	-0.2%	-0.1%	0.0%	+0.9%	+1.4%	+0.9%	0.0%	+0.5%	+0.2%	+0.3%	+0.2%
jVLT	-1.8%	-0.5%	-0.2%	-0.1%	-0.2%	-0.3%	-0.2%	-0.1%	+1.2%	+0.4%	+1.7%	+0.4%	+0.2%	+1.1%
SESA	-1.3%	-0.4%	-1.7%	-0.2%	-0.1%	-0.3%	+3.5%	-0.3%	+11.1%	-0.8%	+0.6%	+0.1%	0%	0%
SMOS	n.a.	n.a.	-1.7%	-0.7%	0%	-1.3%	+5.7%	-0.3%	+0.4%	+0.2%	-0.3%	0%	+0.7%	0%
Average	-2.7%	-1.2%	-1.0%	-0.9%	-0.4%	-0.8%	+2.0%	+2.0%	+0.7%	-0.8%	+0.7%	+0.4%	+1.0%	+0.4%

$StC = \delta StructuralCoupling_{avg}$, $SeC = \delta SemanticCoupling_{avg}$

On eTour and GESA no move class refactoring operations have been proposed with confidence level equal to 0.0

On SMOS no move class refactoring operations have been proposed with confidence level equal to 1.0

7.3.3.1 Employed quality metrics

In our study we measured the increase/decrease in coupling provided by the move class operations suggested by *R*3 using the average structural and semantic coupling of the packages. To measure these types of coupling we employed two well-established quality metrics, i.e., $CCBC$ on the semantic side and MPC on the structural side. Unlike other previous work (see e.g. (18, 127)), we have intentionally chosen quality metrics that are not exploited by *R*3 to suggest move class operations (*R*3 analyzes topics via RTM on the semantic side and ICP on the structural side). However, as in all the software metrics evaluations, there is a risk that the improvement achieved by applying the proposed remodularization is obtained by construction. In fact (i) both MPC and ICP, even if in a different way, are based on calls interaction between the classes of the system and (ii) CCBC and RTM exploit the same information, i.e., terms in comments, identifiers, and string literals of the classes, to capture overlap of semantic concepts between classes. Thus, even if a software metric evaluation is needed to verify that a new re-modularization approach does not negatively affect the coupling, this kind of evaluation cannot be central in the experimentation of a new technique (as done in several previous papers (17, 18, 82, 84, 127)). Indeed, different approaches provide different re-modularizations of a software system that reduce coupling. So, besides achieving a reduction of coupling it is necessary to show that a suggested re-modularization is meaningful from a developer's point of view. This is the reason why we performed the user studies, with a total of 62 developers, reported in Section 6.4.

Table 7.6: Average structural and semantic cohesion trend applying move class operations suggested by $R3$

System	Confidence level													
	1.00		0.69		0.58		0.41		0.34		0.17		0.00	
	StC	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*	*StC*	*SeC*
eTour	+37%	+7%	+4%	0%	0%	+1%	+43%	+17%	0%	0%	+10%	+6%	n.a.	n.a.
eXVantage	+12%	+5%	+11%	0%	+8%	+4%	-9%	0%	0%	+2%	+5%	+4%	0%	0%
GanttProject	0%	+5%	+12%	0%	+4%	+8%	0%	+1%	+1%	+2%	-22%	+7%	-1%	0%
GESA	+140%	+34%	0%	+5%	-10%	+10%	-5%	0%	-10%	0%	-42%	0%	n.a.	n.a.
jEdit	+7%	+8%	+17%	+2%	0%	-2%	-3%	-2%	+2%	+4%	+6%	0%	0%	0%
JHotDraw	+1%	0%	0%	-1%	-3%	-1%	0%	-7%	+10%	+17%	0%	0%	0%	0%
jVLT	+17%	+1%	+18%	+1%	-10%	-7%	-36%	+2%	+51%	0%	-4%	+15%	0%	0%
SESA	+5%	+1%	+7%	0%	+75%	+1%	-71%	-3%	-23%	+2%	+25%	-3%	-15%	0%
SMOS	n.a.	n.a.	+1%	+7%	+8%	+0%	-33%	-12%	-4%	+1%	+24%	+12%	-31%	-3%
Average	+27%	+8%	+8%	+2%	+8%	+2%	-13%	0%	+3%	+3%	0%	+5%	-6%	0%

$StC = \delta StructuralCohesion_{avg}$, $SeC = \delta SemanticCohesion_{avg}$

On eTour and GESA no move class refactoring operations have been proposed with confidence level equal to 0.0

On SMOS no move class refactoring operations have been proposed with confidence level equal to 1.0

7.3.3.2 Package cohesion

We evaluated move class refactorings suggested by $R3$ only from the coupling point of view. Even if low coupling among the software modules is one of the main goal for a good modularization (146, 147, 148), there is a risk that $R3$ might move a class into an unrelated package, i.e., the package that groups many unrelated responsibilities with the only goal of reducing the coupling between packages. To mitigate this threat we also measured the changes in terms of average (structural and semantic) cohesion of the packages in the studied systems. To measure the average structural and semantic cohesion we exploited the same metrics used for the coupling, i.e., CCBC and MPC. We measured the structural cohesion of a package P_i as the average MPC between all the possible couples of classes in P_i and the semantic cohesion of a package P_i as the average CCBC between all the possible couples of classes in P_i. Table 7.6 reports the achieved results showing that, besides strongly decreasing coupling between packages, $R3$ is also able to improve their cohesion for the move class refactoring operations having high confidence level, i.e., higher or equal to 0.58. In the low confidence level scenario, i.e., lower than 0.58, the cohesion of the packages does not show a stable trend, i.e., sometimes the cohesion increases and sometimes it decreases.

7.4 Evaluating *R3* with software developers

In our previous case study (Section 6.3) we evaluated recommended move class refactoring operations by analyzing the difference in terms of quality metrics between pre- and post-refactoring. However, the refactoring operations should not only improve the quality of a software system in terms of metrics, but should also be meaningful from a developer's point of view. For this reason, we performed two studies involving software developers. The first study was conducted on JHotDraw and involved 48 developers, i.e., 29 CS Master students from the University of Salerno, 7 CS Masters, 8 Ph.D. students and faculty members from the College of William and Mary, and 4 industry practitioners from elsewhere. Since the participants of this first study did not participate in the development of JHotDraw, we refer to them as "external developers". The second study was conducted on eTour, GESA, SESA and SMOS with the original developers of the subject systems. In particular, we were able to involve 14 original developers in this study (i.e., 5 for GESA, 5 for SMOS, 2 for eTour, and 2 for SESA). It was necessary to perform both these studies to have a complete evaluation of $R3$. Indeed, the only study with external developers may not be enough since they do not have a deep knowledge of the design of the subject system under analysis. They may not be aware of some of the design choices that could appear as suboptimal, but that are the results of a rational 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 them to the original design, they could be the "fathers" of some bad design choices and consequently could not recognize good move class recommendations as meaningful as suggested by $R3$. 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 and usefulness of the recommendations suggested by $R3$ from different points of view.

In the context of the two studies, the following research questions were formulated:

- $\mathbf{RQ}_1$: Are the refactoring recommendations produced by $R3$ meaningful from a functional point of view?
- $\mathbf{RQ}_2$: Is the rationale provided by $R3$ meaningful for the proposed refactoring operations?

7.4.1 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.

7.4.1.1 Planning

In the context of our first study with developers, to respond to our research questions we selected ten classes from JHotDraw and for each class we asked the participants to identify the package(s) where the class could be placed. The ten classes were selected among those where $R3$ suggests a move class refactoring, i.e., the package identified by $R3$ is different from the original design package. Specifically, five classes were selected with the confidence level of the suggested package higher or equal to 0.58 and five with the confidence level being lower than 0.58. This choice was the result of our first case study (Section 6.3) where we found that, generally, suggested move class operations having a confidence level higher or equal to 0.58 are able to improve the package modularization, while those having confidence level lower than 0.58 often reduce the quality of the software modularization increasing its average coupling.

The participants evaluated the accuracy of $R3$ through a questionnaire (see see Figure 7.3 for an excerpt of the questionnaire and Appendix **??** for the materials used in our study). For each class in the survey, the participants had three possible options (three possible packages from JHotDraw). The three packages consisted of (i) the original package, i.e., the package where the class was originally implemented, (ii) the suggested package by $R3$, and (iii) a randomly selected package. The latter option was considered only to verify whether participants seriously considered this assignment (that is a sanity check).

In order to respond to our first research question ($\mathbf{RQ}_1$), for each suggested package the developers had to specify if the package was adequate to contain the class under analysis (YES), was not adequate (NO), or might have been adequate (MAYBE). Note that more than one package could be marked as adequate for each class in the survey. Developers that often identify a randomly selected package as a correct answer should be considered as outliers and excluded from the analysis[1]. Note that the participants

[1]In our study we did not identify any outliers.

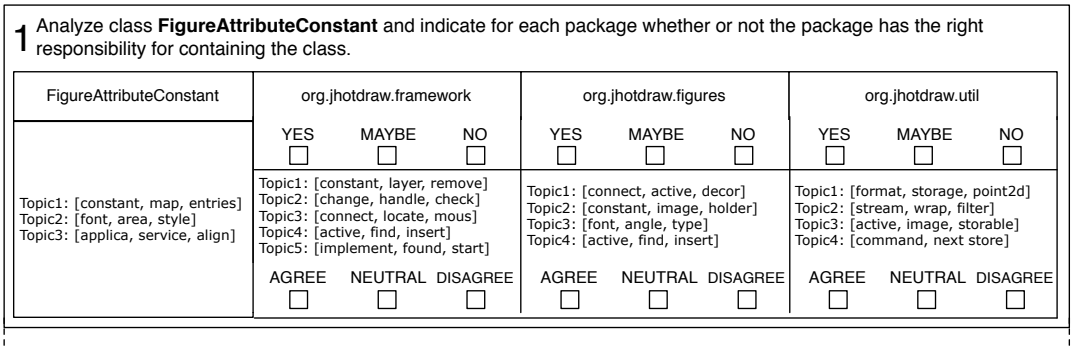

Figure 7.3: An excerpt of the questionnaire used to evaluate R^3.

were not aware of the experimental goals and they did not know the original structure of the system nor the actual packages suggested by $R3$.

We were also interested in evaluating the usefulness of the rationale provided by $R3$ aimed at explaining suggested move class refactoring to the developers ($\mathbf{RQ}_2$). As outlined in Section 7.2, the analysis of underlying latent topics should provide the rationale on why a class should be moved in the suggested package. Thus, for each suggested package and for each class under analysis we also provided the description of their topics extracted using RTM. The developers had to specify whether the rationale provided was meaningful to explain the proposed refactoring (AGREE), was not meaningful (DISAGREE), or could be meaningful (NEUTRAL).

In summary, we had two groups of classes that allowed us to investigate the accuracy of move class refactoring operations recommended by $R3$ in case of high confidence level and low confidence level, respectively. In particular, we had the possibility to analyze whether the package suggested by $R3$ could represent an alternative package for placing the class under analysis.

