A

Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies

GABRIELE BAVOTA, University of Salerno, Italy
MALCOM GETHERS, University of Maryland, Baltimore County, USA
ROCCO OLIVETO, University of Molise, Italy
DENYS POSHYVANYK, The College of William and Mary, USA
ANDREA DE LUCIA, University of Salerno, Italy

Oftentimes, during software maintenance the original program modularization decays, thus reducing its quality. One of the main reasons for such architectural erosion is suboptimal placement of source code classes in software packages. To alleviate this issue, we propose an automated approach to help developers improve the quality of software modularization. Our approach analyzes underlying latent topics in source code as well as structural dependencies to recommend (and explain) refactoring operations aiming at moving a class to a more suitable package. The topics are acquired via Relational Topic Models (RTM), a probabilistic topic modeling technique. The resulting tool, coined as $R3$ (Rational Refactoring via RTM), has been evaluated in two empirical studies. The results of the first study conducted on nine software systems indicate that $R3$ provides a coupling reduction from 10% to 30% among the software modules. The second study with 62 developers confirms that $R3$ is able to provide meaningful recommendations (and explanations) for move class refactoring. Specifically, more than 70% of the recommendations were considered meaningful from a functional point of view.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Documentation, Management

Additional Key Words and Phrases: Software Modularization, Refactoring, Relational Topic Modeling, Empirical Studies, Recommendation System

1. INTRODUCTION

In the software life-cycle the change is the rule and not the exception [Lehman 1980]. A key point for sustainable program evolution is to tackle software complexity. In Object-Oriented (OO) systems, classes are the primary decomposition mechanism, which group together data and operations to reduce complexity. Higher level programming constructs, such as packages, group semantically and structurally related classes aiming at supporting the replacement of specific parts of a system without impacting the complete system. A well modularized system eases the understanding, maintenance, test, and evolution of software systems [DeRemer and Kron 1976].

This work is supported in part by NSF CCF-1016868, NSF CCF-0916260, and NSF CCF-1218129 awards. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.
Author's addresses: Gabriele Bavota and Andrea De Lucia, University of Salerno, Fisciano (SA), Italy; Rocco Oliveto, University of Molise, Pesche (IS), Italy; Malcom Gethers, University of Maryland, Baltimore County, Baltimore, MD 21250, USA; Denys Poshyvanyk, The College of William and Mary, Williamsburg, VA 23185, USA.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1539-9087/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

During maintenance, the structural design of the software system evolves and changes are not always performed following OO guidelines [Eick et al. 2001; Fowler 1999]. 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 reasons for such an architectural erosion is inconsistent placement of source code classes in software packages [Fowler 2000]. Such a scenario, on one hand negatively impacts the package cohesion and on the other hand increases the number of dependencies (coupling) between packages [Lanza and Marinescu 2006].

In such cases, re-modularization of the system is necessary [Nierstrasz et al. 2003; Fowler 1999]. 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., [Harman et al. 2002; Mancoridis et al. 1998; Wu et al. 2005]). 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 [Fowler 1999], 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 proposed approach analyzes underlying latent topics (natural language 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]*. The topics are acquired via Relational Topic Models (RTM) [Chang and Blei 2010], a probabilistic topic modeling technique, recently used to capture coupling among classes in OO software systems [Gethers and Poshyvanyk 2010] and to support move method refactoring [Oliveto et al. 2011]. In this paper, we utilize RTM 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 $R3$ (Rational Refactoring via RTM), has been evaluated in two empirical studies. In the first study we analyzed the ability of $R3$ 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 $R3$ 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. To the best of our knowledge the user study reported in this paper with a total of 62 participants is the largest study carried out to evaluate refactoring operations from a functional point of view. This represents an important contribution of this paper as a recent survey of the refactoring literature reports apparent lack of this type of evaluation [Praditwong et al. 2011].

Summarizing, the specific contributions of this paper are:

— The definition of a novel approach for identifying eligible move class refactoring operations based on RTM. The approach is based on both structural and semantic information extracted from the source code. R3 represents the first recommendation system for improving software modularization that generates explanations for refactoring operations.
— An assessment of the proposed approach on nine software systems to verify if the refactoring suggestions proposed by $R3$ are able to reduce coupling among software modules.
— An evaluation of the quality of the suggested refactoring operations from the (i) external developers' perspective on an open-source software system, i.e., JHotDraw, and (ii) original developers' perspective on four software systems, i.e., eTour, GESA, SESA and SMOS.

The rest of the paper is organized as follows. Section 2 discusses the related literature, while Section 3 presents the details behind $R3$. Section 4 reports the first case study where $R3$ has been evaluated via quality measures, while Section 5 reports the results of the study with users. Finally, Section 6 provides concluding remarks and future work.

2. RELATED WORK

A lot of effort has been devoted to the definition of automatic and semi-automatic approaches aimed at supporting software engineers in the re-modularization of software systems. Since the 80's, many authors investigated how to increase the quality of procedural programs, in terms of maintainability, reusability, and high level design, by restructuring the software architecture.

Many approaches have been proposed to aggregate procedures with high functional cohesion [Cimitile and Visaggio 1995; Shaw et al. 2003; Antoniol et al. 2001]. Most of these approaches are based on identifying strongly connected sub-graphs in the call graph representing the program. Cimitile and Visaggio [Cimitile and Visaggio 1995] proposed a technique based on dominance trees to aggregate procedures in reusable modules. An improvement to such a technique has been proposed by Shaw *et al.* [Shaw et al. 2003] to support program comprehension. Antoniol *et al.* [Antoniol et al. 2001] proposed the use of concept analysis to restructure the architectural source code files organization of legacy systems.

Other techniques have been proposed for the identification of objects or Abstract Data Types (ADTs) in legacy systems. Such approaches generally identify objects or ADTs in legacy code exploiting the relations existing between program routines and global variables and/or user defined data types ([Canfora et al. 2001; Koschke et al. 2006; Tonella 2001; van Deursen and Kuipers 1999] are some of the most recent works). Using a similar approach, Fanta and Rajlich [Fanta and Rajlich 1999] presented a tool-set to encapsulate new classes in procedural C code.

Regarding software re-modularization, most of the existing approaches are based on clustering techniques [Lethbridge and Anquetil 2002]. Wiggerts [Wiggerts 1997] provides the theoretical background for the application of cluster analysis in systems re-modularization. They discuss on how to establish similarity criteria between the entities to cluster and provide the summary of possible clustering algorithms to use in system re-modularization. Anquetil and Lethbridge [Anquetil and Lethbridge 1999] tested some of the algorithms proposed by Wiggerts and compared their strengths and weaknesses when applied to system re-modularization. A more recent work by Shtern and Tzerpos [Shtern and Tzerpos 2009] introduced a method for selecting a clustering algorithm for the system decomposition given specifics needs. Wu *et al.* [Wu et al. 2005] describe a comparative study of clustering algorithms in the context of

software evolution. Their results show that the analyzed clustering algorithms are not ready to be widely adopted for large systems. Maqbool and Babri [Maqbool and Babri 2007] focus on the application of hierarchical clustering in the context of software architecture recovery and modularization. They investigate the measures to use in this domain, categorizing various similarity and distance measures into families according to their characteristics. The re-modularization of software systems was also addressed using concept analysis techniques [Tonella 2001; van Deursen and Kuipers 1999], that provide a way to identify groups of objects that have common attributes.

