RECOMMENDING MOVE METHOD REFACTORINGS USING DEPENDENCY SETS

VITOR MADUREIRA SALES

RECOMMENDING MOVE METHOD REFACTORINGS USING DEPENDENCY SETS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente
Coorientador: Ricardo Terra Nunes Bueno Villela

Belo Horizonte

Março de 2014

VITOR MADUREIRA SALES

RECOMMENDING MOVE METHOD REFACTORINGS USING DEPENDENCY SETS

Dissertation presented to the Graduate Program in Ciência da Computação of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Ciência da Computação.

Advisor: Marco Túlio de Oliveira Valente
Co-Advisor: Ricardo Terra Nunes Bueno Villela

Belo Horizonte

March 2014

Madureira Sales, Vitor

S163r Recommending Move Method Refactorings Using Dependency Sets / Vitor Madureira Sales. — Belo Horizonte, 2014
xxiii, 84 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais — Departamento de Ciência da Computação.
Orientador: Marco Túlio de Oliveira Valente
Coorientador: Ricardo Terra Nunes Bueno Villela

1. Computação - Teses. 2. Refatoração - Teses. 3. Engenharia de software - Teses. 4. Sistemas de recomendação - Teses. I. Orientador. II. Coorientador. III. Título.

519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Recommending move method refactorings using dependency sets

VITOR MADUREIRA SALES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. RICARDO TERRA NUNES BUENO VILLELA - Coorientador
Departamento de Ciência da Computação - UFLA

Fernando J. Castor de Lima Filho

PROF. FERNANDO JOSÉ CASTOR DE LIMA FILHO
Centro de Informática - UFPE

[illegible]

PROF. FERNANDO MAGNO QUINTÃO PEREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 12 de março de 2014.

Agradecimentos

Nenhum trabalho que mereça ser feito é realizado sozinho.

Gostaria de agradecer a toda minha família, especialmente meus pais, José Geraldo e Maria de Fátima, minha irmã Viviane e minha esposa Patrícia que sempre estiveram ao meu lado sendo o meu alicerce em horas adversas.

Gostaria de agradecer ao meu orientador Marco Túlio a oportunidade de trabalhar ao seu lado, fornecendo incontáveis contribuições para o meu crescimento profissional e pessoal.

Gostaria de agradecer ao meu coorientador Ricardo Terra a participação em todo o processo, contribuindo ativamente para a realização e melhoria desse trabalho.

Gostaria de agradecer aos membros dos grupos de pesquisa (ASERG e LabSoft) a cooperação e boa companhia no laboratório e em eventos acadêmicos.

Agradeço aos meus amigos do DCC o prazer da companhia nos "cafezinhos" e convivência.

Gostaria de agradecer aos especialistas que despenderam tempo e conhecimento, contribuindo na realização deste trabalho.

Agradeço à secretaria do DCC todo suporte financeiro e profissional.

Agradeço à CAPES o apoio financeiro.

“Our civilization runs on software.”
(Bjarne Stroustrup)

Resumo

Métodos implementados em classes inapropriadas constituem um *code smell* comum em sistemas orientados a objetos, especialmente quando tais sistemas são mantidos e evoluídos durante anos. A refatoração *Mover Método* é a principal refatoração para resolver tal falha de projeto. Apesar de sua importância, existem poucas ferramentas para auxiliar desenvolvedores na identificação de oportunidades de uso dessa refatoração. Para suprir essa deficiência, apresenta-se uma abordagem que recomenda refatorações *Mover Método* baseada no conjunto de dependências estáticas estabelecidas por um método. Em resumo, a abordagem proposta compara a similaridade entre as dependências estabelecidas por um método com as dependências estabelecidas por métodos de outras classes.

A fim de avaliar a solução, foi projetado e implementado JMove, uma ferramenta que suporta a abordagem proposta nesta dissertação. A abordagem foi avaliada em termos de precisão e *recall* utilizando uma amostra de 14 sistemas de código aberto com 475 instâncias bem definidas do *code smell Feature Envy*, que foram artificialmente sintetizadas. Nesse estudo, alcançou-se uma precisão e *recall* médios de 60,63% e 81,07%, respectivamente. Esses resultados são 129% e 49% melhores do que aqueles obtidos por JDeodorant (um sistema que representa o estado da arte em recomendações de refatoração *Mover Método*) e ainda 556,43% e 378% melhores do que os resultados obtidos pela ferramenta inCode (uma solução comercial para identificação de *code smells*). Finalmente, foi realizada uma segunda avaliação com dois sistemas reais, em que especialistas em tais sistemas avaliaram as recomendações fornecidas por JMove, JDeodorant e inCode.

Palavras-chave : Refatoração *Mover Método*; Sistemas de Recomendação; Conjunto de Dependências.

Abstract

Methods implemented in incorrect classes are common code smells in object-oriented systems, especially in the case of systems maintained and evolved for years. In this scenario, *Move Method* is the key refactoring for tackling this design flaw. Despite its importance and usefulness, there are few tools to assist developers in identifying *Move Method* refactoring opportunities. To address this shortcoming, we propose a novel approach that recommends *Move Method* refactorings based on the set of static dependencies established by a method. In short, our approach compares the similarity of the dependencies established by a particular method with those established by the methods of other classes.

To evaluate our approach, we first designed and implemented JMove, a prototype tool that supports the approach proposed in this dissertation. We evaluate our approach in terms of precision and recall using a sample of 14 open-source systems with well-defined Feature Envy instances, which we have artificially synthesized. More specifically, we evaluated the proposed approach with 475 well-defined Feature Envy instances achieving an average precision and recall of 60.63% and 81.07%. Respectively, these results are 129% and 49% better than those achieved by JDeodorant (a state-of-art *Move Method* recommendation system) and 556.43% and 378% better than those achieved by inCode (a commercial solution for identifying design flaws, such as Feature Envy). Furthermore, we report a second evaluation with two real systems, in which the own system's experts evaluated the recommendations raised by JMove, JDeodorant, and inCode.

Keywords: *Move Method* refactoring; Recommendation systems; Dependency sets.

List of Figures

1.1 Sketch of the proposed approach 3

3.1 Proposed approach 24
3.2 Recommendations in JHotDraw for the best coefficients 33
3.3 Recommendations in JHotDraw for the worst coefficients 34
3.4 Recommendations in JHotDraw using distinct filter's thresholds 34
3.5 JMove's interface 38
3.6 JMove's architecture 38

4.1 SGA's architecture 57
4.2 Intersection between the JMove, JDeodorant, and inCode 61

List of Tables

3.1 Dependency sets for the methods in `CustomerView` and `CustomerDAO` . . . 30
3.2 Similarity values for classes `CustomerView` and `CustomerDAO` 31
3.3 Similarity Coefficients (Extracted from [Terra et al., 2013a]) 32

4.1 Target Systems 43
4.2 Gold Sets size 45
4.3 Precision Results for JHotDraw 47
4.4 Number of Recommendations for JHotDraw 47
4.5 Recall Results 48
4.6 Number of Recommendations 48
4.7 Real-world systems evaluated by experts 56
4.8 Evaluation of the Geplanes' recommendations by an expert 60
4.9 Execution time for JHotDraw (minutes) 63