We analyzed the answers provided by the developers through statistical tests. We collected the rankings of packages in each of the different sets of proposed packages, i.e., original, suggested by $R3$, and random. Then, considering two particular sets, e.g., original *vs.* suggested packages, we used the Mann-Whitney test (122) to analyze the statistical significance of the difference between the ranking of packages in the two sets. The results were intended as statistically significant at $\alpha = 0.05$.

Table 7.7: Developers' answers in different scenarios.

Scenario	Original package			*R*3 suggested package			Random package		
	YES	MAYBE	NO	YES	MAYBE	NO	YES	MAYBE	NO
High Conf.	53%	23%	24%	54%	21%	25%	5%	12%	83%
Low Conf.	69%	23%	8%	36%	28%	36%	3%	8%	89%

Table 7.8: Results of the Mann-Whitney test.

	High Conf.	Low Conf.
original *vs* random	< 0.01	< 0.01
original *vs* suggested	0.48	< 0.01
suggested *vs* random	< 0.01	< 0.01

7.4.1.2 Analysis of the Results

In order to respond to our first research question ($\mathbf{RQ}_1$), Table 7.7 summarizes the answers provided by the participants to the questions regarding the meaningfulness of the suggested refactoring operations. The answers were grouped based on the particular scenario analyzed, i.e., high confidence level and low confidence level, respectively.

Interesting results have been achieved considering *R*3 suggestions with high confidence level. In this case, the analysis of the results provided by the participants reveals that *R*3's recommendations represent a good alternative choice as compared to the original design. In particular, the developers marked as correct 76% of the original packages (53% YES + 23% MAYBE) and 75% of the suggested packages (54% YES + 21% MAYBE). In addition, in 43% of the cases in this scenario the developers preferred the package suggested by *R*3 instead of the original package, i.e., they marked the package suggested by *R*3 with a better score compared to those assigned to the original package.

In the *low confidence level* scenario, developers generally preferred the original packages as design choice, marking the original packages as correct in 92% of cases (69% YES + 23% MAYBE) while the packages suggested by *R*3 were not considered as a good alternative (36% YES + 28% MAYBE).

All these considerations are also supported by the statistical analysis. Table 7.8 reports the results of the Mann-Whitney tests used to compare the ranking of packages in different sets, i.e., original, suggested by *R*3, and random. As we can see, the only

Table 7.9: Participants' evaluations of explanations provided by *R*3.

Evaluation of the suggested refactoring	AGREE	NEUTRAL	DISAGREE
Accepted package	55%	34%	11%
"Maybe" package	24%	60%	16%
Rejected package	11%	16%	73%

case where the original packages did not obtain a statistically significant higher score than the packages suggested by *R*3 is when the confidence level is high. This confirms that in such a scenario the recommendation by *R*3 represents a valuable alternative to the original design. It is worth noting that this result, together with the significant improvement of quality metrics observed in our first study, highlights the goodness of the refactoring operations suggested with high confidence level by *R*3.

Concerning the analysis of the rationale (or explanation) provided by *R*3 when suggesting a move class refactoring ($\mathbf{RQ}_2$), Table 7.9 shows the answers provided by the participants to the related questions. As we can see, the developers considered the rationale provided by *R*3 as meaningful when they accepted a recommended move class refactoring operation. In such cases, they did not agree on the provided rationale only in 11% of cases (59 out of 420). As expected, the scenario completely changed when the developers were not convinced about the refactoring operations, i.e., *"Maybe" package*. In this case, they were neutral with respect to explanations in 60% of cases while they did not find the rationale useful in 16% of cases. Finally, when the developers did not accept a move class refactoring, they generally disagreed with the rationale provided by *R*3 (expected result).

Summarizing, we can conclude that when *R*3 suggests a move class refactoring operation with high confidence level, the refactoring is usually meaningful from a functional point of view. Moreover, the rationale detailing the purpose of the refactoring recommendation is generally rated as useful by the developers.

7.4.1.3 Threats to validity

In this first user study we involved 48 external developers in the evaluation of the move class refactoring operations proposed by *R*3. The main problem with this study is that external developers did not have deep knowledge of the design of the subject system, i.e., JHotDraw, and, as we explained before, they might not have been aware of some

of the design choices that could appear wrong but that are the results of a rational choice. Moreover, the presence of $R3$ explanations in the questionnaire might have driven the external developers (having only partial knowledge of the system) to accept $R3$ suggestions just because the latent topics in the moved class were similar to those present in the target package. To mitigate these threats we conducted the second user study (see Section 7.4.2) involving original developers of two software systems.

Concerning the number of classes (10) analyzed by the participants, it is rather low if compared to the number of classes in the subject system. However, it is important to note that for each class in our survey external developers had to analyze (i) the responsibilities implemented by the class, and (ii) the responsibilities of each of the three proposed packages. It is clear that for a developer who does not have intimate knowledge of the design of the studied system this is a hard and time consuming task. Thus, that was the realistic number of classes that we could possibly evaluate in the user study, which lasted approximately for two hours. It is not easy to perform such an experimentation using a substantially larger number of classes, unless this user study is conducted in multiple sessions, which would involve substantial organizational overhead.

7.4.2 Evaluation with Original Developers

In this section we report the design of the study with original developers and the results obtained.

7.4.2.1 Planning

The four systems involved in the experimentation were eTour, GESA, SESA, and SMOS (see Table 7.1 for the size and versions of these four systems). We asked 14 of the original developers of eTour, GESA, SESA, and SMOS (5 for GESA, 5 for SMOS, 2 for eTour and 2 for SESA) to analyze 20 move class operations suggested by $R3$ (ten having high confidence level, i.e., ≥ 0.58, and ten having low confidence level, i.e., < 0.58). In particular, the developers filled-in a questionnaire (see Appendix **??** for the material used in our study) where, for each of the suggested operations, they had to respond to the question "*Would you apply the proposed refactoring?*" choosing between YES, i.e., the suggested package represents a better design choice than the original package, MAYBE, i.e., the suggested package represents an equivalent alternative to the original

Table 7.10: Participants' evaluations of the refactoring operations proposed by *R*3 on eTour, GESA, SESA, and SMOS.

System	Scenario	YES	MAYBE	NO
eTour	High Confidence	70%	10%	20%
	Low Confidence	0%	50%	50%
GESA	High Confidence	70%	20%	10%
	Low Confidence	0%	30%	70%
SESA	High Confidence	60%	20%	20%
	Low Confidence	0%	50%	50%
SMOS	High Confidence	50%	40%	10%
	Low Confidence	10%	50%	40%

design, and NO, i.e., the original package represents a better design choice than the suggested package. Clearly, the answers provided to this question allowed us to respond to our first research question ($\mathbf{RQ}_1$) related to the meaningfulness of the refactoring operations suggested by *R*3.

Also in this case we evaluated the usefulness of the rationale provided by *R*3 to explain suggested refactoring operations ($\mathbf{RQ}_2$). Thus, for each package (original and envied) and for each class involved in a refactoring operation we also provided the description of their topics extracted using RTM. As in the previous study, the developers had to specify whether the rationale provided was meaningful to explain the proposed refactoring (AGREE), was not meaningful (DISAGREE), or could be meaningful (NEUTRAL).

Developers analyzed suggested move class refactoring operations independently. After that, they performed a review meeting to discuss their scores and reach a consensus. At the end of the meeting the developers provided only one filled-in questionnaire reporting their comprehensive evaluation. We also asked the developers to provide comments on those positively and negatively evaluated cases.

Table 7.11: Participants' evaluations of explanations provided by $R3$ on eTour, GESA, SESA, and SMOS.

System	Evaluation of the suggested refactoring	AGREE	NEUTRAL	DISAGREE
eTour	Accepted move class	72%	0%	28%
	"Maybe" move class	43%	57%	0%
	Rejected move class	0%	33%	67%
GESA	Accepted move class	72%	14%	14%
	"Maybe" move class	10%	80%	10%
	Rejected move class	12%	12%	76%
SESA	Accepted move class	67%	33%	0%
	"Maybe" move class	0%	100%	0%
	Rejected move class	0%	13%	87%
SMOS	Accepted move class	83%	17%	0%
	"Maybe" move class	12%	66%	22%
	Rejected move class	0%	0%	100%

7.4.2.2 Analysis of the Results

Table 7.10 summarizes the answers provided by the original developers to the question "*Would you apply the proposed refactoring?*" while Table 7.11 shows the evaluations provided by the developers to the rationale provided by $R3$.

As we can see, the study conducted with the original developers confirms the findings of the previous study with external developers. In particular, when $R3$ suggests a move class operation with high confidence level, it is generally meaningful from the developers' point of view ($\mathbf{RQ}_1$). In fact, they accepted in the high confidence scenario 62.5% of operations on average, considering a further 22.5% as a good alternative to the original design. In other words, the percentage of suggested refactoring operations appreciated by original developers in the high confidence level scenario hovers at 85% on average. Only 15% of the operations, on average, are discarded by the developers in the high confidence level scenario. On the contrary, operations suggested with low confidence level are generally discarded by developers (see Table 7.10). In particular, only one out of the 40 refactoring operations suggested with low confidence level are accepted by the developers, while the others are either rejected (52,5% on average) or

considered as a possible alternative to the original design (45%). Note that this result, together with the findings of our previous software metrics evaluation and user study with external developers, confirms the goodness of the confidence level as indicator of the quality of the suggested refactoring operations.

Concerning the rationale provided by $R3$ ($\mathbf{RQ}_2$), as already observed for the external developers, also the original developers generally find meaningful the $R3$'s explanation when they accept a move class operation (74% of cases on average - see Table 7.11). On the other hand, when developers discard a refactoring operation generally do not find the $R3$'s explanation meaningful (85% of cases).

Since this study was conducted with original developers, we performed a lot of discussions with them about the reasons behind their evaluations, in order to get qualitative insight about $R3$'s strengths and weaknesses. The results of these discussions are reported in the following grouped by four different cases:

1. refactoring operations having *high* confidence level and *accepted* by developers;
2. refactoring operations having *low* confidence level and *rejected* by developers;
3. refactoring operations having *high* confidence level and *rejected* by developers;
4. refactoring operations having *low* confidence level and *accepted* by developers.

If we consider the confidence level as an indicator to filter good suggestions of the $R3$ method, the first two cases correspond to success cases, while the remaining two cases correspond to failure cases.

R3's suggestions accepted in the high confidence level scenario

In the high confidence level scenario the original developers accepted most of the refactoring operations suggested by $R3$. Some of these refactoring operations accepted by the developers are discussed in the following.

An interesting case from the eTour system was represented by the move of the class *Point3D* from the package *etour.util* to the package *etour.bean*. The two developers involved in the evaluation of the $R3$ suggestions on eTour agreed on the fact that this move class refactoring should be applied. In fact, the package *etour.bean* in eTour groups together all the entity classes (i.e., Java beans) used in the system and, as explained in the comments of *Point3D*, it represents one of the system's entity classes:

Table 7.12: GESA customization parameters.