Mancoridis *et al.* [Mancoridis et al. 1998] proposed an automatic technique to create a high-level view of the system organization. They introduced a search-based approach to identify the organization of a software system. Mitchell and Mancoridis [Mitchell and Mancoridis 2006] use the same technique in Bunch, a tool supporting automatic system decomposition. Search-based approaches are also used in several other works [Harman et al. 2002; Praditwong et al. 2011; Seng et al. 2005; Abdeen et al. 2009]. In particular, Harman *et al.* [Harman et al. 2002] and Seng *et al.* [Seng et al. 2005] use a single-objective genetic algorithm to improve the subsystem decomposition of a software system, where the fitness function is defined using a combination of quality metrics. Praditwong *et al.* [Praditwong et al. 2011] also uses genetic algorithms but exploit a multi-objective fitness function. Abdeen *et al.* [Abdeen et al. 2009] proposed a heuristic search-based approach for automatically reducing the dependencies between the packages of a software system. Starting from an initial decomposition, their technique optimizes the existing package structure by moving classes between the original packages. To the best of our knowledge this is the closest approach to $R3$. However, $R3$ exploits not only structural information to derive refactoring operations, but also conceptual information derived from identifiers and comments. In addition, $R3$ is the first recommendation system for move class refactoring able to provide an evaluation of the proposed remodularization based on quantitative (that is, confidence level) and qualitative data (that is, the rationale behind the proposed remodularization).

The combined use of conceptual and structural measures to suggest remodularization is one of the main characteristics of our approach. Most of the remodularization approaches in the literature exploit information derived only from structural metrics. However, a lot of important information, such as design decisions and rationale, is embedded in the comments and identifiers in source code classes. Other remodularization approaches also exploit conceptual (or semantic) information in addition to structural information [Bavota et al. 2010; Maletic and Marcus 2001; Kuhn et al. 2007; Scanniello et al. 2010; Corazza et al. 2010; Corazza et al. 2011]. Bavota *et al.* [Bavota et al. 2010] used graph theory to identify extract package refactoring operations. This kind of refactoring is used to solve the problem of having a Promiscuous Package in a system, i.e., a package grouping together several responsibilities that should be grouped in different packages. Their approach splits a package (manually identified by a developer as a Promiscuous Package) in new, more cohesive packages, trying to group together classes having similar responsibilities. To this aim, conceptual (i.e., textual overlap) and structural (i.e., method calls) relationships between the classes in the package are taken into account. Note that, on the contrary to $R3$, the approach proposed by Bavota et al. [Bavota et al. 2010] totally ignores the existence of the other packages in the system, i.e., it is not able to move classes among the system packages, but only focuses on the Promiscuous Package to decompose it in new more cohesive packages. On the contrary, $R3$ can be used to automatically analyze all the packages in the system, identifying incorrectly placed classes, and moving them among the system packages, however, it is not able to create new packages. Thus, the two approaches solve different design problems. Scanniello *et al.* [Scanniello et al. 2010] combine structural and semantic information to recover the architecture of ob-

ject oriented systems with a hierarchical structure; structural information is used to identify software layers, while lexical information is employed to partition each identified layer into software modules. Maletic and Marcus [Maletic and Marcus 2001] exploit the combination of semantic and structural measures to identify Abstract Data Types in legacy code while Kuhn *et al.* [Kuhn et al. 2007] broadened the work by Maletic and Marcus by providing a visual notation that gives an overview of all the clusters and their semantic relationships. Corazza *et al.* [Corazza et al. 2010; Corazza et al. 2011] presented a clustering based approach to partition object-oriented systems into subsystems. In particular, they extracted lexical information from the source code and then used a partitioning algorithm [Corazza et al. 2010], i.e., K-Medoids, or a hierarchical algorithm [Corazza et al. 2011], i.e., Hierarchical Agglomerative Clustering, to build subsystems containing semantically related classes. Recently, Pashov et al. [Pashov et al. 2004] presented an approach that uses domain information to assist detection of architecture disproportions and redundancies within a legacy system. The proposed approach takes system features structured in a corresponding model as an input, analyses them, and produces a set of clues and hints showing potential architectural problems and possible solutions to solve these.

It is worth noting that the focus of our approach is different with respect to the aforementioned works. In fact, our approach focuses on how to improve the re-modularization of a software system by moving classes between the original packages, without proposing a whole new re-modularization that could potentially affect developers' understandings of the system decomposition.

Finally, previous applications of topic models (and RTM in particular) in software engineering deserve space in our related work section. One of the first applications of topic models in software engineering was focused on detecting cross-cutting concerns (aspect candidates) in large software repositories [Baldi et al. 2008]. Baldi et al. were also the first to propose entropy as a measure of scattering of cross-cutting concerns [Baldi et al. 2008]. This work motivated a number of other approaches and applications of topic models in software engineering [Liu 2009; Savage et al. 2010; Oliveto et al. 2010; Chen et al. 2012; Thomas et al. 2011; 2010; Bajracharya and Lopes 2009; Grant et al. 2012; Hindle et al. 2009], including the one presented in this paper. RTM, which was proposed as an extension of LDA, has been also applied to capture coupling among classes in OO software systems [Gethers and Poshyvanyk 2010], to support traceability link recovery [Gethers et al. 2011], and to identify move method refactoring opportunities [Oliveto et al. 2011]. The latter approach (called Methodbook) shares some similarities with the approach $R3$ proposed in this paper concerning the underlying algorithmic technique and the use of RTM. However, they have been developed to support different refactoring operations. In fact, while the aim of Methodbook is to suggest move method refactoring operations, $R3$ supports re-modularization through move class refactoring operations. Move method and move class refactoring have different levels of granularity (method vs. class) and are used to solve different problems. In particular, move method refactoring is used to solve the Feature Envy Bad Smell, occurring when a method uses more features of another class than the class in which it is defined and implemented, while move class refactoring is used when a class belongs to a package that groups responsibilities unrelated to those implemented by the class. Clearly, working at different granularity levels, $R3$ and Methodbook also exploit different structural metrics: Methodbook exploits two method-level structural metrics, while $R3$ uses the Information-Flow-based Coupling (ICP) [Lee et al. 1995] between classes. Finally, it is worth noting that $R3$ provides support to software developers in evaluating the goodness of suggested refactoring operations by generating explanations or rationale for suggested operations using topic analysis. The latter unique feature differentiates $R3$ from all the other refactoring approaches.