Contents

Agradecimentos **ix**

Resumo **xiii**

Abstract **xv**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**

1.1 Motivation 1

1.2 An Overview of the Proposed Approach 2

1.3 Outline of the Dissertation 4

1.4 Publications 5

2 Background **7**

2.1 Central Concepts 7

2.1.1 Refactoring 7

2.1.2 Code Smells 8

2.1.3 Software Remodularization 9

2.1.4 Recommendation Systems 10

2.2 Identification of Refactoring Opportunities 11

2.2.1 Design Differencing 12

2.2.2 A search-Based Approaches using Evolutionary Algorithms . . . 12

2.2.3 MethodBook 14

2.2.4 Metrics-based Detection of Code Smells 15

2.2.5 JDeodorant 18

2.3 Final Remarks 20

3 Proposed Approach **23**
3.1 Proposed Algorithm . . . 23
3.1.1 Recommendation Algorithm . . . 24
3.1.2 Similarity Function . . . 25
3.1.3 Target Class Selection . . . 27
3.1.4 The Approach in Action . . . 28
3.2 Exploratory Study: Similarity Coefficients . . . 31
3.2.1 Study Design . . . 32
3.2.2 Similarity Coefficient Results . . . 33
3.2.3 Filter's Threshold Impact . . . 33
3.2.4 Qualitative Analysis . . . 35
3.2.5 Threats to Validity . . . 37
3.3 Tool Support . . . 37
3.4 Final Remarks . . . 39

4 Evaluation **41**
4.1 Evaluation with Open-Source Systems . . . 41
4.1.1 Research Questions . . . 41
4.1.2 Study Design . . . 42
4.1.3 Quantitative Results . . . 46
4.1.4 Qualitative Discussion . . . 49
4.1.5 Threats to Validity . . . 55
4.2 Evaluation with Experts . . . 55
4.2.1 Research Questions . . . 55
4.2.2 Study Design . . . 56
4.2.3 SGA Results . . . 57
4.2.4 Geplanes Results . . . 59
4.2.5 Threats to Validity . . . 62
4.3 Performance Analysis . . . 62
4.4 Final Remarks . . . 64

5 Conclusion **65**
5.1 Summary . . . 65
5.2 Contributions . . . 66
5.3 Limitations . . . 66
5.4 Future Work . . . 67

Bibliography **69**

A Running Example Classes **75**

A.1 Class CustomerView 75

A.2 Class CustomerDAO 77

B SGA running Example Classes **81**

B.1 Class Outsourcing 81

Chapter 1

Introduction

In this chapter, we state the problem and present this dissertation's motivation (Section 1.1). We then provide an overview of our approach (Section 1.2). Finally, we present the outline of the dissertation (Section 1.3) and our publications (Section 1.4).

1.1 Motivation

In object-oriented systems, classes encapsulate an internal state that is manipulated by methods. However, during software evolution, developers inadvertently implement methods in incorrect classes, creating instances of the Feature Envy code smell [Fowler, 1999]. In fact, there are many studies that rank Feature Envy as one of the most recurring code smell [D'Ambros et al., 2010; Sjoberg et al., 2013]. On one hand, the causes of this design flaw are well-known and include deadline pressures, complex requirements, or the partial understanding of the system's design. On the other hand, the consequences can be summarized in the form of a negative impact in the system's maintainability [Lanza et al., 2005; Yamashita and Moonen, 2012].

Besides the causes and consequences, the refactoring actions to remove a Feature Envy are also well-documented. Basically, a *Move Method* refactoring must be applied to move the method from its current class to the class that it envies [Fowler, 1999; Tsantalis and Chatzigeorgiou, 2009]. In practice, such refactoring is usually supported by the automatic refactoring engines that are part of most modern IDEs. Therefore, the task of applying a fixing action is not challenging in case of Feature Envy design flaws. On the other hand, before applying this refactoring, maintainers must perform two program comprehension tasks: (a) detect the Feature Envy instances in the source code, and (b) determine the correct classes to receive the methods detected by the first task. Typically, such tasks are more complex because they require a global understanding of the system's design and implementation, which is a skill that only experienced developers have.

Moreover, *Move Method* is also a key refactoring for improving software architectures. For example, Terra et al. [2013c] proposed an architectural repair recommendation system to fix violations raised by architecture conformance checking approaches. Among the repair actions suggested by this system, several include a recommendation to move a method to another class.

Despite the importance and usefulness of the *Move Method* refactoring, currently there are few tools to assist developers in identifying opportunities to take advantage of this refactoring. As one example, JDeodorant is a *Move Method* recommendation system that follows a classical Feature Envy heuristic with some improvements to detect opportunities of *Move Method* refactorings [Tsantalis and Chatzigeorgiou, 2009]. Basically, for JDeodorant a method m envies a class C when m accesses more services from C than from its own class. However, this heuristic generally produces too many recommendations and a higher number of false positives. For example, in a preliminary experiment with JHotDraw, a system that is well-known for its internal quality, JDeodorant generates 27 recommendations. Therefore, the challenge when designing such recommendation systems is not only to generate correct refactoring recommendations, but also to avoid incorrect recommendations that an experienced developer would easily discard.

1.2 An Overview of the Proposed Approach

As stated in the previous section, the task of detecting Feature Envy instances and determining the correct classes to receive the detected methods is non-trivial, time-consuming, and usually performed in an ad hoc way without adequate tool support. To tackle this problem, we describe in this master dissertation a solution based on recommendation system principles that provides *Move Method* refactoring recommendations.

As illustrated in Figure 1.1, our approach detects methods displaying a Feature Envy behavior. In such cases, we also suggest a target class where the methods should be moved to. More specifically, our approach is centered on the following assumption: *methods in well-designed classes usually establish dependencies to similar types*. For example, suppose that class `CustomerDAO` is used to persist customers in a database. Typically, the dependency sets of the methods in this class include common domain types (such as `Customer`) and also common persistence related types (such as

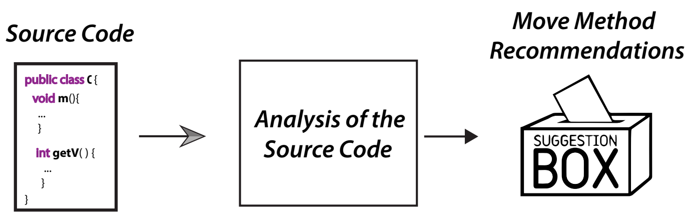


Figure 1.1: Sketch of the proposed approach

`SQLException`). Suppose also that one of such methods, named *getAllCustomers* (as presented in Code 1.1), is inadvertently implemented in class `CustomerView`, responsible for user interface concerns. In this case, the dependency set of *getAllCustomers* will not contain dependencies to types such as `Button`, `Label`, etc., which are common in methods from `CustomerView`. Therefore, since the dependency set of *getAllCustomers* is more similar to the dependency sets in `CustomerDAO` than to the dependency sets in `CustomerView`, our approach may trigger a *Move Method* recommendation, suggesting to move *getAllCustomers* from the former to the latter.

```
public class CustomerView extends JFrame {
  ... declaration of variables
  public List<Customer> getAllCustomers(DB db) throws SQLException {
      List<Customer> result = new ArrayList<Customer>();
      Connection con = db.getConnection();
      PreparedStatement ps =
                        con.prepareStatement("select * from CUSTOMER");
      ResultSet rs = ps.executeQuery();

      while (rs.next()){
          result.add(new Customer(rs.getInt("ID"),rs.getString("NAME")));
      }
      rs.close();
      ps.close();
      con.close();
      return result;
  }
}
```

Code 1.1: `getAllCustomers` method

In particular, to measure the similarity between a given method m with its own class C, we compute the average similarity between the set of dependencies established by m and by the remaining methods in C. Similarly, to measure the similarity between the method m and a class C_i, we compute the average similarity between the dependencies established by m and by the methods in class C_i. If the similarity between a given method m and a class C_i is greater than the similarity between m and its own class C, we infer that m is more similar to C_i than to its current class C. Therefore, C_i is a potential candidate class to receive m.

To measure the similarity between the sets of dependencies established by two methods we rely on similarity coefficients, which are usually employed to measure the similarity between two generic sets. However, to choose the most suitable coefficients, we conducted an exploratory study where we evaluated 18 similarity coefficients using JHotDraw, which is a system commonly used to illustrate object-oriented programming practices and patterns. As the result, we decided to rely on the *Sokal and Sneath 2* coefficient [Sokal and Sneath, 1963, 1973].