Name	Involved Functionality	Description
startTimeLessons	Timetable	String: the start time of the lessons
endTimeLessons	Timetable	String: the end time of the lessons
lunchBreakFlag	Timetable	Boolean: true if a fixed lunch break for all the lessons is planned
lunchBreakStart	Timetable	String: [if lunchBreakFlag==true] the start time of the lunch break
lunchBreakEnd	Timetable	String: [if lunchBreakFlag==true] the end time of the lunch break
availableDays	Timetable	String: the days available to define a timetable, e.g., Mon-Fri

*/*Bean containing the coordinates of a point on the earth's surface.*
*The values of the coordinates must be represented in radians. */*

Also the GESA's developers provided us interesting insight about the reasons behind the acceptance of some refactoring operations in the high confidence level scenario. In particular, interesting cases are those related to four move class operations suggested from the package *customization* to the package *timetableManagement*. *R3* suggests to move these four classes composing the package *customization* to the package *timetable-Management*. All the developers involved in the experimentation marked these four move class refactoring operations as meaningful. Thus, we asked them to comment for us on the rationale behind these refactoring operations. The developers explained that the goal of the package *customization* was to group together all the classes that allowed customizing GESA according to the needs of the University using it. Table 7.12 shows the parameters that can be customized using the classes contained in the *customization* package. It is worth mentioning that all the customization parameters were related to the core functionality of GESA, i.e., the timetable management. For this reason, the developers agreed that the package *customization* should be entirely moved into the package *timetableManagement*, possibly creating a package *timetable-Management.customization*.

Finally, a refactoring operation particularly appreciated by the SESA developers was the move of the class *ShowPendingProjectAction* from its package *personManagement* to the envied package *projectManagement*. The reason is quite simple. SESA assigns "pending" status to all the information (e.g., publications, research projects) input to the system by a user, who is not an administrator. This simply means that the inserted information must be approved by an administrator to be visible to all the

users. The class *ShowPendingProjectAction* shows 'pending research projects" that need to be approved by the administrators. This class was put inside the package *personManagement* by the system developers since it was logically linked to the system administrator. However, there is also a package in SESA grouping all the classes related to the research projects management, i.e., *projectManagement*. For this reason the developers felt that the *R*3's suggested package is a better place to put the analyzed class.

R3's suggestions rejected in the low confidence level scenario

In the low confidence level scenario the original developers rejected most of the refactoring operations suggested by *R*3. In the following we discuss some of these cases explaining the reasons behind the decision of the developers.

A first case is the one from the eTour system and related to the move of the class *AdvertisementManagement* from the package *etour.control.advertisementManagement* to the envied package *etour.control.restaurantManagement*. eTour allows the restaurants registered to the system to insert advertisements shown to the tourists when they are near them. For this reason there are a lot of structural dependencies among the class *AdvertisementManagement* and the package *etour.control.restaurantManagement*. These dependencies are the main explanation behind the *R*3 suggestion, although it is provided with a low confidence level. However, in eTour all the classes implementing responsibilities related to the advertisement management are grouped inside the package *control.AdvertisementManagement* and this explains the negative evaluation of this refactoring by the developers.

Another interesting example of *R*3's suggestion in the low confidence level scenario is the move of the class *ManagerStudent* from the package *userManagement* to the package *examSessionManagement* in the GESA system. The class *ManagerStudent* is the class managing the user role "Student" and was correctly included in the package *userManagement* (that includes all the classes for the users of the system), while the package *examSessionManagement* is the only package that implements functionality that students can access, in particular the reservation for the examination sessions. Both the class *ManagerStudent* and the package *examSessionManagement* where included in the version 2.0 of GESA, while the previous version did not implement any functionality that the students could access. All the developers agree that the move

class refactoring suggested by *R*3 did not make sense and that the package *userManagement* is a good place to put this class. We investigated this to better understand the reasons behind *R*3 recommendations. Besides the fact that the user "Student" can only access the functionality concerned with the reservation of examination sessions, we discovered that the class *ManagerStudent* and the package *examSessionManagement* were implemented by the same developer, who used a standard template (containing the same terms) for the comments describing the responsibilities of both, the class *ManagerStudent* and all the classes in the package *examSessionManagement*. This clearly results in textual similarity even between classes having different responsibilities. In this case, the topic analysis performed by *R*3 identifies strong semantic relationships between classes implementing unrelated responsibilities. However, it is worth noting that *R*3 also identifies meaningful dependencies with other packages, including the current package of the class and this is the reason of the low confidence level provided with the refactoring suggestion.

Finally, most of the suggestions with low confidence level discarded by the SESA developers concerned the move of some of the entity classes (i.e., *Article*, *Book*, and *Publication*) from the package *publicationManagement* to the package *researchTopicManagement*. The developers explained that the research topics management in SESA strongly depends on the classes contained in the package *publicationManagement*. In fact, *Article*, *Book*, and *Publication* are linked to each research topic stored in the system.

In general, the analysis performed with software developers about discarded refactoring operations in the low confidence level scenario highlighted that while in some cases refactoring operations can be reasonable when looked from a quality metric point of view[1] (i.e., structural and semantic coupling), they are not necessarily meaningful from the developers' point of view.

R3's suggestions rejected in the high confidence level scenario

The refactoring operations rejected in the high confidence level scenario represent the real failure cases of *R*3. In fact, in this cases the *R*3's confidence level is not able to filter out these that seem to be bad refactoring suggestions. Thus, even if the percentage of

[1]Note that, as observed in our software metrics evaluation, only few refactoring operations having low confidence level are able to improve software quality metrics.

move class operations rejected by the developers in the high confidence level scenario is very low it is important to analyze some of these cases in order to understand the reasons behind the developers' choice.

An example of move class refactoring proposed by *R*3 with a high confidence level and negatively evaluated by developers can be found in the SMOS system. In that particular case *R*3 proposed to move the class *LoginException* from the package *exceptions* to the package *userManagement*. Even if the class *LoginException* is used only by two classes of the *userManagement* package, the developers did not find this move class meaningful since all the classes implementing possible exceptions in the SMOS system are grouped in the *exceptions* package. This design choice was dictated by the fact that most of the exceptions in SMOS are generic and thus, used by more subsystems (e.g., *MandatoryFieldException*). However, it is worth noting that an alternative design choice could be the one proposed by *R*3, where a class implementing an exception used only by one subsystem is placed inside it.

Also the eTour developers discussed with us an interesting case of high confidence level suggestion that makes no sense from their point of view. It is related to the move of the class *ConvertFile* from its package *etour.utility* to the suggested package *etour.control.advertisementManagement*. *ConvertFile* is used by the classes contained in the *etour.control.advertisementManagement* package to convert all the images uploaded as advertisements by the restaurants registered to the system in the JPEG format. While this explain the rational behind the *R*3 suggestion, the eTour developers felt that the right package to place *ConvertFile* is the *utility* package, grouping together miscellaneous functionalities that might be useful to different subsystems.

The two reported examples of rejected *R*3's suggestions having high confidence level pinpoint how even reasonable refactoring operations do not always justify the need to change the original design from developers' point of view. This highlight as the last word about the application of a refactoring operation should always be left to the developer.

R3's suggestions accepted in the low confidence level scenario

While several refactoring operations suggested with a low confidence level have been classified by the developers as possible alternatives to the original design, only one for the SMOS system has been accepted, thus confirming the ability of the confidence level

as indicator of the goodnesses of the suggested refactoring operation. We considered this as an interesting case to discuss with the developers. The refactoring involved the move of the class *ServletLoadYear* from its package *userManagement* to the envied package *classroomManagement*. The class *ServletLoadYear* is used only by classes in these two packages to load at runtime the list of academic years for which SMOS stores information in the system (e.g., information about the classrooms, students, etc.). *ServletLoadYear* was originally included in the package *userManagement*, because this package was developed before *classroomManagement*. The developers accepted the refactoring suggestion, because this class is used by more classes in *classroomManagement* than in *userManagement*. However, as this class is an utility class the choice of whether it should be placed in one or the other package is questionable. Indeed, the developers clarified that this class would have been a candidate to be placed in a package grouping other utility classes, but such a package was not included in the system. It is worth noting that $R3$ supports move class refactoring operations and is not intended to create new packages. However, while $R3$ suggestions with low confidence level should not be considered as good move class refactoring operations, they could be investigated to possibly identify other types of refactoring opportunities.

7.4.2.3 Threats to validity

In our second user study we involved 14 original developers of four software systems, namely eTour, GESA, SESA, and SMOS. The original developers had thorough knowledge of all the design choices that led to the original design. Thus, they were good candidates for evaluating the meaningfulness of the refactoring operations proposed by $R3$. However, as with external developers, involving original developers as participants has a downside. In fact, as explained before, some of them could be the "fathers" of some of bad design choices and consequently might not have been able to recognize a good move class suggested by $R3$ as meaningful. However, the results obtained and thorough discussions with them about some of the good suggestions provided by $R3$ demonstrate that the developers provided an objective evaluation of the analyzed move class operations.

The number of move class operations (20) in the experimentation with the original developers is twice as large as compared to the study with external developers. This is reasonable as in this case the participants had knowledge of system modularization

and they only had to analyze the move class operations recommended by R3 as an alternative to the original design. Still such a number of refactoring operations might be considered as small. However, we preferred to dedicate more time to have more meaningful and detailed discussions with the developers about some interesting cases rather than asking them to analyze a higher number of move class operations.

7.5 Final Remarks

In this Chapter is presented and evaluated *R*3, an approach based on RTM, a probabilistic topic modeling technique, to improve the quality of software modularization. The proposed approach analyzes underlying latent topics in classes and packages as well as it uses structural dependencies to recommend refactoring operations aiming at moving classes to more suitable packages. Unlike most of the previous work, the proposed approach avoids the creation of a whole new remodularization (and the consequent creation/removal of existing packages), proposing a set of move class operations that can be applied independently one from each other. In addition, *R*3 is the first refactoring recommendation tool also providing some feedback to the developer about the goodnesses of the suggested operations (i.e., confidence level) and rationale behind the proposed recommendations.

The approach has been first evaluated through well-established metrics that capture quality improvement achieved while applying the proposed refactoring operations on nine software systems. The results achieved indicated that *R*3 provides a coupling reduction ranging from 10% to 30% among the software modules. Then, we evaluated the refactoring recommendations by *R*3 in two user studies: one conducted with 14 original developers of four software systems and one with 44 students and academics plus four professional software developers on an open source software system. The results achieved in this second case study indicated that more than 70% of the recommendations provided by *R*3 with high confidence level were considered meaningful from a functional point by developers.

8

ARIES: Automated Refactoring In EclipSe

The material in this Chapter has been presented in (149).

8.1 Introduction

ARIES is an Eclipse plug-in born with the aim of providing the software engineer a complete support in performing different refactoring operations. ARIES implements the approaches to support Extract Class, Extract Package, Move Method, and Move Class refactoring described in Chapter 3.