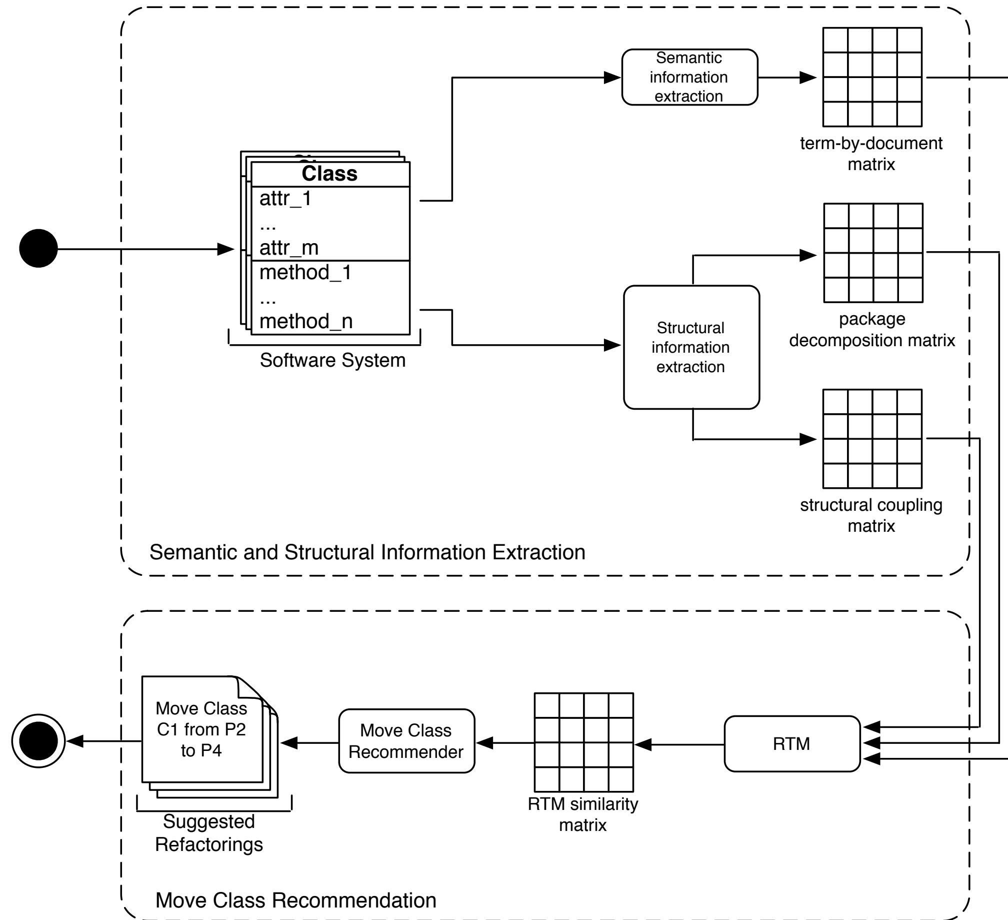

Fig. 1: Identifying move class refactoring with $R3$.

3. *R3*: RATIONAL REFACTORING VIA RTM

We propose an approach, namely $R3$, 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 we are able to identify possible move class refactoring opportunities (i.e., 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 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, $R3$ also exploits static analysis to (i) derive dependencies among classes (stored in the *structural*

Algorithm 1 *R3*: Rational Refactoring via RTM

```
##Procedure to identify potential move class refactoring opportunities
##based on analysis of underlying latent topics in source code as well as
##dependencies between source code entities
##S: a software system to be analyzed
procedure R3(S)
   ##Extract words contained in comments, identifiers, and string
   ##literals for each class in the software system
   SemInfo ← ExtractSemanticInfo(S)
   ##Light-weight static analysis to detect dependencies (i.e., method
   ##calls) between classes
   StrucInfo ← ExtractStructuralInfo(S)
   ##Light-weight static analysis to extract package decomposition
   PkgInfo ← ExtractPkgDecom(S)
   ##Generate term-by-document matrix using information extracted from
   ##software system
   Docs ← GenerateDocs(SemInfo)
   ##Generate links using dependencies between classes and package
   ##decomposition information
   Links ← GenerateLinks(StrucInfo, PkgInfo)
   ##Create an RTM representation of the software system
   RTMModel ← RTM(Docs, Links)
   ##Identify move class refactoring opportunities from the set of
   ##classes in the software system
   for all c ∈ C do
      ##Determine the five classes most similar to the current class c by
      ##analyzing the RTM representation of the system
      RelavantCls ← IdenRelCls(RTMModel, c, C, 5)
      ##Select the package which contains the highest number of most
      ##similar classes and also compute the confidence level
      Pkg, CL ← IdenPkg(RelavantCls)
      ##Extract topic information from RTMModel for the given class
      ClsTopics ← GetTopics(RTMModel, c)
      ##Extract topic information from RTMModel for the given package
      PkgTopics ← GetTopics(RTMModel, Pkg)
      ##Generate a rationale based on the topics which the user can use
      ##to understand the relationship between the class and the package
      Rat ← GenerateRationale(ClsTopics, PkgTopics)
      ##Store information related to refactoring suggestions in
      ##RefactoringInfo
      RefactoringInfo ← SaveRefInfo(c, Pkg, Rat, CL)
   end for
   ##Return to the user all refactoring opportunities and related
   ##information
   return RefactoringInfo
end procedure
```

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 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 $R3$ 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. Algorithm 1 reports the pseudo-code of $R3$ while in the next subsections we detail on all the steps behind it.

3.1. 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 [Dit et al. 2011]. 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 1). A generic entry $w_{i,j}$ of this matrix denotes a measure of the weight (i.e., relevance) of the ith term in the jth document. In order to weight the relevance of a term in a document we employ the *tf-idf* weighting schema [Baeza-Yates and Ribeiro-Neto 1999]. The *term-by-document matrix* weighted with the *tf-idf* schema represents a common model for representing conceptual information, that has been previously used to support different software maintenance tasks (see e.g., [Gethers and Poshyvanyk 2010; Oliveto et al. 2011; Marcus and Poshyvanyk 2005; Poshyvanyk and Marcus 2006]).

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, other-

wise it is equal to 0. Concerning the dependencies among the classes of the system, we capture them using the Information-Flow-based Coupling (ICP) [Lee et al. 1995] 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. Similarly to the majority of coupling metrics in the literature, this metric is defined at the system level, i.e., they count for a given class C all method calls between C and all the other classes in the system. For our approach we need to redefine ICP to take into account coupling between a pair of classes. We use the ICP metric as redefined by Poshyvanyk et al. [Poshyvanyk et al. 2009]; 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:

$$ICP_{C_i \to C_j} = \sum_{k=1}^{|calls(C_i,C_j)|} p(call(C_i, C_j)_k)$$

where $p(call(C_i, C_j)_k)$ is the number of parameters in the kth call from C_i to C_j. Thus, the generic entry $c_{i,j}$ of the *calls interaction matrix* is computed as $ICP_{C_i \to C_j}$.