In order to evaluate our approach, we implemented a prototype tool, called JMove. Basically, JMove is an Eclipse plug-in that supports the approach proposed in this dissertation. We evaluate our approach in terms of precision and recall, using a sample of 14 open-source systems with well-defined Feature Envy instances, which we have synthesized manually. We report that our solution provides an average precision of 60.63% and an average recall of 81.07%. Respectively, these results are 129% and 49% better than those achieved by JDeodorant (a state-of-art *Move Method* recommendation system) and 556.43% and 378% better than those achieved by inCode (a commercial solution for identifying design flaws, such as Feature Envy). Furthermore, we conduct a second evaluation with two real systems. Experts on these two systems evaluated the recommendations raised by JMove, JDeodorant, and inCode. In this case we concluded that JMove was able to identify real Move Method opportunities.

1.3 Outline of the Dissertation

We organized the remainder of this work as follows:

- **Chapter 2** covers central concepts related to this dissertation, including a discussion on refactoring and code smells. We also present some remodularization approaches based on recommendation system principles. Finally, we provide an overview on tools for detecting refactoring opportunities.

- **Chapter 3** presents the proposed recommendation system, including the description of its underlying algorithms and an example that illustrates its operation. Furthermore, this chapter presents an exploratory study to select the most appropriated similarity coefficient used by our approach. Finally, we present a tool, called JMove, that implements our approach.

- **Chapter 4** compares our approach with two other recommendations systems using 14 open-source systems where we artificially created the methods to be moved. We detail the dataset used in this evaluation and the methodology we followed to generate the gold sets used to calculate precision and recall. Furthermore, we evaluate our approach with two real-world systems, when an expert in each system evaluated the triggered recommendations.

- **Chapter 5** presents the final considerations of this dissertation, including the contributions, limitations, and future work.

1.4 Publications

This dissertation generated the following publications and therefore contains material from them:

- Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. Recommending Move Method Refactorings Using Dependency Sets. *In 20th Working Conference on Reverse Engineering (WCRE)*, pages 232-241, 2013.
- Vitor Sales, Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente. JMove: Seus Métodos em Classes Apropriadas. *IV Congresso Brasileiro de Software: Teoria e Prática (Sessão de Ferramentas)*, p. 1-6, 2013.
- Vitor Sales. Revealing Move Method Opportunities. Poster, presented in *1st Latin-American School on Software Engineering*, 2013. (Prize of *third best* poster in Master's student category).

Chapter 2

Background

In this chapter, we discuss background work related to this dissertation. First, Section 2.1 presents central concepts of our study: a description of refactoring, since our approach relies on a well-known refactoring to improve the system design (Section 2.1.1); a solid definition of code smells, which are related to our work in terms of being a design flaw we aim to solve (Section 2.1.2); a discussion on remodularization approaches because they represent potential alternatives to architecture degradation (Section 2.1.3); and an overview on recommendation systems because our approach is based on recommendation system principles (Section 2.1.4). Second, Section 2.2 provides an overview on the identification of refactoring opportunities since our approach detects *Move Method* refactoring opportunities using dependency sets. Last, Section 2.3 concludes this chapter with a general discussion.

2.1 Central Concepts

2.1.1 Refactoring

An intrinsic property of real-world systems is their need to evolve. As software evolves it is modified and adapted to new requirements. During the evolution process, the software source code usually becomes more complex and deviates from its original design, which leads to a decrease in quality [Lehman, 1980].

Refactoring, as defined by Fowler [1999], is a change made to the internal structure of a system to make it easier to understand and cheaper to modify, while preserving its observable behavior. Thus, refactoring can be seen as a technique for reorganizing code. Despite the benefits that continuous refactoring can provide, the refactoring process is commonly overlooked due to budget and time constraints. On the other hand, architectural deviations—i.e., violations from the intended architecture—have a cumulative effect, which makes the maintenance and system evolution increasingly costly [Terra and Valente, 2009].

Although the understanding of refactorings is straightforward, the task of formalizing and automating them is not trivial [Schäfer et al., 2009; Borba et al., 2004; Steimann and Thies, 2009; Tsantalis and Chatzigeorgiou, 2009; Verbaere et al., 2006; Opdyke, 1992; Soares et al., 2010, 2013]. The mechanics behind refactorings is usually specified in natural language and does not cover all possible scenarios and preconditions. In fact, even a bug-free implementation for typical refactorings—i.e., refactorings whose scopes are limited to few classes—has proved to be a complex task [Steimann and Thies, 2009].

2.1.2 Code Smells

Knowing how refactoring works and its benefits is not sufficient to determine when a refactoring should be applied. The decision on when to start and to stop the refactoring process is as important as knowing the mechanics behind refactorings. Although no metrics or heuristics can compete with human intuition, code smells are usually considered good indicators of design problems that can be solved through refactorings.

According to Fowler [1999], a code smell is a symptom in the source code that typically corresponds to a deeper problem in the system. Although code smells do not prevent the operation of the system, they may indicate deficiencies in the design that may hamper maintenance and evolution. Although Fowler [1999] defined 23 code smells, we present here only the Feature Envy one because it is directly related to the approach proposed in this dissertation.

Feature Envy:

> "A classic smell is a method that seems more interested in a class other than the one it actually is in. The most common focus of the envy is the data. We've lost count of the times we've seen a method that invokes half-a-dozen getting methods on another object to calculate some value. Fortunately the cure is obvious, the method clearly wants to be elsewhere, so you use *Move Method* to get it there" [Fowler, 1999].

More specifically, we can solve the feature envy code smell by three distinct ways: (i) we can use the *Move Method* refactoring to move a method from its current class to the class it envies; (ii) when only a code snippet of the method is responsible for the envy, we can first apply the *Extract Method* refactoring and then apply the *Move Method* refactoring to move the extracted method to the class it envies; and (iii) we can move the envied attributes to the class that envies them.

2.1.3 Software Remodularization

Remodularization is a process that modifies the modular design of a system for the purposes of adaptation or evolution. Remodularization does not imply changes in the system behavior. In fact, it can be very convenient to change the modular design of a system without changing its external behavior. In this manner, remodularization is quite similar to the definition of refactoring. However, remodularization usually denotes a large number of refactorings sequentially applied to improve the system architecture.

Basically, such approaches are proposed to support the cases where a renovation in the underlying architecture or even a complete rearchitecting effort are required. In such instances, the transformations may include not only moving methods, but also extracting classes, splitting packages, etc. For example, Moghadam and Ó Cinnéide [2012] proposed an approach to automatically refactor the source code towards a desired design, provided in the format of a UML class diagram. The proposed approach compares the current and desired models and expresses the design differences as a set of refactorings. In contrast, instead of a desired UML model, our approach suggests *Move Method* refactorings to enhance a preexisting design.

Hierarchical clustering is another technique commonly proposed to evaluate alternative software decompositions [Anquetil and Lethbridge, 1999; Santos et al., 2014]. As an example, Bunch is a tool that recommends system decompositions by partitioning graphs of the entities (e.g., classes) and relations (e.g., dependencies) in the source code [Mitchell and Mancoridis, 2006]. However, the effectiveness of clustering in reengineering tasks is often challenged. For example, Glorie et al. [2009] reported an experiment in which clustering and formal concept analyses have failed to produce an acceptable partitioning of a monolithic medical imaging application.

To reveal the differences observed between current and planned software architectures, Maffort et al. [2013] proposed a conformance tool called ArchLint. The authors combine static and historical source code analysis techniques in order to provide a lightweight alternative for architecture conformance. They consider ArchLint a lightweight approach because it requires only two inputs on the system under analysis: (i) a high-level component specification and (ii) the history of revisions, without requiring further refinements on component specification.