In the following is presented the Extract Class functionality of ARIES. The other functionalities are implemented following the same design. A video of the ARIES Extract Class feature is available on Youtube[1].

8.2 ARIES at Work: Extract Class Refactoring

ARIES supports Extract Class refactoring with a three steps wizard. In the first two steps the tool provides support to the software engineer to identify and analyze Blobs in the system under analysis. In the third step the software engineer receives recommendations on how to refactor the candidate Blobs. The following sections present details on the three steps of the Extract Class refactoring process in ARIES.

[1] http://www.youtube.com/watch?v=csfNhgJlhH8

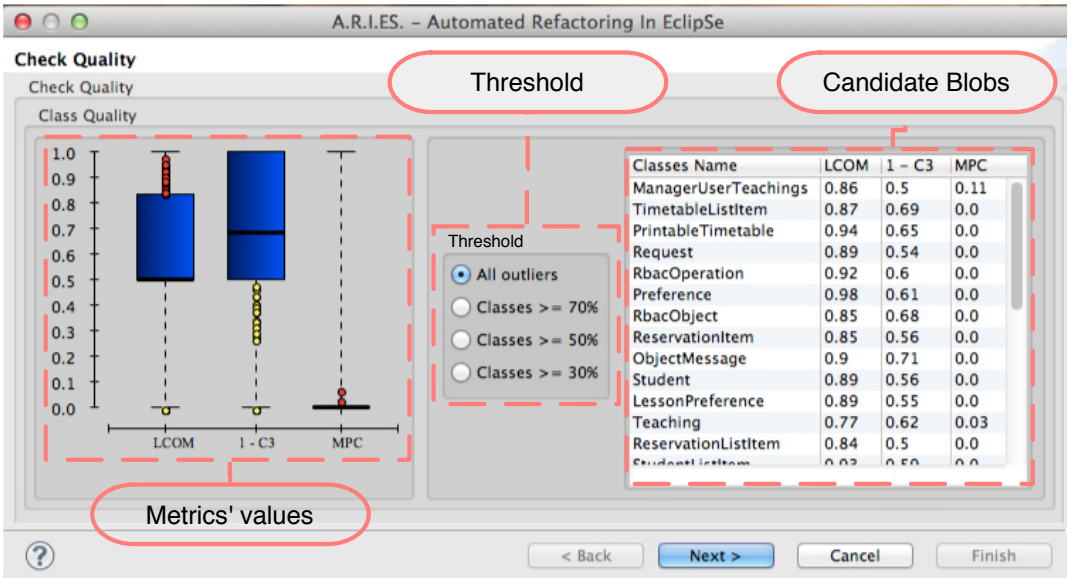

Figure 8.1: ARIES: Identification of candidate Blobs.

8.2.1 Identifying Candidate Blobs

ARIES supports the software engineer in the detection of Blobs through a quality check of the entire software system (or of a specified subsystem). Note that ARIES does not compute an *overall* quality of the classes, but it considers only cohesion and coupling as the main indicators of class quality in this context. Hence, Blobs are usually outliers or classes having a quality much lower than the average quality of the system under analysis (100). The identification of Blobs in ARIES is based on such a conjecture.

The software engineer starts the quality check selecting the *Check Quality* command in the main menubar. ARIES computes three quality metrics for each class of the (sub)system, namely, Lack of Cohesion of Methods (LCOM5) (9), Conceptual Cohesion of Classes (C3) (60), and Message Passing Coupling (MPC) (109). The results are shown to the developers as three boxplots (one for each quality metric) highlighting any negative outlier, i.e., classes having cohesion (coupling) markedly lower (higher) than the other classes of the system (see Figure 8.1). Note that for the C3 metric ARIES shows the values of $1 - C3$. In this way for all the three measures the negative outliers are reported on the top of the boxplot.

All the outliers are reported in the list shown in the right side of Figure 8.1. Note

that the list will include all the classes that are negative outliers for at least one of the three metrics. In case no outliers are identified, ARIES allows to "relax" the process used to identify candidate Blobs. In particular, instead of a statistical identification of the outliers, the software engineer can select a threshold $\lambda \in \{70, 50, 30\}$ that allows to recover as candidate Blobs all the classes having a quality (in terms of the employed quality metrics) lower than $\lambda\%$ of the average quality. In the scenario shown in Figure 8.1 the software engineer decides to analyze the quality of the system in order to identify Blobs. ARIES shows the boxplots for the values of LCOM, C3 and MPC. The software engineer decides to improve the quality of some classes having a quality (in terms of cohesion and coupling) sensibly worse than the average quality of the system. In the top of the list of outliers there is the class `ManagerUserTeaching` that seems to be a good candidate for refactoring. In order to obtain a detailed view on `ManagerUserTeaching`, the developer selects the class from the list and clicks on the "Next" button activating the second step of the wizard.

Note that the *Identification* step is not mandatory in order to perform Extract Class refactoring. The developer can directly select a class in the Package Explorer and start the class extraction process by clicking the Extract Class refactoring button in the main toolbar.

8.2.2 Analyzing Candidate Blobs

The second step of the ARIES wizard aims at helping the software engineer in better analyzing the classes that are candidates for refactoring. The tool shows the preview of the class under analysis as well as its topic map (see Figure 8.2). The topic map of a generic class C is built by analyzing the term frequency in the methods it contains. The five most frequent terms (the terms present in the highest number of methods) are used to construct the topic map of C that, for this reason, is represented by a pentagon where each vertex represents one of the main topics. Each vertex is connected to the center of the pentagon by an axis representing the percentage of methods in C that implements the corresponding topic. The graphical representation of the main topics of C is then obtained by tracing lines between the percentage points on each of the five axes indicating the percentage of methods belonging to C that implement the corresponding topic. Note that a stop-word list is used to automatically prune out common English words and Java keywords. This stop-word list can be customized using

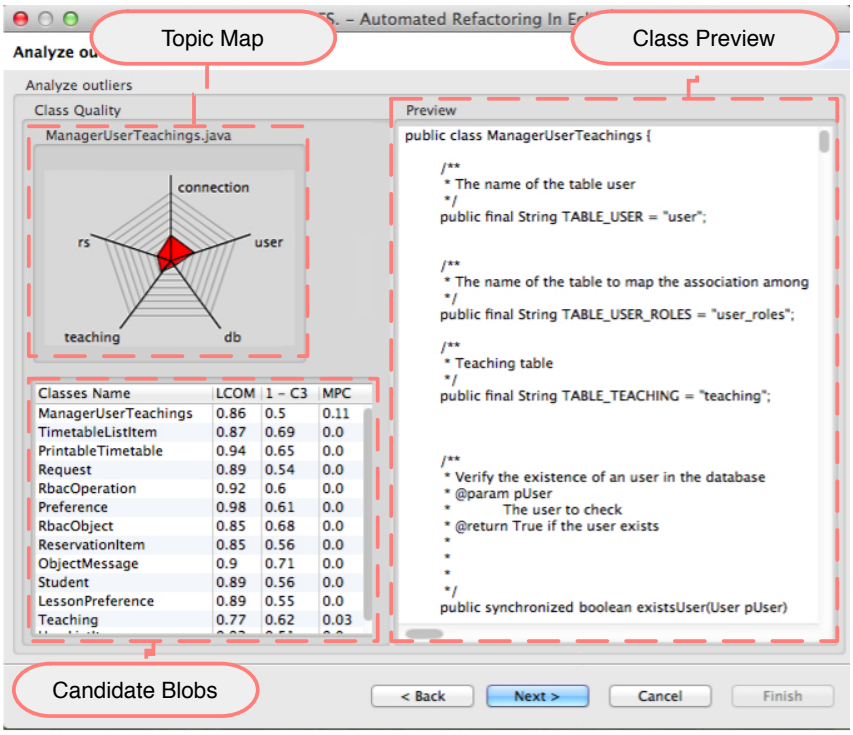

Figure 8.2: ARIES: Analysis of candidate Blobs.

the ARIES preference panel. The topic map provided for the Blob is meant to help the developer in understanding which are the different responsibilities implemented in the class. Clearly not all topic maps will be equally helpful, as they depend on identifiers and comments in the code.

In the scenario shown in Figure 8.2 the software engineer is analyzing the class `ManagerUserTeaching`. From the analysis of the topic map it is easy to identify three different responsibilities of the class, i.e., database connection (indicated by the terms `connection`, `db`, and `rs`), user management (indicated by the term `user`), and teaching management (indicated by the term `teaching`). While the first responsibility is a common for each control class (each control class in the analyzed system accesses the database to manage particular information), the other two responsibilities are quite different suggesting that the quality of the class (in terms of cohesion) could be improved splitting it in different classes.

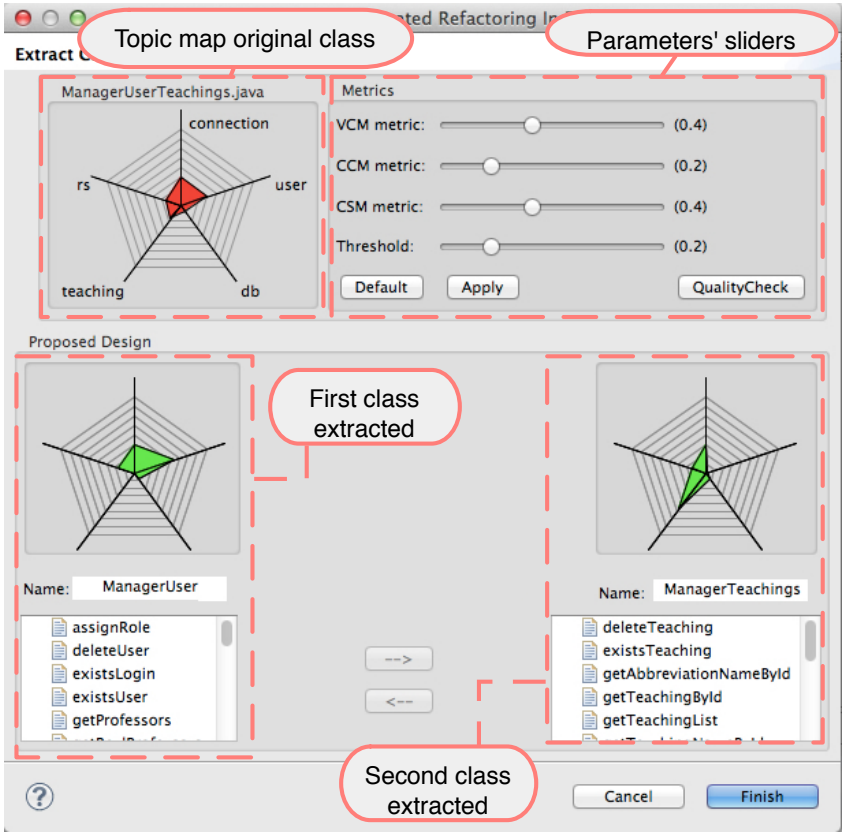

Figure 8.3: ARIES: Extract Class refactoring.