3.2. 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[1] to generate a topic distribution model (see Figure 1). RTM[2] is a statistical topic modeling technique [Chang and Blei 2010], originally used in the area of natural language processing, for representing and analyzing textual documents and relationships among them. The basic idea behind RTM is that textual documents (that is, source code classes represented by 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 [Chang and Blei 2010]. 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. Note that RTM can take as input multiple sources of "explicit relationships". In our approach, these 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. The similarities are obtained using RTM's link probability function. The function determines the strength of the relationship between two documents based on the topic distributions (see [Chang and Blei 2010] for details). 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 1).

RTM is one of the few available approaches that can be used for integrated modeling of structured and unstructured information in software. The ability to model both class content (identifiers, comments, and string literals) and relationships among them (dependencies and original design), makes RTM an ideal underlying mechanism to support move class refactoring. More details behind RTM can be found in Appendix A, while examples of applying RTM in other software engineering contexts appear in the research literature [Gethers and Poshyvanyk 2010; Oliveto et al. 2011; Gethers

[1]The implementation of RTM used in this study was developed by the authors of [Chang and Blei 2010] and can be download at http://cran.r-project.org/web/packages/lda/
[2]The interested reader can find more details about RTM in Appendix A.

et al. 2011]. To allow replications, the configuration of the RTM parameters used in $R3$ can be found in Appendix A.1. However, it is worth noting that the problem of tuning up parameters for topic models has been recently solved to support specific software engineering tasks, such as feature location, traceability link recovery, and source code summarization [Panichella et al. 2013]. This technique can be easily extended to configure hyper-parameters of RTM on specific software repositories.

3.3. Identifying Move Class Refactoring Opportunities

$R3$ 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 evaluated the performances of $R3$ using different cut-points by manual inspection, verifying when $R3$ provided meaningful recommendations. In particular, we run R3 on a system[3] that two of the authors developed in the past. The same two authors performed the manual inspection. We experimented with cut-points of 1, 3, 5, 7, and 10, finding that the recommendations are becoming more meaningful when using 5 as cut-point. $R3$ 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. When designing $R3$ we experimented a more sophisticated solution by suggesting as target package the one containing the highest percentage of most similar classes of the class under analysis. In particular, we selected the top similar classes (also in this case trying different cut-points) for the class under analysis and then we identified as target package the one containing the highest percentage of these classes (and not the highest number, as done in the final version of $R3$). However, since we did not find significant differences between the performances of $R3$ adopting this more complicated heuristic, we adopted the simplest solution. 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. Note that, also the μ parameter could be set by adopting a more sophisticated approach like the one presented in [Panichella et al. 2013].

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[4]. Given the textual information extracted from this class as well as a list of other classes, which are structurally connected to *ActionExportProfileXMI*, $R3$ 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 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, $R3$ 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 $R3$, 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

[3]This system has not been used in the evaluation of $R3$ presented in the following sections.
[4]http://argouml.tigris.org/ verified on 5/25/2012

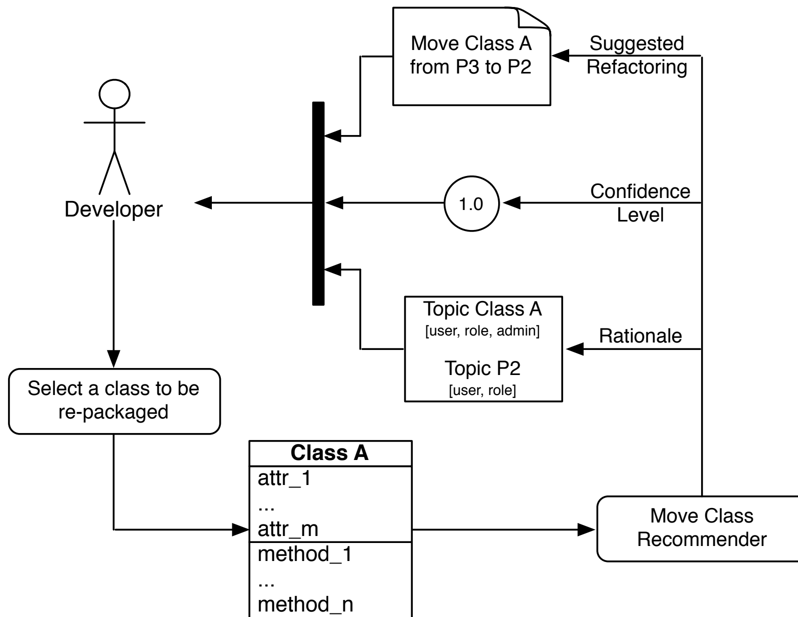

Fig. 2: Interaction between $R3$ and the software engineer.

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[5], also support the recommendation by $R3$. 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 $R3$ to make suggestions that both improve software quality from the perspective of structural and conceptual metrics.

3.4. Putting Software Developers in the Loop

While $R3$ 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 $R3$ to identify if there are any other more suitable packages for this class. Alternatively, the whole system can be used as an input to $R3$ 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, $R3$ provides an evaluation and an explanation behind the recommended refactoring operation (see Figure 2). This evaluation is provided in the form of a confidence level, while the explanation is based on qualitative data extracted via topic analysis.

[5]http://argouml-stats.tigris.org/nonav/javadocs/javadocs-0.32/ verified on 5/25/2012

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 to the same package, *org.argouml.profile*. Thus, the suggestion has the lowest uncertainty ($H(X) = 0$) and, consequently, the highest confidence (*confidenceLevel* = 1). Contrary, if each of the top five similar class comes from a different package, then the uncertainty of the suggestion is the highest possible ($H(X) = 1$), leading to the lowest confidence level (*confidenceLevel* = 0).

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. The topics for a generic package P_i are generated by RTM by considering all the classes contained in P_i as a single document from which extracting the topics. 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 3.3, 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 $R3$.

4. SOFTWARE METRICS EVALUATION

One widely accepted rule to increase the maintainability of software systems is to pursue low coupling among the software modules [Yourdon and Constantine 1979; Pressman 1992; Sommerville 2001]. The *goal* of our first case study is to (i) verify whether the move class operations suggested by $R3$ 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[6], jEdit[7], JHotDraw[8], and jVLT[9], are open-source projects, two are industrial projects, namely eXVantage[10], GESA[11], and three, eTour, SESA, and SMOS, have been developed by different teams of Master students of the University of Salerno in the context of an Advanced Software Engineering course. GanttProject is a cross-platform desktop tool for project scheduling and management. jEdit is a text editor for programmers that provides syntax highlighting and native support for over 130 file formats. JHotDraw is a Java GUI framework for structured drawing editors, while jVLT is a vocabulary learning tool. eXVantage is a product line of eXtreme Visual-Aid Novel Testing and Generation tools, focuses on providing code coverage information to software developers and testers. GESA automates the most important activities in the management of university courses, i.e., timetable creation, classroom allocation. It has been deployed and used at the University of Molise since 2007. 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. eTour is an electronic touristic guide while 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. Table I reports the size, in terms of KLOC, number of classes, and number of packages, and the versions of the systems. Moreover, Table I 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) [Li and Henry 1993] 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