The recommendation approach described in this dissertation was inspired by a previous work on architecture conformance and repair [Terra and Valente, 2009; Terra et al., 2013c]. More specifically, Terra et al. [2013c] proposed a recommendation system, called ArchFix, that supports a catalog of architectural repair recommendations to fix violations raised by architecture conformance checking approaches (e.g., ArchLint). Some repair actions in ArchFix include a recommendation to move a method to another class, which is inferred using the set of dependencies established by the source method and the target class. However, methods located in the wrong class are provided as an input to the recommendation algorithm (and represent an architectural violation). On the other hand, the goal of the approach proposed in this dissertation is to automatically discover such methods, i.e., we investigate methods that do not necessarily denote architectural violations.

2.1.4 Recommendation Systems

Recommendation Systems are software applications that aim to support users in their decision-making while interacting with large amounts of information. Recommendation Systems help to overcome information overload by showing users only the most relevant items [Robillard et al., 2010]. More specifically, Recommendation Systems for Software Engineering (RSSEs) provide potentially valuable information for software engineering tasks in a given context. In this section, we discuss some RSSEs with focus on demonstrating the feasibility of a solution based on recommendation system principles for enhancing the system design.

CodeBroker [Ye and Fischer, 2005] retrieves relevant software components based on source code. It analyzes developer comments in the code and the method's signature to extract queries. Thereafter, it detects context-relevant code components that may help users to implement the functionality of interest. CodeBroker uses a combination of textual-similarity analysis and type-signature matching to identify relevant elements. Moreover, the tool works in push mode, i.e., it produces recommendations each time developers write comments.

Similarly, Holmes et al. [2006] propose Strathcona, an approach that relies on the structure of the source code under development to retrieve relevant examples. Strathcona automatically extracts a set of structural facts (e.g., method calls) about the context of an indicated source code fragment and searches for examples that contain similar structural context in the repository. As a distinguishing feature regarding other approaches, relevant examples are automatically extracted from a source code fragment indicated by the developer.

In a similar research line, to help developers in using an API (Application Programming Interface), Montandon et al. [2013] proposed APIMiner, a platform that instruments the standard Java-based API documentation format with concrete source code examples of usage, which are extracted from a private repository. APIMiner extracts examples from an internal repository of source code projects, which should be populated before starting to use the platform. After that, APIMiner summarizes and ranks the examples off-line, i.e., during a pre-processing phase. Finally, APIMiner automatically generates a new Javadoc documentation, with buttons that provide access to the extracted source code examples.

2.2 Identification of Refactoring Opportunities

Although it is possible to manually refactor a system, tool support is considered crucial [Mens and Tourwé, 2004]. Basically, there are two types of approaches related to the degree of automation provided by refactoring tools: semi-automatic and automatic approaches.

In a semi-automatic approach, the intervention of the developer is required to identify which parts of the software need to be refactored or to select which refactorings should be applied. After that, the application of the refactoring is automated. Based on two studies of non-trivial cases, Tokuda and Batory [2001] estimated that semi-automatic tools can increase the productivity factor in ten times or more. Although semi-automatic refactoring tools that involve too much human interaction might make refactoring a consuming activity, they represent the most useful approach in practice because a significant part of the knowledge required to perform the refactoring cannot be extracted from the software [Mens and Tourwé, 2004].

As an alternative approach to semi-automatic approaches, some researchers investigated solutions that offer a fully automated refactoring process, i.e., the developer intervention is not necessary [Moghadam and Ó Cinnéide, 2012, 2011; Seng et al., 2006; Kataoka et al., 2001; Moore, 1996; Casais, 1994]. On the other hand, automatic approaches usually present to developers the refactored code after the whole process, which may cause misunderstanding, in the sense that certain parts of the refactored software become even more difficult to understand.

In recent years, many techniques have been proposed to deal with the identification of the refactorings. Since our proposed approach works on recommending *Move Method* refactorings, an overview the state-of-the-art on identification techniques of refactorings is presented in following sections.

2.2.1 Design Differencing

Moghadam and Ó Cinnéide [2011] proposed a platform named Code-Imp (Combinatorial Optimisation for Design Improvement), which performs automated refactorings for Java. Code-Imp uses an automatic refactoring-based search, i.e., refactorings are mapped as a search problem and the goal is to maximize the value of an objective function. Thus, given an objective function, random refactorings are applied in order to move around the solution space seeking a configuration that maximizes this function. The search process is guided by a set of 27 metrics and the fitness function can be defined based on any combination of these metrics. Refactorings are accepted when, in addition to preserving the behavior of the system, they improve the values of the chosen set of metrics.

Using the Code-Imp tool, Moghadam and Ó Cinnéide [2012] proposed an approach based on differences between UML (Unified Modeling Language) class models. In this approach, developers first extract the UML class model of the system to refactor using a tool named JDEvAn [Xing and Stroulia, 2008]. In practice, the extracted class model represents the original architecture of the system. Second, developers modify such extracted class model towards a desired architecture. Third, an algorithm called UMLDiff extracts the differences between the original extracted class model and the modified one. The differences are mapped to one or more refactorings supported by Code-Imp. Finally, the results from this process are the refactorings needed to transform the current architecture into the desired one.

To validate their approach, the authors conducted experiments using six open-source Java applications. In each experiment the application under investigation was randomly, massively, and automatically refactored to change its design structure. As expected, this process led to a decrease in the quality of the design of the software creating a degraded system architecture. After that, the authors rebuilt the original source code from its low-quality refactored version based on the detected design differences. The authors conclude that the original program could be refactored to the desired architecture with an accuracy of over 90%. Although their evaluation indicated a high degree of accuracy, it was conducted in a very restricted context and hence the results may not be as good in real scenarios.

2.2.2 A search-Based Approaches using Evolutionary Algorithms

Seng et al. [2006] investigated how search-based techniques can be applied to reconditioning the class structure. They described a novel search-based approach that assists software engineers to improve the system quality, using evolutionary algorithms and simulated refactorings.

Specifically, the approach relies on a standard fact extraction technique to transform the source code into a figurative model called phenotype. Phenotype consists of the abstract source code model and several model refactorings, which simulate the actual source code refactorings. The intermediate model is created to allow the simulation of refactorings in the source code and to evaluate their impact on a fitness function. The goal is to find a set of refactorings that—when applied to the source code model (phenotype)—maximizes the value of the fitness function and hence improves the system quality. In addition, all refactorings performed on this model are stored in a structure named genotype, which consists of an ordered list of refactorings to transform the initial model `A` into an improved model `B`.

The fitness function measures the improvement in the quality of the source code. This function is a weighted sum of several metric values that should be maximized. It captures several properties of interest, such as coupling, cohesion, complexity, and stability. The authors rely on several metrics from Briand's catalogue to assess these properties [Briand et al., 1998]. For example, they use *Response for class* (RFC) and *Information-flow-based-coupling* (ICP) to measure coupling and metrics *Tight class cohesion* (TCC), *Information-flow-based-cohesion* (ICH), and *Lack of cohesion* (LCOM5) to measure cohesion.

To conduct the evolutionary process, they consider multiple models that form their population at a time. The initial population of models is created by copying the initial model, extracted from the original source code, x times. During the first evolutionary step, elements of the current population are modified using a random model refactoring or a new model is created by combining genomes of two models using a crossover operator. After this step, there are $x + y$ models, i.e., the x preexisting models and the y recent created models. In order to reduce the population to its initial size of x elements, the most promising elements are selected using a tournament selection strategy.