8.2.3 Refactoring the Blobs

Figure 8.3 shows the final step of the Extract Class refactoring feature of ARIES. The upper part of Figure 8.3 contains the topic map of the class to be refactored. The right part contains all the sliders to configure parameters of the Extract Class refactoring approach, i.e., the weights for the similarity measures and the threshold *minCoupling* (see Chapter 4). Although initial default values achieved through an empirical assessment 4 are provided to the developer, she can modify any parameter, changing on-the-fly the resulting refactoring recommendation, shown in the bottom part of Figure 8.3.

ARIES reports for each class that should be extracted from the Blob the following information: (i) its topic map; (ii) the set of methods composing it; and (ii) a text field where the developer can assign a name to the class. The tool also allows the developer to

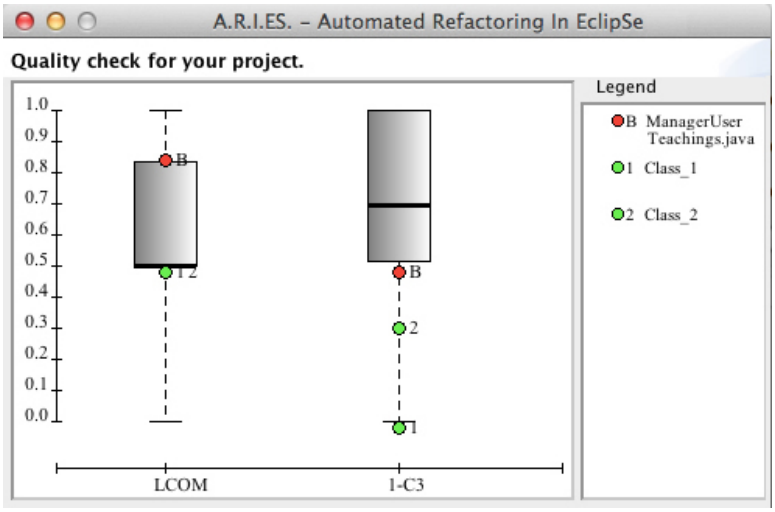

Figure 8.4: ARIES: Quality Check of the refactoring operation

customize the proposed refactoring moving the methods between the extracted classes.

In the scenario of Figure 8.3, ARIES splits the class `ManagerUserTeaching` into two classes. The topic maps of the extracted classes help to understand the rationale behind the refactoring recommendation. The first class is in charge of managing the users, while the second class is responsible of teaching management. From the analysis of the topic maps it is also possible to see that database connection is a responsibility of both classes. This means that both classes access the database to manage users and teachings, respectively.

Besides a conceptual analysis (based on the topic maps) of the refactoring proposed by ARIES, the developer can quantify the quality improvement obtained applying the proposed refactoring. Using the functionality "Quality Check", ARIES highlights on the boxplots of the metrics showed in the first step of the wizard, the values of the metrics for the new classes and the original Blob (see Figure 8.4). In this way the developer can analyze the quality of the new classes as compared to the overall quality of the system.

To terminate the extraction process and automatically generate the new classes, the software engineer can click the "Finish" button (see right lower corner in Figure 8.3). ARIES will generate the new classes making sure that the changes made by the refactoring do not introduce any syntactic error.

8.3 Final Remarks

This Chapter presented ARIES, an Eclipse plug-in supporting the software engineer during refactoring operations. ARIES provides support in (i) the identification of the design flaws to remove from the system through quality metrics, (ii) their analysis supported by topic maps, and (iii) the identification and application of the refactoring solution. Future work will be devoted to the implementation of more sophisticated approaches to detect design flaws in a software system (see e.g., (72)).

9

Conclusion

9.1 Concluding Remarks

This thesis presented a framework of approaches to support the software engineer in performing four different refactoring operations, i.e., Extract Class, Extract Package, Move Method, and Move Class refactoring. All the approaches exploit both structural and semantic (lexical) information extracted from the source code to suggest refactoring solutions. The proposed techniques have been deeply evaluated posing particular attention on evaluation conducted with software developers.

The contributions can be summarized as follow:

1. A novel approach to support Extract Class refactoring (115, 116). Structural and semantic information are exploited to identify which methods of the Blob class implement similar responsibilities and thus, should be grouped together inside a new extracted class. Unlike the approaches based on clustering algorithms (see e.g., (19)) the proposed approach automatically identifies the appropriate number of classes that should be extracted from a Blob class. The proposed approach has been experimented on real Blobs of open source systems to evaluate how good the proposed solution is from a cohesion and coupling metrics point of view, and how good the proposed refactoring solution is considered by software engineers as it is. Then, we evaluated on a set of 11 classes how much the proposed solution is useful as starting point to perform a refactoring, and how well the proposed refactoring solution approximate a refactoring made by the original developers. The results show that (i) the refactoring solutions proposed

by our approach strongly increases the cohesion of the refactored classes without leading to significant increases in terms of coupling, (ii) the refactoring solutions proposed by our approach are considered useful to developers performing extract class refactoring and (iii) the proposed approach is able to approximate a manually performed refactoring at 91% on average.

2. A novel approach to support Extract Package refactoring (130). The approach splits a given promiscuous package into new packages having higher cohesion by analyzing structural and semantic dependencies between the classes grouped in it. The approach is able to automatically identify the correct number of packages that should be extracted from the promiscuous one. A deep evaluation conducted with 16 developers on five software systems highlights the goodnesses of the suggested Extract Package refactoring operations.

3. Methodbook, an approach to support Move Method refactoring (138, 139). Methodbook exploits Relational Topic Model (RTM) (62) to suggest Move Method refactoring opportunities in software. 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 quality of five software systems from a quality metrics point of view. The results indicate that the Methodbook's suggestions having high confidence level are able to significantly improve cohesion and coupling of the subject systems. Moreover, on three out of five experimented systems Methodbook uniformly outperforms JDeodorant, while on the remaining two systems the two approaches almost reached a tie. In a 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 30 students on an open source software system. The results indicate that Methodbook provides meaningful recommendations for move method refactoring from a developer's point of view. In addition, the developers generally prefer Methodbook's recommendations compared to those produced by JDeodorant.

4. *R*3 (Rational Refactoring via RTM) (144), an approach to improve the modularization quality of an object-oriented system through Move Class refactoring

operations. *R*3 exploits RTM to analyze underlying latent topics in classes and packages and uses structural dependencies to recommend refactoring operations aiming at moving classes to more suitable packages. In addition, *R*3 is the first refactoring recommendation tool also providing some feedback to the developer about the goodnesses of the suggested operations (i.e., confidence level) and rationale behind the proposed recommendations. *R*3 has been evaluated in two empirical studies. In the first study we analyzed the ability of *R*3 to propose refactoring operations that lead to reduced coupling among software modules in nine software systems. The results achieved indicated that *R*3 provides a coupling reduction ranging from 10% to 30% among the software modules. In the second study, we evaluated *R*3 refactoring recommendations with developers in two case studies, one conducted with 14 original developers of four software systems and one with 44 students and academics plus 4 professional software developers on another open source software system. The results achieved in this second case study indicated that more than 70% of the recommendations provided by *R*3 with high confidence level were considered meaningful from a functional point by developers.

The framework of approaches described in this thesis has also been implemented in ARIES (Automated Refactoring In EclipSe) (149), an Eclipse plug-in supporting the software engineer in performing different refactoring operations.

9.2 Further Work

We are working to enrich our framework by experimenting new ways of supporting refactoring/re-modularization activities. In particular, we started analyzing the use of Interactive Genetic Algorithms (IGAs) in the context of software re-modularization (150). IGAs are a variant of Genetic Algorithms (GAs) in which the fitness function is partially or entirely evaluated by a human while the GA evolves. In (150) we presented an approach in which part of the fitness (capturing aspects such as intra-module, extra-module dependencies, or modularization quality) is automatically evaluated, while the human adds penalties for artifacts that are not where they should be. We performed a preliminary evaluation showing that the IGAs are able to propose re-modularizations (i) more meaningful from a developers point-of-view, and (ii) not worse, and often

even better in terms of modularization quality, with respect to those proposed by the non-interactive GAs. For these reasons we plan to further investigate the performances of IGAs in software re-modularization as well as in supporting different refactoring operations.

A second direction we are investigating is the application of Game-Theory in the context of refactoring (137). The rational behind this choice is that during refactoring developers must often find solutions to problems while balancing competing goals, e.g., cohesion versus coupling. We believe contrasting goals can be often dealt with game theory techniques. Indeed, game theory is successfully used in other fields, especially in economics, to mathematically propose solutions to strategic situation, in which an individual's success in making choices depends on the choices of others. We demonstrated the applicability of game theory (and in particular of its non-cooperative game field) to refactoring in (137). The preliminary evaluation performed in an artificial scenario demonstrates the applicability and the benefits provided by the use of Game-Theory. However, we still need to investigate additional game theory techniques, such as cooperative games, and determine through a deeper evaluation how refactoring techniques based on Game-Theory compare to the approaches presented in this thesis and to the state of the art in general.

Finally, we are also investigating through empirical studies what is the "price to pay" for refactoring in terms of bug introduction. In (151) we used an existing tool, namely Ref-Finder (117), to automatically detect refactoring operations of 52 different types on 63 releases of three Java software systems. Of the 15,008 refactoring operations detected by the tool, 12,922 operations have been manually validated as actually refactorings. Then, we used the SZZ algorithm (152, 153) to determine whether the 12,922 refactoring operations induced bug fixes. Results show that while, in general, the percentage of bugs fixes likely induced by refactorings is relatively low (i.e., 15%), there are some specific kinds of refactorings that are very likely to induce fixes. In particular, *Pull Up Method* and *Extract Subclass* (two refactoring operations related to changes applied to the class hierarchy) induce (in percentage) more fixes than the others. In this context we want to replicate our study in order to corroborate the achieved results. Moreover, we are planning more studies aimed at analyzing what is the effect of refactoring operations on several quality aspects of software systems (e.g., quality metrics, design patterns, etc.).

Appendix A

Publications Presented in this Thesis

A.1 Accepted

A.1.1 Journal

- G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia. **Methodbook: Recommending Move Method Refactorings via Relational Topic Models**. *Transactions on Software Engineering (TSE)*. 2013. To Appear.
- G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, A. De Lucia. **Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies**. *Transactions on Software Engineering and Methodologies (TOSEM)* (2013) To appear.
- G. Bavota, A. Marcus, A. De Lucia, R. Oliveto. **Automating Extract Class Refactoring: an Improved Method and its Evaluation**. *Empirical Software Engineering (EMSE)*. 2013. To appear.
- G. Bavota, A. Marcus, A. De Lucia, R. Oliveto. **Using Structural and Semantic Measures to Improve Software Modularization**. *Empirical Software Engineering (EMSE)* (2013) To appear. doi: 10.1007/s10664-012-9226-8
- G. Bavota, A. De Lucia, R. Oliveto. **Identifying Extract Class Refactoring Opportunities Using Structural and Semantic Cohesion Measures**. *The Journal of Systems and Software (JSS)* 84 (2011), pp. 397-414.