[6] http://www.ganttproject.biz/ verified on 09/09/2011
[7] http://www.jedit.org/ verified on 09/09/2011
[8] http://www.jhotdraw.org/ verified on 09/09/2011
[9] http://jvlt.sourceforge.net/ verified on 09/09/2011
[10] http://www.research.avayalabs.com/ verified on 09/09/2011
[11] http://www.distat.unimol.it/gesa/ verified on 09/09/2011

Table I: Software systems used in the case study

System	KLOC	Classes	Packages	$StructuralCoupling$ Mean	$StructuralCoupling$ Median	$StructuralCoupling$ St. Dev.	$SemanticCoupling$ Mean	$SemanticCoupling$ Median	$SemanticCoupling$ 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	-	-	-	-	-	-

of a class to methods in other classes, and therefore the dependency of local methods to methods implemented by other classes. It has been demonstrated that the MPC directly correlates with the maintenance effort [Li and Henry 1993]. 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) [Poshyvanyk et al. 2009] 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.

4.1. 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[12]. The execution of $R3$ was quite fast, ranging from the 3 minutes needed on the SESA system up to the 17 minutes required for JEdit. 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 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 highest confidence level (see Section 3). 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 by a decreasing confidence level allows to easily analyze if there is a correlation between the confidence level of the suggested refactoring operations and increase/decrease of coupling in the system. In particular, the confidence level might be a good indicator for the goodness

[12] $R3$ was applied on each system in isolation.

Table II: 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	$C_1, C_2, C_3, C_4, C_5 \in P_1$	$\frac{5}{5}, \frac{0}{5}, \frac{0}{5}, \frac{0}{5}, \frac{0}{5}$

Table III: 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%

of $R3$ recommendations in case we 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 II. For example, if all the top 5 most similar classes belong 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 belong to the same package, the entropy will be 0 and thus, the confidence level will be 1.

4.2. Experiment results

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

Table III 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 IV shows the changes in terms of structural and semantic coupling

Table IV: 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

Table V: 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

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

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

Table VI: 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

4.3. Threats to validity

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

4.3.1. Employed quality metrics. In our study we measured the increase/decrease in coupling provided by the move class operations suggested by $R3$ 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. [Praditwong et al. 2011; Seng et al. 2006]), we have intentionally chosen quality metrics that are not exploited by $R3$ to suggest move class operations ($R3$ 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 [Praditwong et al. 2011; Seng et al. 2006; Abdeen et al. 2009; O'Keeffe and O'Cinneide 2006; Seng et al. 2005]). 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 5.

4.3.2. 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 [Yourdon and Constantine 1979; Pressman 1992; Sommerville 2001], 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 VI 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.

5. EVALUATING *R3* WITH SOFTWARE DEVELOPERS

In our previous case study (Section 4) 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[13]. The first study was conducted on JHotDraw and involved 48 developers, i.e., 29 computer science Master's students from the University of Salerno, 7 computer science Master's students, 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 authors 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?

5.1. Evaluation with External Developers

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

5.1.1. Planning. 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

[13]The materials used in these studies are available for replication purposes [Bavota et al. 2012].

1 Analyze class **FigureAttributeConstant** and indicate for each package whether or not the package has the right responsibility for containing the class.

FigureAttributeConstant	org.jhotdraw.framework	org.jhotdraw.figures	org.jhotdraw.util
	YES ☐ MAYBE ☐ NO ☐	YES ☐ MAYBE ☐ NO ☐	YES ☐ MAYBE ☐ NO ☐
Topic1: [constant, map, entries] Topic2: [font, area, style] Topic3: [applica, service, align]	Topic1: [constant, layer, remove] Topic2: [change, handle, check] Topic3: [connect, locate, mous] Topic4: [active, find, insert] Topic5: [implement, found, start]	Topic1: [connect, active, decor] Topic2: [constant, image, holder] Topic3: [font, angle, type] Topic4: [active, find, insert]	Topic1: [format, storage, point2d] Topic2: [stream, wrap, filter] Topic3: [active, image, storable] Topic4: [command, next store]
	AGREE ☐ NEUTRAL ☐ DISAGREE ☐	AGREE ☐ NEUTRAL ☐ DISAGREE ☐	AGREE ☐ NEUTRAL ☐ DISAGREE ☐

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

the result of our first case study (Section 4) 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 Figure 3 for an excerpt of the questionnaire and [Bavota et al. 2012] 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[14]. Note that the participants 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 3, 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.

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

Table VII: Developers' answers in different scenarios.

Scenario	Original package			$R3$ 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 VIII: 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

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 [Conover 1998] 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 $\alpha = 0.05$.

5.1.2. Analysis of the Results. In order to respond to our first research question ($\mathbf{RQ}_1$), Table VII 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 $R3$ suggestions with high confidence level. In this case, the analysis of the results provided by the participants reveals that $R3$'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 $R3$ instead of the original package, i.e., they marked the package suggested by $R3$ 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 $R3$ were not considered as a good alternative (36% YES + 28% MAYBE).

All these considerations are also supported by the statistical analysis. Table VIII reports the results of the Mann-Whitney tests used to compare the ranking of packages in different sets, i.e., original, suggested by $R3$, and random. As we can see, the only case where the original packages did not obtain a statistically significant higher score than the packages suggested by $R3$ is when the confidence level is high. This confirms that in such a scenario the recommendation by $R3$ 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 $R3$.

Concerning the analysis of the rationale (or explanation) provided by $R3$ when suggesting a move class refactoring ($\mathbf{RQ}_2$), Table IX shows the answers provided by the participants to the related questions. As we can see, the developers considered the rationale provided by $R3$ as meaningful when they accepted a recommended move class

Table IX: Participants' evaluations of explanations provided by $R3$.

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

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 $R3$ (expected result).

Summarizing, we can conclude that when $R3$ 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.

5.1.3. Threats to validity. In this first user study we involved 48 external developers in the evaluation of the move class refactoring operations proposed by $R3$. 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 5.2) 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.

5.2. Evaluation with Original Developers

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

5.2.1. Planning. The four systems involved in the experimentation were eTour, GESA, SESA, and SMOS (see Table I 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 [Bavota et al. 2012] for the material used in our study) where, for each of the suggested opera-

Table X: Participants' evaluations of the refactoring operations proposed by $R3$ 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%

tions, 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 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 $R3$.

Also in this case we evaluated the usefulness of the rationale provided by $R3$ 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.

5.2.2. Analysis of the Results. Table X summarizes the answers provided by the original developers to the question "*Would you apply the proposed refactoring?*" while Table XI 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 X). 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,

Table XI: 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%

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 $R3$ ($\mathbf{RQ}_2$), the original developers generally find meaningful the $R3$'s explanation when they accept a move class operation (74% of cases on average - see Table XI). On the other hand, when developers discard a refactoring operation generally do not find the $R3$'s explanation meaningful (85% of cases). These results are inline with those obtained in the experiment performed with the external developers.