The optimization stops after a predefined number of evolutionary steps, which is chosen based on the number of model elements and on the number of refactorings. The model with the best rating according to the fitness function is chosen as the best solution to improve the system quality. Since refactorings responsible for this model generation are stored as genotype, users can identify which refactorings they need to carry out on the original source code.

The authors evaluated their approach using JHotDraw, a well-known open-source system. They modified JHotDraw by misplacing ten randomly selected methods. Given the non-deterministic nature of the approach, they perform series of ten runs using the modified system. As a result, nine out of ten methods were moved back during each run, which indicates that the approach was able to restore the original design.

2.2.3 MethodBook

Oliveto et al. [2011] proposed MethodBook, an approach to identify *Move Method* refactoring opportunities and to remove the Feature Envy bad smell from source code. The approach considers both structural and conceptual relationships between methods to identify sets of methods that share the same responsibilities. They investigate friendships between methods, using Relational Topic Model [Chang and M. Blei, 2010], to infer the target class where the method should be moved to.

The concept of friendships between methods is used as a metaphor to identify envied classes. Basically, two methods are considered friends when they share responsibilities, i.e., both methods operate on the same data structures or rely on the same features or concepts. Metaphorically, the authors claim that methods that are "good friends" should belong to the same class.

In practice, this approach can be explained with an analogy to Facebook. In Facebook, users can add friends, send them messages, and update their personal profiles to notify friends about themselves. In particular, Facebook also relies on Relational Topic Model (RTM) to analyze users profiles and to suggest new friends or groups of people sharing similar interests. Analogously, in MethodBook, methods and classes play the same role as people and groups in Facebook, respectively. Moreover, methods' bodies are analogous to profiles in Facebook and contain information about structural (e.g., method calls) and conceptual relationships (e.g., similar comments) with other methods.

Their approach relies on RTM to identify "friends" of a method and therefore to suggest *Move Method* refactoring opportunities. In particular, MethodBook suggests as a target class the one that contains the higher number of "friends" with the method under analysis. The model generated by RTM is used to determine the degree of similarity among methods in the system and to rank friendships among these methods. In this phase, a cut point is used to identify the n best friends, which represent the methods having the highest similarity with the method under analysis. Once the "best" friends are identified, MethodBook analyzes their classes to determine the envied class, which contains the highest number of friend methods with the examined method. This choice is justified by the conjecture that the higher the number of friends placed in the same class, the higher its quality in terms of cohesion and coupling.

The authors performed a preliminary evaluation of their approach on ArgoUML, an open-source system. The goal was to investigate whether MethodBook is able to identify meaningful *Move Method* refactoring operations for a given method. First, they randomly extracted 1,000 methods from classes of ArgoUML. Second, the authors applied MethodBook to identify the envied class for each extracted method, which is likely to be the class the method was extracted from. As the result, their approach achieved a recall of 75% and a precision of 70%. However, it is worth noting that the method has to be given as input to the approach.

2.2.4 Metrics-based Detection of Code Smells

Marinescu [2004] proposed an approach to help developers and maintainers to detect code smells. The proposed detection strategies consist of metrics-based rules to capture deviations from good design principles. Using these strategies, engineers can directly localize classes or methods affected by a particular code smell (e.g., Feature Envy), rather than having to infer the real design problem from a large set of abnormal metric values.

According to the author, a detection strategy is the quantifiable expression of a rule by which design fragments that are conforming to that rule can be detected in the source code. Therefore, a detection strategy is a generic mechanism for analyzing a source code model using metrics. The use of metrics in detection strategies is based on filtering and composition mechanisms. While the objective of filtering is to reduce the initial data set, compositions are used to correlate the interpretation of multiple metrics.

The author classifies the filters in various categories. Filters such as *HigherThan* and *LowerThan* are parameterized with an absolute threshold, e.g., "a class should not be coupled with more than six other classes". Another category of filters have their value relative to the original data. Filters such as *TopValues* and *BottomValues* require a parameter that specifies the number of entities to be retrieved, rather than specifying the maximum or minimum value allowed in the result set. The parameters may be absolute—e.g., "keep 20 entities with the highest values"—or relative—e.g., "retrieve 10% of all measured entities having the lowest values".

In addition to filtering mechanisms, detection strategies may be defined by composition mechanisms using the operators *and*, *or*, and *butnot*. In terms of set operators, the *and* operator is mapped to intersection, the *or* to union, and the *butnot* to minus.

As a clarification example, the author described a strategy for detecting God Classes, which is defined as:

"An object that controls way too many other objects in the system and has grown beyond all logic to become the class that does everything" [Riel, 1996].

The starting point is given by one or more informal rules that define comprehensively the design problem. Particularly for God Classes, the authors use the following set of three heuristics [Riel, 1996]:

Rule #1: *Top-level classes in a design should share work uniformly;*
Rule #2: *Beware of classes with much non-communicative behavior; and*
Rule #3: *Beware of classes that access directly data from other classes.*

The first step for constructing a detection strategy is to translate informal rules in a set of correlated symptoms. In the God Class case, the first rule refers to the lack of a uniform distribution of intelligence among classes. Since it refers to the high complexity of a class, the author selected the *Weighted Method Count (WMC)* metric to quantify complexity [Chidamber and Kemerer, 1994]. The second rule is related to the level of intraclass communication. Since it refers to low cohesion, the author selected the *Tight Class Cohesion (TCC)* metric [Bieman and Kang, 1995]. The third rule addresses a special type of coupling, that is, direct accesses to attributes defined in other classes. Since it refers to the access of foreign data, the author selected *Access to Foreign Data (ATFD)* metric to represent this property [Lanza et al., 2005].

The second step is to determine the proper filtering mechanism that should be used with each metric. Because the first symptom is a "high class complexity", the author chose the *TopValues* filter for the WMC metric. For the "low cohesion" symptom, the author chose the *BottomValues* filter. Finally, for the third symptom (to capture access to foreign data), the author chose the *HigherThan* filter. Afterwards, it is necessary to correlate these symptoms using the composition operators. For the mentioned symptoms, the author infers that all these three symptoms should coexist in God Classes, and therefore, the author used the *and* ($\wedge$) operator to connect all symptoms. In summary, Equation 2.1 illustrates the detection strategy adopted for God Classes.

$$\begin{gathered} GodClass(C) = \\ (\mathbf{WMC}(C), TopValues(25\%)) \wedge (\mathbf{TCC}(C), BottomValues(25\%)) \wedge (\mathbf{ATFD}(C), HigherThan(1)) \end{gathered} \tag{2.1}$$

One of the most critical tasks in defining a detection strategy is to set the thresholds for each data filter. For Example, in Equation 2.1, the author uses a simplistic approach and consider a 25% value for both the *TopValues* and the *BottomValues* filters. The selection of a threshold for the ATFD metric is simpler because no direct access to data of other classes is permitted and, hence, the threshold value is one.

The threshold values have direct influence on the accuracy of the detection strategies. In fact, this problem intrinsically characterizes metrics-based approaches. More specifically, to determine "correct" threshold values is one of the major limitations of such approaches. In practice, threshold values are obtained by empirical processes, guided by past experiences and based on hints from metrics' authors. There are approaches working on this problem, e.g., an approach based on a "tuning machine" that tries to automatically find proper threshold values [Mihancea and Marinescu, 2005], approaches that extract thresholds automatically from a software repository [Oliveira et al., 2014]. Despite increasing efforts in this direction, setting the best value up for thresholds remains a central challenge in metric-based approaches.