A.1.2 Conferences

- G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia. **An Empirical Study on the Developers Perception of Software Coupling**. In: *Proceedings of*

the 35th International Conference on Software Engineering (ICSE 2013), San Francisco, USA, 2013. 10 pages. To appear.

- G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, O. Strollo. **When does a Refactoring Induce Bugs? An Empirical Study**. In: *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM 2012)*, Riva del Garda, Italy, 2012, pp. 104-113.
- G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, R. Oliveto. **Putting the Developer in-the-loop: an Interactive GA for Software Re-Modularization**. In: *Proceedings of the 4th Symposium on Search Based Software Engineering (SSBSE 2012)*, Riva del Garda, Italy, 2012, pp. 75-89.
- G. Bavota. **Using Structural and Semantic Information to Support Software Refactoring**. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012, Doctoral Symposium, pp. 1479-1482.
- G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, F. Palomba. **Supporting Extract Class Refactoring in Eclipse: The ARIES Project**. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012, Formal Tool Demo, pp. 1419-1422.
- R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, A. De Lucia. **Identifying Method Friendships to Remove the Feature Envy Bad Smell (NIER Track)**. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, Waikiki, Honolulu, Hawaii, 2011, pp. 820-823.
- G. Bavota, A. De Lucia, A. Marcus, R. Oliveto. **Software Re-Modularization based on Structural and Semantic Metrics**. In: *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE 2010)*, Beverly, Massachusetts, USA, 2010, pp. 195-204.
- G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, Y-G. Gueheneuc. **Playing with Refactoring: Identifying Extract Class Opportunities through Game Theory**. In: *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2010)*, Timisoara, Romania, 2010.
- G. Bavota, A. De Lucia, A. Marcus, R. Oliveto. **A Two-Step Technique for Extract Class Refactoring**. In: *Proceedings of the 25th International Conference on Automated Software Engineering (ASE 2010)*, Antwerp, Belgium, 2010, pp. 151-154.

Appendix B

Other Articles Published during the PhD period

B.1 Journal

- A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, D. Binkley. **Evaluating Test-to-Code Traceability Recovery Methods through Controlled Experiments**. *Journal of Software: Evolution and Process (JSME)* (2013) To appear.

- G. Bavota, C. Gravino, R. Oliveto, A. De Lucia, G. Tortora, M. Genero, J. Cruz-Lemus. **A Fine-Grained Analysis of the Support Provided by UML Class Diagrams and ER Diagrams During Data Model Maintenance**. *Software and Systems Modeling (SOSYM)* (2013) To appear.

B.2 Conferences

- S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, Tim Menzies. **Automatic Query Reformulations for Text Retrieval in Software Engineering**. In: *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, San Francisco, USA, 2013, To appear.

- S. Haiduc, G. Bavota, R. Oliveto, A. De Lucia, A. Marcus. **Automatic Query Performance Assessment during the Retrieval of Software Artifacts**. In: *Proceedings of the 27th International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, 2012, pp. 90-99.

- G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, D. Binkley. **An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Mainte-**

nance. In: *Proceedings of the 28th International Conference on Software Maintenance (ICSM 2012)*, Riva del Garda, Italy, 2012, pp. 56-65.

- S. Haiduc, G. Bavota, R. Oliveto, A. Marcus, A. De Lucia. **Evaluating the Specificity of Text Retrieval Queries to Support Software Engineering Tasks**. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012, NIER Track, pp. 1273-1276.
- G. Bavota, A. De Lucia, F. Fasano, R. Oliveto, C. Zottoli. **Teaching Software Engineering and Software Project Management: An Integrated and Practical Approach**. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012, Software Engineering Education Track, pp. 1155-1164.
- G. Bavota, L. Colangelo, A. De Lucia, S. Fusco, R. Oliveto, A. Panichella. **TraceME: Traceability Management in Eclipse**. In: *Proceedings of the 28th International Conference on Software Maintenance (ICSM 2012)*, Tool Demo, Riva del Garda, Italy, 2012, pp. 642-645.
- G. Bavota, C. Gravino, R. Oliveto, A. De Lucia, G. Tortora, M. Genero, J. A. Cruz-Lemus. **Identifying the Weaknesses of UML Class Diagrams during Data Model Comprehension**. In: *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2011)*, Wellington, New Zealand, pp. 168-182.
- A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, D. Binkley. **SCOTCH: Improving Test-to-Code Traceability using Slicing and Conceptual Coupling**. In: *Proceedings of the 27th International Conference on Software Maintenance (ICSM 2011)*, Williamsburg, VA, USA, 2011, pp. 63-72.
- A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, D. Binkley. **SCOTCH: Slicing and Coupling based Test to Code trace Hunter**. In: *Proceedings of the 18th Working Conference on Reverse Engineering*, Tool Demo, Limerick, Ireland, 2011, pp. 443-444.

References

[1] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. **Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures**. *Journal of Systems and Software*, **84**:397–414, March 2011. vii, ix, 11, 28, 43, 45, 47, 50, 51, 53, 55, 59, 60, 61, 62, 64, 65, 67, 69, 70, 71, 72, 73, 74, 75, 76, 77, 86, 87, 112, 121

[2] Adrian Trifu and Radu Marinescu. **Diagnosing Design Problems in Object Oriented Systems**. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 155–164, Pittsburgh, PA, USA, 2005. IEEE Press. ix, 9, 10

[3] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999. 1, 2, 7, 16, 20, 37, 43, 74, 119, 147, 152

[4] T. Mens and T. Tourwe. **A survey of software refactoring**. *IEEE Transactions on Software Engineering*, **30**(2):126–139, 2004. 1

[5] Victor R. Basili, Lionel Briand, and Walcélio L. Melo. **A Validation Of Object-Oriented Design Metrics As Quality Indicators**. *IEEE Transactions on Software Engineering*, **22**(10):751–761, 1995. 1, 8

[6] Aaron B. Binkley and Stephen 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*, pages 452–455, Kyoto, Japan, 1998. 1, 8

[7] Lionel C. Briand, Juergen Wuest, and Hakim Lounis. **Using Coupling Measurement for Impact Analysis in Object-Oriented Systems**. In *Proceedings of the 15th IEEE International Conference on Software Maintenance*, pages 475–482, Oxford, UK, 1999. IEEE Press. 1, 8

[8] Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. **Investigating quality factors in object-oriented designs: an industrial case study**. In *Proceedings of the 21st International Conference on Software Engineering*, pages 345–354, Los Angeles, California, United States, 1999. ACM Press. 1, 8

[9] S. R. Chidamber and C. F. Kemerer. **A Metrics Suite for Object Oriented Design**. *Transactions on Software Engineering*, **20**(6):476–493, June 1994. 1, 8, 21, 51, 133, 180

[10] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, 1992. 1

[11] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004. 2

[12] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004. 2

[13] Robert C. Martin. *Clean Code: A handbook of agile software craftsmanship*. Prentice Hall, 2009. 2

[14] E. Casais. **An Incremental Class Reorganization Approach**. In *Proceedings of European Conference on Object-Oriented Programming*, pages 114–132, Utrecht, the Netherlands, 1992. 2, 20

[15] I. Moore. **Automatic inheritance hierarchy restructuring and method refactoring**. In *Proceedings of 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 235–250, San Jose, California, USA, 1996. ACM Press. 2, 20

[16] K. Maruyama and K. Shima. **Automatic method refactoring using weighted dependence graphs**. In *Proceedings of 21st International Conference on Software Engineering*, pages 236–245, Los Alamitos, California, USA, 1999. ACM Press. 2, 21

REFERENCES

[17] M. O'Keeffe and M. O'Cinneide. **Search-Based Software Maintenance**. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering*, pages 249–260, Bari, Italy, 2006. IEEE CS Press. 2, 21, 75, 112, 161

[18] Olaf Seng, Johannes Stammel, and David Burkhart. **Search-based determination of refactorings for improving the class structure of object-oriented systems**. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1909–1916, Seattle, Washington, USA, 2006. 2, 18, 19, 75, 112, 161

[19] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiou, and Jörg Sander. **Decomposing object-oriented class modules using an agglomerative clustering technique**. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 93–101, Edmonton, Canada, 2009. 2, 11, 44, 187

[20] Nikolaos Tsantalis and Alexander Chatzigeorgiou. **Identification of Move Method Refactoring Opportunities**. *IEEE Transactions on Software Engineering*, **35**:347–367, 2009. 2, 16, 18, 19, 122, 127, 133, 134

[21] Nikolaos Tsantalis and Alexander Chatzigeorgiou. **Identification of extract method refactoring opportunities for the decomposition of methods**. *Journal of Systems and Software*, **84**(10):1757–1782, October 2011. 2, 21, 22

[22] [online]2011 [cited June 15, 2011]. [link]. 2

[23] L.H. Etzkorn and C.G. Davis. **Automatically identifying reusable OO legacy code**. *IEEE Computer*, **30**(10):66 –71, 1997. 3, 22

[24] A. Michail and D. Notkin. **Assessing software libraries by browsing similar classes, functions and relationships**. In *Proceedings of 21st International Conference on Software Engineering*, pages 463–472, Los Angeles, California, USA, 1999. IEEE CS Press. 3, 22

[25] Yunwen Ye and Gerhard Fischer. **Supporting reuse by delivering task-relevant and personalized information**. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 513–523, 2002. 3, 22

[26] Y. Pan, L. Wang, L. Zhang, B. Xie, and F. Yang. **Relevancy based semantic interoperation of reuse repositories**. In *Proceedings of 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 211–220, Newport Beach, California, USA, 2004. ACM Press. 3, 22

[27] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. **Information Retrieval Models for Recovering Traceability Links between Code and Documentation**. In *Proceedings of 16th IEEE International Conference on Software Maintenance*, pages 40–51, San Jose, California, USA, 2000. IEEE CS Press. 3, 22

[28] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. **Recovering traceability links between code and documentation**. *IEEE Transactions on Software Engineering*, **28**(10):970–983, 2002. 3, 22, 23

[29] A. Marcus and J. I. Maletic. **Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing**. In *Proceedings of 25th International Conference on Software Engineering*, pages 125–135, Portland, Oregon, USA, 2003. IEEE CS Press. 3, 22

[30] J. H. Hayes, A. Dekhtyar, and J. Osborne. **Improving Requirements Tracing via Information Retrieval**. In *Proceedings of 11th IEEE International Requirements Engineering Conference*, pages 138–147, Monterey, California, USA, 2003. IEEE CS Press. 3, 22

[31] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. **Enhancing an Artefact Management System with Traceability Recovery Features**. In *Proceedings of 20th IEEE International Conference on Software Maintenance*, pages 306–315, Chicago, Illinois, USA, 2004. IEEE CS Press. 3, 22

[32] M. Lormans and A. Van Deursen. **Reconstructing requirements coverage views from design and test using traceability recovery via LSI**. In *Proceedings of 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 37–42, Long Beach, California, USA, 2005. ACM Press. 3, 22