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 XII: 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 (i.e., *Customization*, *ManagerCustomization*, *ServletRefreshCustomizationForm*, and *ServletUpdateCustomization*) composing the package *customization* to the package *timetableManagement*. 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 XII 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 *timetableManagement.customization*. Also the explanations provided by $R3$ were convincing for subjects, like, for example:

MOVE class $Customization$ implementing the topics $[lesson, timetable, hour]$
FROM its package $customization$ grouping the topics
$[customization, parameters, timetable]$
TO the package $timetableManagement$ grouping the topics
$[timetable, lesson, hour, teaching]$

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 $R3$'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 $R3$. 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 $R3$ 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 $R3$'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 $R3$ 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 $R3$ 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 $R3$ identifies strong semantic relationships between classes implementing unrelated responsibilities. However, it is worth noting that $R3$ 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 might be reasonable from a quality metric point of

view[15] (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 $R3$. In fact, in this cases the $R3$'s confidence level is not able to filter out these that seem to be bad refactoring suggestions. Thus, even if the percentage of 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 $R3$ with a high confidence level and negatively evaluated by developers can be found in the SMOS system. In that particular case $R3$ 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 $R3$, 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 explains the rationale behind the $R3$ 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 $R3$'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 highlights that 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 *class-*

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

roomManagement 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.

5.2.3. 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 authors of some bad design choices and consequently might not 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.

6. CONCLUSION AND LESSONS LEARNED

We have presented $R3$, 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, $R3$ 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 $R3$ provides a coupling reduction ranging from 10% to 30% among the software modules. Then, we evaluated the refactoring recommendations by $R3$ 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 $R3$ with high confidence level were considered meaningful by developers.

The evaluation of $R3$ and in particular the deep discussions with the original developers provided us worthwhile information useful to guide future work in software re-modularization (and refactoring) field.

First, we noticed that the explanations provided by $R3$ to software developers when proposing a refactoring, i.e., the confidence level and the textual rationale, seem to be crucial for the suggested refactoring operations. In fact, the confidence level turned out to be a very good indicator of the goodness of the suggested refactoring operations. We now know that developers can mostly ignore the $R3$'s suggestions having a low confidence level (i.e., < 0.58), since the likelihood of having a meaningful suggestion with such a confidence level is quite low. In fact, among a total of 40 refactoring operations having low confidence level analyzed by the original developers, only one (the case of the class *ServletLoadYear* in the system SMOS, discussed in Section 5.2.2) was considered meaningful by them, although we discovered that the application of the move class operation was rather questionable. It seems that $R3$'s suggestions with low confidence level should not be considered as valuable move class refactoring operations, rather as possible opportunities for other types of refactoring opportunities, while in most cases suggestions with high confidence level corresponds to meaningful move class operations.

However, the experiments performed with developers also highlight that the confidence level alone is not enough to state the goodness of a refactoring operation. In fact, while $R3$'s suggestions having high confidence level were generally appreciated by software developers, our studies showed that there are some refactoring operations in this scenario that, even if reasonable, do not justify the need to change the original design from developers' point of view. We also observed that often refactoring operations rejected by them were suggested by $R3$ due to the strong structural relationships (i.e., method calls) existing among the moved class and the classes in the suggested package. Thus, even if these refactoring operations are able to reduce the coupling between the systems packages they were, for different reasons, classified by the developers as "not meaningful" (for example, the move of the class *LoginException* in the SMOS system). These cases highlight the fact that an evaluation of a re-modularization/refactoring technique based only on software quality metrics is not sufficient. This conclusion can also be inferred from a recent work by Anqueti and Laval [Anquetil and Laval 2011] where the authors show that not always cohesion/coupling metrics are reliable when measuring the modularization quality of a software system. This is why we strongly believe that evaluations based on software metrics need to be complemented with experiments performed with software developers in order to get real insights about the actual value of the technique. These observations also pinpoint that, even if better refactoring tools might be developed in the future, the final word about any refactoring operation should be left up to developers, discouraging the implementation of fully automated refactoring tools.

The analysis of the refactoring operations classified by the developers as "not meaningful" also highlighted that some refactoring suggestions generated by $R3$ might be not appreciated by developers due to the presence of crosscutting concerns in the software system (again, see e.g. the movement of the class *LoginException* in the SMOS system). In this case, refactoring approaches trying to group together code components implementing similar responsibilities (like $R3$) might suggest refactoring operations not useful for developers. Even if the $R3$ evaluation performed with developers on five systems highlighted that these cases are rare in real usage scenarios, it can be worthwhile to perform a deeper analysis of crosscutting concerns when designing refactoring/re-modularization recommendation systems.

As for the textual explanations provided by $R3$ via topic analysis, it was generally appreciated by the developers, even though our approach represents the first early

attempt to automatically explain refactoring operations. Clearly, more sophisticated techniques can be exploited to provide more information about suggested refactoring operations, e.g., changes in cohesion/coupling metrics, similar refactoring operations previously applied on the same system, and so on. However, a dedicated study is required to analyze which information sources, among all possible types of information that can be provided to a developer, are indeed useful for explaining the rational behind refactoring operations. Even though this is out of the scope of this paper, we think that our work may contribute to pave the way in this research direction.

Our results also showed that the use of semantic information can be worthwhile in re-modularization tools, confirming findings of our previous work [Bavota et al. 2010]. The semantic information exploited came from terms present in comments, identifiers, and string literals of the analyzed classes. Only in one case we observed side effects of the information extracted from comments on the $R3$'s suggestions (i.e., the move class refactoring operations suggested for the class *ManagerStudent* of the GESA system). In particular, we observed that standard templates used in comments to describe responsibilities of different classes could provide misleading information about the semantic similarity of these classes. A possible solution to this problem has been described by De Lucia *et al.* [De Lucia et al. 2011], 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 an IR method (more than a standard weighting schema like $tf - idf$). The application of these filters could further improve the performances of $R3$ and thus, we plan in future to investigate it. We also plan to evaluate the usefulness of the terms present in different parts of the source code (e.g., considering or not the terms present in comments) when exploiting semantic information for software re-modularization tasks. In the future we also plan to conduct a deeper empirical analysis of the impact of the RTM parameters on the performances of $R3$. In fact, we exploited the RTM parameters defined in previous work [Gethers and Poshyvanyk 2010]. Even if the achieved results are good across a wide range of experimented systems, empirically assessing the RTM parameters would make strengthen the generalization of results. In addition, a more sophisticated approach to tune RTM parameters could also be experimented, like for example the approach defined by Panichella et al. [Panichella et al. 2013] to tune the parameters of LDA.

Finally, the results of the experiment performed with developers also highlighted that approaches like $R3$ could be useful not only during software maintenance to perform software re-modularization tasks, but also, for example, at the end of the development phase in order to "validate" the source code organization defined by the developers. In fact, while the original design of JHotDraw and GESA have been strongly modified during their maintenance, the remaining three systems used in the developers' evaluation still exhibit their original design. Nevertheless, also on these three systems $R3$ was able to find several move class operations well evaluated by the original developers.