The authors claim to have also defined detections strategies for more than ten design problems, such as *Shotgun Surgery*, *Misplaced Class*, *God Method*, and *Feature Envy*. Because it is directly related to the approach proposed in this dissertation, we present the Feature Envy detection strategy, which is defined as follows:

$$\begin{aligned} &FeatureEnvy(m) = \\ &\quad \mathbf{ATFD}(m), HigherThan(FEW)) \wedge \\ &\qquad\qquad (\mathbf{LAA}(m), LessThan(ONE\ THIRD)) \wedge (\mathbf{FDP}(m), LessOrEqualThan(FEW)) \end{aligned} \tag{2.2}$$

where metrics ATFD (Access to Foreign Data) measures the number of distinct attributes the measured element accesses, LAA (Locality of Attribute Accesses) measures the relative number of attributes that a method accesses on its class, and FDP (Foreign Data Providers) measures the number of classes where the accessed attributes belong to.

Marinescu et al. [2005] designed IPlasma, a tool that implements the aforementioned approach. Such tool is an integrated environment for quality analysis of object-oriented software systems that supports more than 80 design metrics. It can also detect design flaws—such as Feature Envy, Data Class, and God Class. Recently, the tool has evolved to a commercial version, named inFusion.[1]

[1]www.intooitus.com/products/infusion

2.2.5 JDeodorant

Proposed by Tsantalis and Chatzigeorgiou [2009], JDeodorant follows a classical heuristic to detect Feature Envy bad smells: a method m envies a class C' when m accesses more services from C' than from its own class. However, JDeodorant includes two major improvements to this heuristic: (i) an Entity Placement metric and (ii) a set of *Move Method* refactoring preconditions.

In order to avoid an explosion in the number of false positives, JDeodorant defines a metric, called Entity Placement, to evaluate the quality of possible *Move Method* recommendations. This metric is used to evaluate whether a potential recommendation reduces a system-wide measurement of coupling and also improves a system-wide measurement of cohesion.

To calculate the Entity Placement metric, JDeodorant creates an *entity set* for each attribute and method. The *entity set* for a method m is formed by the attributes that m accesses directly or through accessor methods and by the methods that are called by m. On the other hand, the *entity set* for an attribute is formed by the methods that directly or through accessors methods access it. Apart from the *entity sets* of methods and attributes, the *entity set* of a class C contains only its attributes and methods.

In the Entity Placement metric, a high similarity value between a method and a class denotes a large number of common entities in their *entity sets*. JDeodorant relies on the Jaccard similarity coefficient to calculate the similarity between *entity sets*, as defined in Equation 2.3.

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.3}$$

On the other hand, the Jaccard distance—which measures the dissimilarity between sets—is obtained by subtracting Jaccard similarity value from one, as defined in Equation 2.4.

$$d_J(A, B) = 1 - Jaccard(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{2.4}$$

JDeodorant assumes that the Jaccard distance from a class to the inner entities should be as small as possible to achieve high cohesion. Moreover, the Jaccard distance from a class to outer entities (i.e., entities that do not belong to the class) should be as large as possible to achieve low coupling. Therefore, the Entity Placement metric is defined considering for each class the ratio between the average inner and outer entity distances. For a given class, the closer to zero such ratio is, the safer to conclude that entities are correctly placed. Assume that e_i refers to inner entities and e_j to outer entities, the Entity Placement value for a class C is defined by Equation 2.5.

$$EntityPlacement(C) = \frac{\dfrac{\sum\limits_{e_i \in C} d_J(e_i, C)}{|entities \ \in \ C|}}{\dfrac{\sum\limits_{e_j \notin C} d_J(e_j, C)}{|entities \ \notin \ C|}} \tag{2.5}$$

Finally, JDeodorant defines a rich set of *Move Method* refactoring preconditions to check whether the refactoring recommendations preserve the system's behavior and design quality. These preconditions are divided into three groups: Compilation, Behavior-Preservation, and Quality Preconditions.

Compilation Preconditions:

1. The target class should not contain a method having the same signature as the method to be moved.
2. The method to be moved should not override an abstract method. Moving a method that overrides an abstract method can generate compilation problems because the overriding of abstract method is mandatory for concrete classes.
3. The method to be moved cannot contain `super` method invocations.
4. The target class cannot be an interface because interfaces contain only abstract methods.

Behavior-Preservation Preconditions:

1. The target class should not inherit a method having the same signature as the method to be moved. Moving such method would lead to an overriding of the inherited method, affecting the behavior of the target class.
2. The method to be moved should not override an inherited method in its original class. Moving such method would affect the behavior of the source class because it would enable the inheritance of the method defined in its superclass.

3. The methods that originally invoke the method to be moved should be modified to invoke it through the new reference. Therefore, the method should have a reference to the target class through its formal parameters or source class fields.

4. The method to be moved should not be synchronized due to concurrency issues.

Quality Preconditions:

1. The method to be moved should not contain assignments to a class field, including inherited fields. In fact, moving such method would increase the coupling between the source and target classes because the method will remain coupled to the source class. Furthermore, a method that changes the value of a field has stronger conceptual binding with the class it belongs to, when compared to a method that simply accesses the value of the field.

2. The method to be moved should have a one-to-one relationship with the target class. This precondition does not allow the method to be moved to arrays or Java Collection types.

JDeodorant considers only *Move Method* refactorings, and therefore it targets only the Feature Envy design problem. Nevertheless, the tool has been recently extended to also identify *Extract Method* [Tsantalis and Chatzigeorgiou, 2011] and Extract Class refactorings [Fokaefs et al., 2012].

2.3 Final Remarks

This chapter provided the background necessary to fully understand the approach proposed in this dissertation. In Section 2.1, we presented central concepts of our study: a description of refactoring since our approach relies on a primitive refactoring (*Move Method*) to improve system architectures (Section 2.1.1); the relevance of bad smells to spot opportunities of refactorings. We specified and then described the Feature Envy code smell, which is directly related to the recommendation system proposed in this dissertation (Section 2.1.2); a discussion on several remodularization tools and techniques (Section 2.1.3); and an overview on RSSEs. These concepts are indeed essential since our approach is a recommendation system that aims to achieve a better software design by recommending *Move Method* refactorings Section 2.1.4.

In Section 2.2, we provide an overview of the state-of-the-art techniques on identification of refactoring opportunities, since our approach aims to recommend *Move*

Method refactorings to improve the system architecture. In particular, we describe two main approaches: (i) *Detection Strategy*, which consists of metrics-based rules that localize classes or methods affected by a particular code smell, and (ii) *JDeodorant*, which follows a classical heuristic to detect Feature Envy code smells. These approaches are especially important because we compare our approach with both of them.

In the next chapter, we present the *Move Method* refactoring recommendation system proposed in this dissertation, including underlying algorithms, an exploratory study for choosing the most appropriate similarity coefficient, and the design of the *JMove* tool, an Eclipse plug-in that supports our approach.

Chapter 3

Proposed Approach

Chapter 2 provided the background for understanding this dissertation. Thereupon, this chapter then describes our approach: a system that recommends *Move Method* refactorings based on the set of static dependencies established by a method.

We organized this chapter as follows. Section 3.1 presents a specification of the proposed recommendation system, including the process of determining the most appropriate class for a method, the description of underlying algorithms and similarity functions, and an example that illustrates the operation of our system. Section 3.2 describes an exploratory study to determine the most suitable coefficient to our approach. Section 3.3 presents JMove, a tool that implements the proposed approach. Last, Section 3.4 concludes with a general discussion.

3.1 Proposed Algorithm

Our recommendation approach detects methods located in incorrect classes and then suggests moving such methods to more suitable ones. More specifically, we first evaluate the set of static dependencies established by a given method m located in a class C, as illustrated in Figure 3.1. After that, we compute two similarity coefficients: (a) the average similarity between the set of dependencies established by method m and by the remaining methods in C; and (b) the average similarity between the dependencies established by method m and by the methods in another class C_i. If the similarity measured in the step (b) is greater than the similarity measured in (a), we infer that m is more similar to the methods in C_i than to the methods in its current class C. Therefore, C_i is a potential candidate class to receive m.