[33] S. Yadla, J. Huffman Hayes, and A. Dekhtyar. **Tracing Requirements to Defect Reports: An Application of Information Retrieval Techniques**. *Innovations in Systems and Software Engineering: A NASA Journal*, **1**(2):116–124, 2005. 3, 22

[34] A. Marcus, J. I. Maletic, and A. Sergeyev. **Recovery of Traceability Links Between Software Documentation and Source Code**. *International Journal of Software Engineering and Knowledge Engineering*, **15**(5):811–836, 2005. 3, 22

[35] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. **Can Information Retrieval Effectively Support Traceability Link Recovery?** In *Proceedings of 14th IEEE International Conference on Program Comprehension*, pages 307–316, Athens, Greece, 2006. IEEE CS Press. 3, 22

[36] A. De Lucia, R. Oliveto, and P. Sgueglia. **Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery**. In *Proceedings of 22nd IEEE International Conference on Software Maintenance*, pages 299–309, Sheraton Society Hill, Philadelphia, Pennsylvania, 2006. IEEE CS Press. 3, 22

[37] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. **Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods.** *IEEE Transactions on Software Engineering*, **32**(1):4–19, 2006. 3, 22

[38] M. Lormans and A. van Deursen. **Can LSI help Reconstructing Requirements Traceability in Design and Test?** In *Proceedings of 10th European Conference on Software Maintenance and Reengineering*, pages 45–54, Bari, Italy, 2006. IEEE CS Press. 3, 22

[39] M. Lormans, H. Gross, A. van Deursen, R. van Solingen, and A. Stehouwer. **Monitoring Requirements Coverage using Reconstructed Views: An Industrial Case Study**. In *Proceedings of 13th Working Conference on Reverse Engineering*, pages 275–284, Benevento, Italy, 2006. IEEE CS press. 3, 22

[40] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. **Recovering Traceability Links in Software Artefact Management Systems using Information Retrieval Methods**. *ACM Transactions on Software Engineering and Methodology*, **16**(4), 2007. 3, 22

[41] A. Abadi, M. Nisenson, and Y. Simionovici. **A Traceability Technique for Specifications**. In *Proceedings of 16th IEEE International Conference on Program Comprehension*, pages 103–112, Amsterdam, the Netherlands, 2008. IEEE CS Press. 3, 22

[42] Marco Lormans, Arie Deursen, and Hans-Gerhard Gross. **An industrial case study in reconstructing requirements views**. *Empirical Software Engineering*, **13**(6):727–760, 2008. 3, 22

[43] R. Helm and Y. S. Maarek. **Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries**. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–61, Phoenix, Arizona, USA, 1991. ACM Press. 3, 23

[44] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. **An Information Retrieval Approach to Concept Location in Source Code**. In *Proceedings of 11th Working Conference on Reverse Engineering*, pages 214–223, Delft, The Netherlands, 2004. IEEE CS Press. 3, 23

[45] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. **CVSSearch: searching through source code using CVS comments**. In *Proceedings of 17th IEEE International Conference on Software Maintenance*, pages 364–373, Florence, Italy, 2001. IEEE CS Press. 3, 23

REFERENCES

[46] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. **Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace**. In *Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, Georgia, USA, 2007. ACM Press. 3, 23

[47] G. Antoniol, J.H. Hayes, Y.-G. Gueheneuc, and M. di Penta. **Reuse or rewrite: Combining textual, static, and dynamic analyses to assess the cost of keeping a system up-to-date**. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 147 –156, 2008. 3, 23

[48] Pierre F. Baldi, Cristina V. Lopes, Erik J. Linstead, and Sushil K. Bajracharya. **A theory of aspects as latent topics**. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 543–562, New York, NY, USA, 2008. ACM. 3, 23

[49] Emily Hill, Lori Pollock, and K. Vijay-Shanker. **Automatically capturing source code context of NL-queries for software maintenance and reuse**. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 232–242, 2009. 3, 23

[50] Brendan Cleary, Chris Exton, Jim Buckley, and Michael English. **An empirical analysis of information retrieval based concept location techniques in software comprehension**. *Empirical Software Engineering*, **14**(1):93–130, 2009. 3, 23

[51] F. Asadi, M. Di Penta, G. Antoniol, and Y.-G. Gu andh andneuc. **A Heuristic-Based Approach to Identify Concepts in Execution Traces**. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 31 –40, 2010. 3, 23

[52] Denys Poshyvanyk, Malcom Gethers, and Andrian Marcus. **Concept Location using Formal Concept Analysis and Information Retrieval**. *Transactions on Software Engineering and Methodologies*, **21**(4):To appear, 2012. 3, 23

[53] G. Canfora and L. Cerulo. **Impact Analysis by Mining Software and Change Request Repositories**. In *Proceedings of 11th IEEE International Symposium on Software Metrics*, pages 20–29, Como, Italy, 2005. IEEE CS Press. 3, 23

[54] Emily Hill, Lori Pollock, and K. Vijay-Shanker. **Exploring the neighborhood with dora to expedite software maintenance**. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 14–23, 2007. 3, 23

[55] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. **Using information retrieval based coupling measures for impact analysis**. *Empirical Software Engineering*, **14**(1):5–32, 2009. 3, 23, 24, 29, 34, 35, 47, 58, 89, 91, 125, 157

[56] Malcom Gethers, Huzefa Kagdi, Bogdan Dit, and Denys Poshyvanyk. **An adaptive approach to impact analysis from change requests to source code**. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 540–543, 2011. 3, 23

[57] A. Marcus and J. I. Maletic. **Identification of High-Level Concept Clones in Source Code**. In *Proceedings of 16th IEEE International Conference on Automated Software Engineering*, pages 107–114, San Diego, California, USA, 2001. IEEE CS Press. 3, 23

[58] Scott Grant and James R. Cordy. **Vector space analysis of software clones**. In *Proceedings of International Conference on Program Comprehension*, pages 233–237, 2009. 3, 23

[59] Robert Tairas and Jeff Gray. **An information retrieval process to aid in the analysis of code clones**. *Empirical Software Engineering*, **14**(1):33–56, February 2009. 3, 23

[60] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. **Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems**. *IEEE Transaction on Software Engineering*, **34**(2):287–300, 2008. 3, 8, 23, 24, 34, 51, 125, 180

[61] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. **An Empirical Study on the Developers Perception of Software Coupling**. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE 2013, page To appear., 2013. 3

[62] Jonathan Chang and David M. Blei. **Hierarchical Relational Models for Document Networks**. *Annals of Applied Statistics*, 2010. 4, 39, 117, 121, 148, 188

[63] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, 1998. 4, 8

[64] Martin Fowler. **Refactoring Catalog. http://refactoring.com/catalog/**. 7, 12

[65] James O. Coplien and Neil B. Harrison. *Organizational Patterns of Agile Software Development*. Prentice-Hall, Upper Saddle River, NJ (2005), 1^{st} edition, 2005. 7

[66] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. **Structured Design**. *IBM Systems Journal*, **13**(2):115–139, 1974. 7

[67] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. **Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction.** *IEEE Transactions on Software Engineering*, **31**(10):897–910, 2005. 8

[68] Yixun Liu, Denys Poshyvanyk, Rudolf Ferenc, Tibor Gyimóthy, and Nikos Chrisochoides. **Modeling class cohesion as mixtures of latent topics**. In *Proceedings of 25th IEEE International Conference on Software Maintenance*, pages 233–242, Edmonton, Canada, 2009. IEEE CS Press. 8

[69] Radu Marinescu. **Detection Strategies: Metrics-Based Rules for Detecting Design Flaws**. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society. 9, 17

[70] Padmaja Joshi and Rushikesh K. Joshi. **Concept Analysis for Class Cohesion**. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 237–240, Kaiserslautern, Germany, 2009. 10, 17

[71] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. **A Bayesian Approach for the Detection of Code and Design Smells**. In *Proceedings of the 2009 Ninth International Conference on Quality Software*, pages 305–314, Washington, DC, USA, 2009. IEEE Computer Society. 10

[72] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. **DECOR: A Method for the Specification and Detection of Code and Design Smells**. *IEEE Transactions on Software Engineering*, **36**(1):20–36, January 2010. 10, 185

[73] F. Simon, F. Steinbr, and C. Lewerentz. **Metrics based refactoring**. In *Proceedings of 5th European Conference on Software Maintenance and Reengineering*, pages 30–38, Lisbon, Portugal, 2001. IEEE CS Press. 11, 17

[74] T. A. Wiggerts. **Using Clustering Algorithms in Legacy Systems Remodularization**. In *Proceedings of 4th Working Conference on Reverse Engineering*, page 33, Amsterdam, The Netherlands, 1997. IEEE CS Press. 12

[75] Nicolas Anquetil and Timothy Lethbridge. **Experiments with Clustering as a Software Remodularization Method**. In *Proceedings of 6th Working Conference on Reverse Engineering*, pages 235–255, Atlanta, Georgia, USA, 1999. IEEE CS Press. 12, 13

[76] Brian S. Mitchell and Spiros Mancoridis. **Comparing the Decompositions Produced by Software Clustering Algorithms Using Similarity Measurements**. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 744–753, 2001. 13

[77] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. **Comparison of Cluster-**

ing Algorithms in the Context of Software Evolution. In *Proceedings of 21st IEEE International Conference on Software Maintenance*, pages 525–535, Budapest, Hungary, 2005. IEEE CS Press. 13, 20, 147

[78] Onaiza Maqbool and Haroon A. Babri. **Hierarchical Clustering for Software Architecture Recovery**. *IEEE Transactions on Software Engineering*, **33**(11):759–780, 2007. 13

[79] Spiros Mancoridis, Brian S. Mitchell, C. Rorres, Yih-Farn Chen, and Emden R. Gansner. **Using Automatic Clustering to Produce High-Level System Organizations of Source Code**. In *Proceedings of 6th International Workshop on Program Comprehension*, Ischia, Italy, 1998. IEEE CS Press. 13, 14, 20, 147

[80] Brian S. Mitchell and Spiros Mancoridis. **On the Automatic Modularization of Software Systems Using the Bunch Tool**. *IEEE Transactions on Software Engineering*, **32**(3):193–208, 2006. 13

[81] D. Doval, S. Mancoridis, and B. S. Mitchell. **Automatic Clustering of Software Systems Using a Genetic Algorithm**. In *Proceedings of the Software Technology and Engineering Practice*, STEP '99, pages 73–82. IEEE Computer Society, 1999. 14

[82] Hani Abdeen, Stéphane Ducasse, Houari A. Sahraoui, and Ilham Alloui. **Automatic Package Coupling and Cycle Minimization**. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 103–112, Lille, France, 2009. IEEE CS Press. 14, 20, 75, 93, 110, 147, 161

[83] Mark Harman, Robert M. Hierons, and Mark Proctor. **A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization**. In *Proceedings of the Genetic and Evolutionary Computation Conference*, New York, USA, 2002. Morgan Kaufmann Publishers Inc. 14, 20, 147