Acknowledgments

We would like to thank all the students, researchers, and industry professionals who responded to our survey. We would also like to thank anonymous TOSEM reviewers for their careful reading of our manuscript and high-quality feedback. Their detailed comments have helped us to substantially revise, extend, and improve the original version of this paper.

REFERENCES

2009. Modeling class cohesion as mixtures of latent topics. In *International Conference on Software Maintenance (ICSM)*. Number 25. 233–242.

ABDEEN, H., DUCASSE, S., SAHRAOUI, H. A., AND ALLOUI, I. 2009. Automatic package coupling and cycle minimization. In *Proceedings of the 16th Working Conference on Reverse Engineering*. IEEE CS Press, Lille, France, 103–112.

ANQUETIL, N. AND LAVAL, J. 2011. Legacy software restructuring: Analyzing a concrete case. In *CSMR*. 279–286.

ANQUETIL, N. AND LETHBRIDGE, T. 1999. Experiments with clustering as a software remodularization method. In *Proceedings of 6th Working Conference on Reverse Engineering*. IEEE CS Press, Atlanta, Georgia, USA, 235–255.

ANTONIOL, G., DI PENTA, M., CASAZZA, G., AND MERLO, E. 2001. A method to re-organize legacy systems via concept analysis. In *Proceedings of 9th International Workshop on Program Comprehension*. IEEE CS Press, Toronto, Canada, 281–292.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison-Wesley.

BAJRACHARYA, S. AND LOPES, C. 2009. Mining search topics from a code search engine usage log. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*. MSR '09. IEEE Computer Society, Washington, DC, USA, 111–120.

BALDI, P. F., LOPES, C. V., LINSTEAD, E. J., AND BAJRACHARYA, S. K. 2008. A theory of aspects as latent topics. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, New York, NY, USA, 543–562.

BAVOTA, G., GETHERS, M., OLIVETO, R., AND POSHYVANYK, D. 2012. http://distat.unimol.it/reports/r3/.

BAVOTA, G., LUCIA, A. D., MARCUS, A., AND OLIVETO, R. 2010. Software re-modularization based on structural and semantic metrics. In *Proceedings of the 17th Working Conference on Reverse Engineering*. IEEE CS Press, Beverly, MA, USA, 195–204.

BLEI, D. M., NG, A. Y., AND JORDAN, M. I. 2003. Latent dirichlet allocation. *The Journal of Machine Learning Research 3*, 993–1022.

CANFORA, G., CIMITILE, A., DE LUCIA, A., AND DI LUCCA, G. A. 2001. Decomposing legacy systems into objects: an eclectic approach. *Information and Software Technology 46,* 3, 401–412.

CHANG, J. AND BLEI, D. M. 2010. Hierarchical relational models for document networks. *Annals of Applied Statistics*.

CHEN, T., THOMAS, S. W., NAGAPPAN, M., AND HASSAN, A. E. 2012. Explaining software defects using topic models. In *Proceedings of the 9th Working Conference on Mining Software Repositories*.

CIMITILE, A. AND VISAGGIO, G. 1995. Software salvaging and the call dominance tree. *Journal of Systems and Software 28,* 2, 117–127.

CONOVER, W. J. 1998. *Practical Nonparametric Statistics* 3rd Edition Ed. Wiley.

CORAZZA, A., MARTINO, S. D., MAGGIO, V., AND SCANNIELLO, G. 2011. Investigating the use of lexical information for software system clustering. In *CSMR*. 35–44.

CORAZZA, A., MARTINO, S. D., AND SCANNIELLO, G. 2010. A probabilistic based approach towards software system clustering. In *CSMR*. 88–96.

DE LUCIA, A., DI PENTA, M., OLIVETO, R., PANICHELLA, A., AND PANICHELLA, S. 2011. Improving ir-based traceability recovery using smoothing filters. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*. 21–30.

DEREMER, F. AND KRON, H. H. 1976. Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering 2,* 2, 80–86.

DIT, B., GUERROUJ, L., POSHYVANYK, D., AND ANTONIOL, G. 2011. Can better identifier splitting techniques help feature location? In *Proceedings of 19th IEEE International Conference on Program Comprehension*. IEEE CS Press, Kingston, Canada.

EICK, S., GRAVES, T., KARR, A., MARRON, J., AND MOCKUS, A. 2001. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering 27,* 1, 1–12.

FANTA, R. AND RAJLICH, V. 1999. Restructuring legacy c code into c++. In *Proceedings of the IEEE International Conference on Software Maintenance*. ICSM '99. IEEE Computer Society, Washington, DC, USA, 77–85.

FOWLER, M. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley.

FOWLER, M. 2000. Refactoring catalog. http://refactoring.com/catalog/.

GETHERS, M., OLIVETO, R., POSHYVANYK, D., AND LUCIA, A. D. 2011. On integrating orthogonal information retrieval methods to improve traceability recovery. In *In Proceedings of 27th IEEE International Conference on Software Maintenance*. 133–142.

GETHERS, M. AND POSHYVANYK, D. 2010. Using relational topic models to capture coupling among classes in object-oriented software systems. In *In Proceedings of 26th IEEE International Conference on Software Maintenance*. 1–10.

GETHERS, M., SAVAGE, T., DI PENTA, M., OLIVETO, R., POSHYVANYK, D., AND DE LUCIA, A. 2011. Codetopics: Which topic am i coding now? In *Proceedings of 33rd IEEE/ACM International Conference on Software Engineering*. ACM Press, Honolulu, Hawaii, USA.

GRANT, S., CORDY, J. R., AND SKILLICORN, D. B. 2012. Using topic models to support software maintenance. *Software Maintenance and Reengineering, European Conference on 0*, 403–408.

HARMAN, M., HIERONS, R. M., AND PROCTOR, M. 2002. A new representation and crossover operator for search-based optimization of software modularization. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers Inc., New York, USA.

HINDLE, A., GODFREY, M. W., AND HOLT, R. C. 2009. What's hot and what's not: Windowed developer topic analysis. *Software Maintenance, IEEE International Conference on 0*, 339–348.

KOSCHKE, R., CANFORA, G., AND CZERANSKI, J. 2006. Revisiting the delta ic approach to component recovery. *Science of Computer Programming 60,* 2, 171–188.

KUHN, A., DUCASSE, S., AND GÎRBA, T. 2007. Semantic clustering: Identifying topics in source code. *Information and Software Technology 49,* 3, 230–243.

LANZA, M. AND MARINESCU, R. 2006. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.

LEE, Y., LIANG, B., WU, S., AND WANG, F. 1995. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proceedings of International Conference on Software Quality*. Maribor, Slovenia, 81–90.

LEHMAN, M. M. 1980. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software 1*, 213–221.