In the remainder of this section, we describe the recommendation algorithm proposed in this dissertation (Section 3.1.1), the similarity functions that play a central

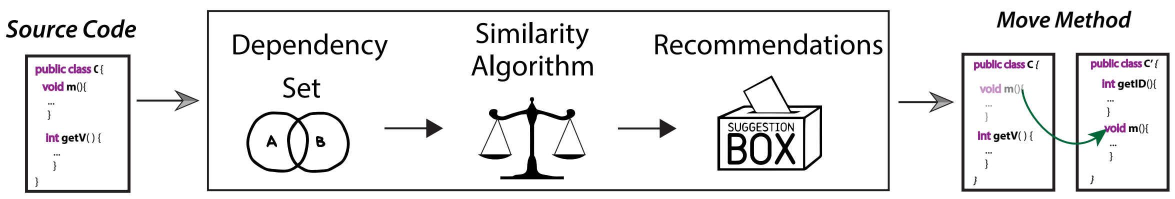


Figure 3.1: Proposed approach

role in this algorithm (Section 3.1.2), and the strategy we propose to decide the most suitable class to receive a particular method (Section 3.1.3).

3.1.1 Recommendation Algorithm

Algorithm 1 presents the proposed recommendation algorithm. Assume a system S with a method m implemented in a class C. For all class $C_i \in S$, the algorithm verifies whether m is more similar to the methods in C_i than to the methods in its original class C (line 6). In the positive cases, we insert C_i in a list T that contains the candidate target classes to receive m (line 7). Finally, we search in T for the most suitable class to receive m (line 10). In case we find such a class C', we make a recommendation to move m to C' (line 11).

Algorithm 1 Recommendation algorithm

```
Input: Target system $S$
Output: List with *Move Method* recommendations
 1: $Recommendations \leftarrow \emptyset$
 2: for all method $m \in S$ do
 3:     $C$ = get_class($m$)
 4:     $T \leftarrow \emptyset$
 5:     for all class $C_i \in S$ do
 6:         if similarity($m, C_i$) > similarity($m, C$) then
 7:             $T = T + C_i$
 8:         end if
 9:     end for
10:     $C'$ = best_class($m, T$)
11:     $Recommendations = Recommendations + move(m, C')$
12: end for
```

This algorithm relies on two key functions: (a) *similarity*(m, C) that computes the average similarity between method m and the methods in a class C; and (b) *best_class*(m, T) that receives a list T of candidate classes to receive m and returns the most suitable one. These functions are described in the following sections.

3.1.2 Similarity Function

Our approach relies on the set of static dependencies established by a method m to compute its similarity with the methods in a class C, as described in Algorithm 2. Initially, we compute the similarity between method m and each method m' in C (line 4). In the end, the similarity between method m and C is defined as the arithmetic mean of the similarity coefficients computed in the previous step. In this algorithm, $NOM(C)$ denotes the number of methods in a class C (lines 8 and 10).

Algorithm 2 Similarity Algorithm

```
Input: Method $m$ and a class $C$
Output: Similarity coefficient between $m$ and $C$
 1: $sim \leftarrow 0$
 2: for all $method\ m' \in C$ do
 3:     if $m \neq m'$ then
 4:         $sim = sim + meth\_sim(m, m')$
 5:     end if
 6: end for
 7: if $f \in C$ then
 8:     return $sim$ / $[NOM(C) - 1]$
 9: else
10:     return $sim$ / $NOM(C)$
11: end if
```

The key function in Algorithm 2 is $\texttt{meth\_sim(m,m')}$, which computes the similarity between the sets of dependencies established by two methods (line 4). Based on an exploratory study where we evaluated 18 similarity coefficients (described in Section 3.2), we decide for the use of the *Sokal and Sneath 2* coefficient [Sokal and Sneath, 1963, 1973; Everitt et al., 2011], defined as:

$$\texttt{meth\_sim(m,m')} = \frac{a}{a + 2(b + c)} \tag{3.1}$$

where

- $a = \mid Dep(m) \ \bigcap \ Dep(m') \mid$
- $b = \mid Dep(m) \ - \ Dep(m') \mid$
- $c = \mid Dep(m') \ - \ Dep(m) \mid$

In this definition, $Dep(m)$ is a set with the dependencies established by method m. This set includes the types the implementation of method m establishes dependency with. More specifically, we consider the following dependencies:

- *Method calls*: if method m calls another method m', the class of m' is added to $Dep(m)$.

- *Field accesses*: if method m reads from or writes to a field f, the type of f is added to *Dep*(m). In the case where f is declared in the same class as m, then the class itself is also added to *Dep*(m).
- *Object instantiations*: if method m creates an object of a type C, then C is included in *Dep*(m).
- *Local declarations*: if method m declares a variable or formal parameter v, the type of v is included in *Dep*(m).
- *Return types*: the return type of m is added to *Dep*(m).
- *Exceptions*: if method m can raise an exception E or if method m handles E internally, then the type of E is added to *Dep*(m).
- *Annotations*: if method m receives an annotation A, then the type of A is included in *Dep*(m). [-0.15cm]

When building the dependency sets we ignore the following types: (a) primitive types; and (b) types and annotations from `java.lang` and `java.util` (like `String`, `HashTable`, `Object`, and `SupressWarnings`). Since virtually all classes establish dependencies with these types, they do not actually contribute for the measure of similarity. This decision is quite similar to text retrieval systems that exclude stop words because they are rarely helpful in describing the content of a document.

Example: Code 3.1 shows a method `foo` located in a class `Bar`, whose dependency set is extracted as follows:

$$Dep(foo) = \{\ Annot, R, B, Z, A, C, D, Bar\ \}$$

```
public class Bar {
  private C c;
  @Annot
  public R foo(B b) throws Z {
      A a = b.getA();
      if (a != null) {
          a.valueR = this.c.getD().valueR;
      }
      return a.valueR;
  }
}
```

Code 3.1: Example to illustrate dependency sets

Next, we explain why each type was included in the dependency set: `Annot` (annotation used in line 3), `R` (return type in line 4), `B` (parameter declared in line 4 and method call in line 5), `Z` (exception declaration in line 4), `A` (local variable declaration in line 5 and field access in lines 7 and 9), `C` (method call in line 7), `Bar` and `D` (field access in line 7).

It is worth noting that *Dep*(m) is a set—and not a multiset. Therefore, multiple dependencies to the same type are represented only once. As an example, *Dep*(*foo*) has a unique reference to `B`, even though there is a formal parameter of type `B` (line 4) and a method call having a target of type `B` (line 5). Fundamentally, Terra et al. [2013a] have provided evidences that traditional sets achieve better precision results than multisets when evaluating the structural similarity of program entities.

For the sake of clarity, we omitted from Algorithm 2 a test that discards two kinds of methods when calculating the similarity of dependency sets:

- A method m that is the only method in its class because our approach is based on the similarity between a given method and the remaining methods in the class.

- A method m whose dependency set has less than four dependencies because methods with few dependencies are more subject to imprecise or spurious similarity measures. Moreover, by establishing this lower threshold, we also filter out accessor methods (*getters* and *setters*), which by their very nature are rarely implemented in incorrect locations.

3.1.3 Target Class Selection

Assume that T is a list with classes $C_1, C_2, \ldots, C_n$ that are more similar to a method m than its current class C, as computed by Algorithm 1. Assume also that the classes in T are in descending order by their similarity with method m—i.e., most similar classes first—as computed by Algorithm 2.

To reduce the chances of false positives we created a filter to avoid recommendations between two very similar classes. A move recommendation is *not* created when: (i) T has less than three classes, and (ii) the difference between the similarity coefficient value of C_1 (the first class in the list) and C (the original class of m) is less than or equal to 25%. In such cases, we consider that the difference between the dependencies established by C_1 and C is not discrepant enough to recommend a move operation.