[84] Olaf Seng, Markus Bauer, Matthias Biehl, and Gert Pache. **Search-based improvement of subsystem decompositions**. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1045–1051, Washington, Columbia, USA, 2005. ACM Press. 14, 75, 161

[85] Brian S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD thesis, Drexel University, Philadelphia, 2002. 14

[86] Kata Praditwong, Mark Harman, and Xin Yao. **Software Module Clustering as a Multi-Objective Search Problem**. *IEEE Transactions on Software Engineering*, **37**(2):264–282, 2011. 14

[87] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. **Investigating the Use of Lexical Information for Software System Clustering**. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 35–44, 2010. 15

[88] Anna Corazza, Sergio Di Martino, and Giuseppe Scanniello. **A Probabilistic Based Approach towards Software System Clustering**. In *CSMR*, pages 88–96, 2010. 15

[89] J. I. Maletic and A. Marcus. **Supporting Program Comprehension Using Semantic and Structural Information**. In *Proceedings of 23rd International Conference on Software Engineering*, pages 103–112, Toronto, Ontario, Canada, 2001. IEEE CS Press. 15, 91

[90] Giuseppe Scanniello, Anna D'Amico, Carmela D'Amico, and Teodora D'Amico. **Using the Kleinberg Algorithm and Vector Space Model for Software System Clustering**. In *Proceedings of the 18th International Conference on Program Comprehension*, pages 180–189, 2010. 15

[91] 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*, **41**(6):391–407, 1990. 15, 22, 33, 34, 125

[92] A. Kuhn, S. Ducasse, and T. Gîrba. **Semantic Clustering: Identifying Topics**

in Source Code. *Information and Software Technology*, **49**(3):230–243, 2007. 15, 72, 103

[93] Jon M. Kleinberg. **Authoritative Sources in a Hyperlinked Environment**. *Journal of the ACM*, **46**(5):604–632, 1999. 16

[94] J. A. Hartigan. *Clustering Algorithms*. Wiley, 1975. 16

[95] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. **Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships**. In *Proceedings of the 23rd International Conference on Software Maintenance*, pages 94–103, 2007. 16

[96] B. Du Bois, S. Demeyer, and J. Verelst. **Refactoring - improving coupling and cohesion of existing code**. In *Proceedings of 11th Working Conference on Reverse Engineering*, pages 144–151, Delft, the Netherlands, 2004. IEEE CS Press. 17

[97] D. C. Atkinson and T. King. **Lightweight Detection of Program Refactorings**. In *Proceedings of 12th Asia-Pacific Software Engineering Conference*, pages 663–670, Taipei, Taiwan, 2005. IEEE CS Press. 17

[98] T. Bodhuin, G. Canfora, and L. Troiano. **SORMASA: A tool for suggesting model refactoring actions by metrics-led genetic algorithm**. In *Proceedings of 1st Workshop on Refactoring Tools*, pages 23–24, Berlin, Germany, 2007. 18

[99] W. G. Griswold and D. Notkin. **Automated assistance for program restructuring**. *ACM Transactions on Software Engineering and Methodologies*, **2**(3):228–269, 1993. 20, 147

[100] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006. 20, 147, 180

[101] Oscar Nierstrasz, Stéphane Ducasse, and Serge Demeyer. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., 2003. 20, 147

[102] A. Abadi, R. Ettinger, and Y. A. Feldman. **Fine slicing for advanced method extraction**. In *3rd Workshop on Refactoring Tools*, 2009. 21

[103] E. Van Emden and L. Moonen. **Java Quality Assurance by Detecting Code Smells**. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 97–106, Washington, DC, USA, 2002. IEEE Computer Society. 22

[104] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. **An Information Retrieval Approach for Automatically Constructing Software Libraries**. *IEEE Transactions on Software Engineering*, **17**(8):800–813, 1991. 22

[105] B. Fischer. **Specification-based browsing of software component libraries**. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 74 –83, oct 1998. 22

[106] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999. 22, 31, 32, 33, 95, 120, 151

[107] K. Thulasiraman and M. N. S. Swamy. *Graphs: theory and algorithms*. John Wiley & Sons, Inc., 1992. 26

[108] YS. Lee, BS. Liang, SF. Wu, and FJ. Wang. **Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow**. In *Proceedings of International Conference on Software Quality*, pages 81–90, Maribor, Slovenia, 1995. 29, 58, 89, 91, 151

[109] W. Li and S. Henry. **Maintenance metrics for the object oriented paradigm**. In *Proc. of METRICS*, pages 52–60, 1993. 29, 51, 125, 126, 156, 157, 180

[110] 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*, pages 18–21, Shanghai, China, 2006. ACM Press. 30, 47, 121

[111] K. Sparck Jones. **A statistical interpretation of term specificity and its application in retrieval**. *Journal of Documentation*, **28**:11–21, 1972. 32

[112] G. Salton, A. Wong, and C. S. Yang. **A vector space model for information retrieval**. *Communications of the ACM*, **18**(11):613–620, 1975. 33

[113] J. K. Cullum and R. A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, **1**, chapter Real rectangular matrices. Birkhauser, Boston, 1998. 34

[114] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. **Latent dirichlet allocation**. *The Journal of Machine Learning Research*, **3**:993–1022, 2003. 40

[115] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. **A Two-Step Technique for Extract Class Refactoring**. In *Proceedings of 25th IEEE International Conference on Automated Software Engineering*, pages 151–154, 2010. 43, 112, 121, 187

[116] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. **Automating Extract Class Refactoring: an Improved Method and its Evaluation**. *Empirical Software Engineering*, page To appear, 2013. 43, 187

[117] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. **Template-based reconstruction of complex refactorings**. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010*, pages 1–10. IEEE Computer Society, 2010. 45, 78, 190

[118] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. **Automating Extract Class Refactoring: an Improved Approach and its Evaluation. Online Appendix. https://dl.dropbox.com/u/20652688/emseAppendix.zip**, 2012. 46, 53, 54, 55, 67, 70, 78

[119] Rainer Koschke, Gerardo Canfora, and Jörg Czeranski. **Revisiting the Delta IC approach to component recovery**. *Science of Computer Programming*, **60**(2):171–188, 2006. 48

[120] Zhihua Wen and Vassilios Tzerpos. **An Effectiveness Measure for Software Clustering Algorithms**. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, IWPC '04, pages 194–203. IEEE Computer Society, 2004. 54, 80

[121] J. Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates, 2nd edition edition, 1988. 58

[122] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998. 60, 65, 137, 165

[123] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 26 (Maximum Flow). MIT Press and McGraw-Hill, 2nd edition, 2001. 60

[124] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. **A Bayesian Approach for the Detection of Code and Design Smells**. In *Proceedings of the 9th International Conference on Quality Software*, pages 305–314, Hong Kong, China, 2009. IEEE CS Press. 63, 74

[125] K. J. Stewart, D. P. Darcy, and S. L. Daniel. **Opportunities and challenges applying functional data analysis to the study of open source software evolution**. *Statistical Science*, **21**(2):167–178, 2006. 63

[126] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers, 1992. 64, 96, 135

[127] Kata Praditwong, Mark Harman, and Xin Yao. **Software Module Clustering as a Multi-Objective Search Problem**. *IEEE Transactions on Software Engineering*, **37**(2):264–282, 2011. 75, 161

[128] E. Arisholm and D.I.K. Sjoberg. **Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software**. *IEEE Transactions on Software Engineering*, **30**(8):521–534, 2004. 75, 144

[129] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. **Software Re-Modularization Based on Structural and Semantic Metrics**. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 195–204, Beverly, MA, USA, 2010. IEEE CS Press. 89

[130] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. **Using Structural and Semantic Measures to Improve Software Modularization**. *Empirical Software Engineering*, page To appear, 2013. 89, 188

[131] Andrea De Lucia, Rocco Oliveto, and Luigi Vorraro. **Using structural and semantic metrics to improve class cohesion**. In *Proceedings of 28th International Conference on Software Maintenance*, pages 27–36, Beijing, China, 2008. IEEE CS Press. 91

[132] Roberto Almeida Bittencourt and Dalton Dario Serey Guerrero. **Comparison of Graph Clustering Algorithms for Recovering Software Architecture Module Views**. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 251–254, Washington, DC, USA, 2009. IEEE Computer Society. 92

[133] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm*. John Wiley and Sons, 1994. 92

[134] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. **Software Re-Modularization based on Structural and Semantic Metrics**. Technical report, University of Salerno, http://www.sesa.dmi.unisa.it/TR2011_EMSE.pdf, 2011. 97, 98, 99, 113, 114, 115

[135] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. **Improving IR-based Traceability Recovery Using Smoothing Filters**. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*, pages 21–30, 2011. 106

[136] R. K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, 3rd edition, 2003. 110, 132, 144

[137] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. **Playing with refactoring: Identifying extract class opportunities through game theory**. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–5, 2010. 112, 190

[138] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia. **Identifying Method Friendships to Remove the Feature Envy Bad Smell**. In *Proceedings of the 33rd IEEE/ACM International Conference on Software Engineering*, Hawaii, USA, 2011. ACM Press. 117, 188

[139] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. **Methodbook: Recommending Move Method Refactorings via Relational Topic Models**. *Transactions on Software Engineering*, page To appear, 2013. 117, 188

[140] 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, 2011. IEEE CS Press. 120, 151

[141] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991. 123

[142] Lionel C. Briand, John W. Daly, and Jürgen Wüst. **A Unified Framework for Cohesion Measurement in Object-OrientedSystems**. *Empirical Software Engineering.*, **3**:65–117, July 1998. 125

[143] M. Gethers and D. Poshyvanyk. **Using Relational Topic Models to capture coupling among classes in object-oriented software systems**. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1 –10, 2010. 133

REFERENCES

[144] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. **Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies**. *Transactions on Software Engineering and Methodologies*, page To appear., 2013. 147, 188

[145] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. **Does code decay? Assessing the evidence from change management data**. *IEEE Transactions on Software Engineering*, **27**(1):1–12, 2001. 147

[146] Edward Yourdon and Larry Constantine. *Structured Design: Foundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, 1979. 156, 162

[147] Roger Pressman. *Software Engineering: A Practitioner's Approach. 3rd Edition*. McGraw-Hill, 1992. 156, 162

[148] Iann Sommerville. *Software Engineering. 6th Edition*. Addison-Wesley, 2001. 156, 162

[149] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Fabio Palomba. **Supporting extract class refactoring in Eclipse: The ARIES project**. In *ICSE*, pages 1419–1422, 2012. 179, 189

[150] Gabriele Bavota, Filomena Carnevale, Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. **Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization**. In *SSBSE*, pages 75–89, 2012. 189

[151] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. **When does a Refactoring Induce Bugs? An Empirical Study**. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM 2012)*, pages 104–113, 2012. 190

[152] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. **When do changes induce fixes?** In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*. ACM, 2005. 190

[153] Sunghun Kim, E. James Whitehead Jr., and Yi Zhang. **Classifying Software Changes: Clean or Buggy?** *IEEE Transactions on Software Engineering*, **34**(2):181–196, 2008. 190