LETHBRIDGE, T. C. AND ANQUETIL, N. 2002. *Approaches to clustering for program comprehension and remodularization*. Springer-Verlag New York, Inc., 137–157.

LI, W. AND HENRY, S. 1993. Maintenance metrics for the object oriented paradigm. In *Proc. of METRICS*. 52–60.

MALETIC, J. I. AND MARCUS, A. 2001. Supporting program comprehension using semantic and structural information. In *Proceedings of 23rd International Conference on Software Engineering*. IEEE CS Press, Toronto, Ontario, Canada, 103–112.

MANCORIDIS, S., MITCHELL, B. S., RORRES, C., CHEN, Y.-F., AND GANSNER, E. R. 1998. Using automatic clustering to produce high-level system organizations of source code. In *Proccedings of 6th International Workshop on Program Comprehension*. IEEE CS Press, Ischia, Italy.

MAQBOOL, O. AND BABRI, H. A. 2007. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering 33,* 11, 759–780.

MARCUS, A. AND POSHYVANYK, D. 2005. The conceptual cohesion of classes. In *Proceedings of 21st IEEE International Conference on Software Maintenance*. IEEE CS Press, Budapest, Hungary, 133–142.

MITCHELL, B. S. AND MANCORIDIS, S. 2006. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering 32,* 3, 193–208.

NIERSTRASZ, O., DUCASSE, S., AND DEMEYER, S. 2003. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc.

O'KEEFFE, M. AND O'CINNEIDE, M. 2006. Search-based software maintenance. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering*. IEEE CS Press, Bari, Italy, 249–260.

OLIVETO, R., GETHERS, M., BAVOTA, G., POSHYVANYK, D., AND DE LUCIA, A. 2011. Identifying method friendships to remove the feature envy bad smell. In *Proceedings of the 33rd IEEE/ACM International Conference on Software Engineering*. ACM Press, Hawaii, USA.

OLIVETO, R., GETHERS, M., POSHYVANYK, D., AND DE LUCIA, A. 2010. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*. ICPC '10. IEEE Computer Society, Washington, DC, USA, 68–71.

PANICHELLA, A., DIT, B., OLIVETO, R., DI PENTA, M., POSHYVANYK, D., AND DE LUCIA, A. 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In

Proceedings of the 35th IEEE/ACM International Conference on Software Engineering. ICSE'13. IEEE Computer Society.

PASHOV, I., RIEBISCH, M., AND PHILIPPOW, I. 2004. Supporting architectural restructuring by analyzing feature models. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Tampere, Finland, 25–34.

POSHYVANYK, D. AND MARCUS, A. 2006. The conceptual coupling metrics for object-oriented systems. In *Proceedings of 22nd IEEE International Conference on Software Maintenance*. IEEE CS Press, Philadelphia, Pennsylvania, USA, 469 – 478.

POSHYVANYK, D., MARCUS, A., FERENC, R., AND GYIMÓTHY, T. 2009. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering 14,* 1, 5–32.

PRADITWONG, K., HARMAN, M., AND YAO, X. 2011. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering 37,* 2, 264–282.

PRESSMAN, R. 1992. *Software Engineering: A Practitioner's Approach. 3rd Edition*. McGraw-Hill.

SAVAGE, T., DIT, B., GETHERS, M., AND POSHYVANYK, D. 2010. Topicxp: Exploring topics in source code using latent dirichlet allocation. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. ICSM '10. IEEE Computer Society, Washington, DC, USA, 1–6.

SCANNIELLO, G., D'AMICO, A., D'AMICO, C., AND D'AMICO, T. 2010. Architectural layer recovery for software system understanding and evolution. *Software Practice & Experience 40,* 10, 897–916.

SENG, O., BAUER, M., BIEHL, M., AND PACHE, G. 2005. Search-based improvement of subsystem decompositions. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, Washington, Columbia, USA, 1045–1051.

SENG, O., STAMMEL, J., AND BURKHART, D. 2006. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Seattle, Washington, USA, 1909–1916.

SHAW, S. C., GOLDSTEIN, M., MUNRO, M., AND BURD, E. 2003. Moral dominance relations for program comprehension. *IEEE Transactions on Software Engineering 29,* 9, 851–863.

SHTERN, M. AND TZERPOS, V. 2009. Methods for selecting and improving software clustering algorithms. In *Proceedings of 17th IEEE International Conference on Program Comprehension*. IEEE CS Press, Vancouver, Canada, 248–252.

SOMMERVILLE, I. 2001. *Software Engineering. 6th Edition*. Addison-Wesley.

THOMAS, S. W., ADAMS, B., HASSAN, A. E., AND BLOSTEIN, D. 2010. Validating the use of topic models for software evolution. In *Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation*. 55–64.

THOMAS, S. W., ADAMS, B., HASSAN, A. E., AND BLOSTEIN, D. 2011. Modeling the evolution of topics in source code histories. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 173–182.

TONELLA, P. 2001. Concept analysis for module restructuring. *IEEE Transaction Software Engineering 27,* 4, 351–363.

VAN DEURSEN, A. AND KUIPERS, T. 1999. Identifying objects using cluster and concept analysis. In *Proceedings of 21st International Conference on Software Engineering*. ACM Press, Los Angeles, California, USA, 246 – 255.

WIGGERTS, T. A. 1997. Using clustering algorithms in legacy systems remodularization. In *Proceedings of 4th Working Conference on Reverse Engineering*. IEEE CS Press, Amsterdam, The Netherlands, 33.

WU, J., HASSAN, A. E., AND HOLT, R. C. 2005. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of 21st IEEE International Conference on Software Maintenance*. IEEE CS Press, Budapest, Hungary, 525–535.

YOURDON, E. AND CONSTANTINE, L. 1979. *Structured Design: Foundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall.

A. RELATIONAL TOPIC MODEL

Relational Topic Model [Chang and Blei 2010] 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.* [Chang and Blei 2010]), 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.

There are two steps required 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*, 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 [Blei et al. 2003]:

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

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

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

The link probability function ψ_ϵ is defined as:

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

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

One key distinction between establishing link probabilities in RTM and the canonical LDA is the underlying data used. Here, RTM uses topic assignments to make link predictions whereas to compute document similarities we use topic proportions for each document. This difference is discussed in more detail in the original work of Chang *et al.* [Chang and Blei 2010].

Proposed applications of Relational Topic Models [Chang and Blei 2010] include identifying potential friends within a social network of users, suggesting citations for a given scientific paper, locating web pages relevant to a web page of interest, and analyzing software artifacts to assist with software maintenance tasks and other tasks [Gethers and Poshyvanyk 2010; Gethers et al. 2011; Oliveto et al. 2011; Panichella et al. 2013; Baldi et al. 2008; Bajracharya and Lopes 2009; Liu 2009; Gethers et al. 2011].

A.1. RTM configuration used in $R3$

In $R3$ we configured the RTM parameters as done in [Gethers and Poshyvanyk 2010]. In particular, the following setting was used:

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