On the other hand, when such conditions do *not* hold, a recommendation *move*(m, C_i) is created for the first class $C_i \in T$ that satisfies the preconditions of the *Move Method* refactoring ($1 \leq i \leq n$). Basically, as usual in the case of refactorings [Opdyke, 1992; Fowler, 1999], a *Move Method* has its application conditioned by a set of preconditions, fundamentally to ensure that the program's behavior is preserved after the refactoring. For example, the target class C_i should not contain a method with the same signature as method m. When such preconditions are not satisfied by a pair (C, C_i), we automatically verify the next pair (C, C_{i+1}). No recommendation is returned when the refactoring preconditions fail for all pair of classes.

Our approach currently relies on preconditions of the *Move Method* automatic refactoring engine provided by the Eclipse IDE. However, it is well-known that Eclipse implements weak preconditions for some refactorings [Steimann and Thies, 2009; Schäefer and de Moor, 2010; Soares et al., 2013]. For this reason, we strengthened the Eclipse preconditions by supporting the following five preconditions originally proposed by Tsantalis and Chatzigeorgiou [2009]:

- The target class should not inherit a method having the same signature of the moved method.
- The method to be moved should not override an inherited method in its original class.
- The method to be moved should not be synchronized.
- The method to be moved should not contain assignments to its original class fields (including inherited fields).
- The method to be moved should have a one-to-one relationship with the target class.

3.1.4 The Approach in Action

In this section, we describe an illustrative example in order to demonstrate the mechanics of our approach.

Problem: Assume a hypothetical web-based system that has a method intentionally implemented in a wrong class, i.e., a method that violates the desired system design. Specifically, we implemented a method responsible for database issues—such as obtaining the connections, mapping Java objects to SQL data types, and performing SQL commands—in class `CustomerView`, as illustrated in Code 3.2. However, `CustomerView` belongs to the presentation layer, which is responsible for UI (User Interface) whereas the method should belong to a class in the persistence layer.

```
1:public class CustomerView extends JFrame {
    ... declaration of variables
2:  public CustomerView() {
        ... piece of code
3:      JButton btnNewButton = new JButton("Find");
4:      btnNewButton.addActionListener(new ActionListener() {
5:          public void actionPerformed(ActionEvent e) {
6:              ... piece of code
7:          }
8:      });
9:  }

10:  public List<Customer> getAllCustomers(DB db) throws SQLException {
11:     List<Customer> result = new ArrayList<Customer>();
12:     Connection con = db.getConnection();
13:     PreparedStatement ps =
14:                       con.prepareStatement("select * from CUSTOMER");
15:     ResultSet rs = ps.executeQuery();

16:     while (rs.next()){
17:         result.add(new Customer(rs.getInt("ID"),rs.getString("NAME")));
18:     }
19:     rs.close();
20:     ps.close();
21:     con.close();
22:     return result;
23: }
24:}
```

Code 3.2: Class *CustomerView*

Class `CustomerView` defines three methods: (i) constructor *CustomerView* (lines 2–9), (ii) *actionPerformed* (lines 5–7), and (iii) *getAllCustomers* (lines 10–23). Although the first two methods actually belong to the presentation layer, *getAllCustomers* should be defined in the persistence layer. One can argue that *actionPerformed* is a method defined in an inner class. However, our approach considers every method in a class to measure similarity. It includes special methods, such as accessors methods, constructors (e.g., *CustomerView*), and methods in inner classes (e.g., *actionPerformed*). Although we consider these methods, they are not liable to be moved and therefore our approach does not trigger recommendations for moving them.

Code 3.3 presents class `CustomerDAO`, a class in the persistence layer where our subject method *getAllCustomers* should be defined according to the desired design. The assumption behind our approach is that methods that rely on the same types should be defined in the same class. For instance, *getAllCustomers* establishes dependency with `BD` and `SQLException`, which are types used by every method in `CustomerDAO`. For clarification purposes, Appendix A presents the complete code of classes `CustomerView` and `CustomerDAO`.

```
public final class CustomerDAO {

  public List<Customer> getCities(DB con) throws SQLException {
      ... piece of code
  }
  public String getCityOfOrigin(DB con, Ticket tic) throws SQLException {
      ... piece of code
  }
  public String getDestinyCity(DB con, Ticket tic) throws SQLException {
      ... piece of code
  }
}
```

Code 3.3: Class CustomerDAO

Solution: We first calculate the set of static dependencies established by every method in the system since our approach measures the similarity between a given method m and its current class and also between m and all other classes of the system. In this example, we focus on the measure of the similarity (i) between method *getAllCustomers* and its current class CustomerView and (ii) between method *getAllCustomers* and class CustomerDAO, the most appropriate class according to the desired system design. Table 3.1 reports the dependency sets for the methods in classes CustomerView and CustomerDAO, which were extracted by the $Dep(m)$ function defined in Section 3.1.2.

CustomerView		**CustomerDAO**	
Method	*Dependency Set*	*Method*	*Dependency Set*
getAllCustomers	{SQLException, Connection, Customer, PreparedStatement, DB, ResultSet}	*getCities*	{SQLException, Connection, Customer, PreparedStatement, DB, ResultSet}
CustomerView	{CustomerView, JMenu, ActionListener, JMenuBar, JPanel, JFrame, JMenuItem, JTextField, Window, JButton, JTextPane}	*getCityOfOrigin*	{SQLException, Connection, Ticket, PreparedStatement, DB, ResultSet}
actionPerformed	{CustomerView, JTextField, Customer, ActionEvent}	*getDestinyCity*	{SQLException, Connection, Ticket, PreparedStatement, DB, ResultSet}

Table 3.1: Dependency sets for the methods in CustomerView and CustomerDAO

Algorithm 2 (as defined in Section 3.1.2) defines that the similarity between a method m and a class C—i.e., $similarity(m, C)$—is the arithmetic mean of the similarity values between m and all other methods in C. In the case of our example, it is necessary to calculate $meth\_sim(getAllCustomers, CustomerView)$ and $meth\_sim(getAllCustomers, actionPerformed)$ in order to obtain the $similarity(getAllCustomers, CustomerView) = 0.029$, as reported in Table 3.2. Similarly, this same procedure is performed for every class in the system. As an example, Table 3.2 also reports the similarity values measured for `CustomerDAO`.

CustomerView	**CustomerDAO**
$meth\_sim(getAllCustomers, CustomerView) = 0.000$ $meth\_sim(getAllCustomers, actionPerformed) = 0.058$	$meth\_sim(getAllCustomers, getCities) = 1.000$ $meth\_sim(getAllCustomers, getCityOfOrigin) = 0.550$ $meth\_sim(getAllCustomers, getDestinyCity) = 0.550$
$\boldsymbol{similarity(getAllCustomers, CustomerView) = 0.029}$	$\boldsymbol{similarity(getAllCustomers, CustomerDAO) = 0.700}$

Table 3.2: Similarity values for classes `CustomerView` and `CustomerDAO`

Next, Algorithm 1 (as defined in Section 3.1.1)creates a ranked list of candidate classes T to receive $getAllCustomers$. In short, T contains classes that have similarity values greater than the similarity between $getAllCustomers$ and its current class `CustomerView` (0.029), as can be observed next:

$T = \{[\texttt{CustomerDAO}, 0.70],\ [\texttt{ProductDAO}, 0.55], [\texttt{DepartmentDAO}, 0.55],\ ...\ \}$

As expected, `CustomerDAO` is the most similar class throughout the system. In fact, list T contains other classes that implement database functionalities (e.g., `ProductDAO` and `DepartmentDAO`). However, function `best_class(m, T)` chooses the most similar class that complies with all preconditions of the *Move Method* refactoring. Since `CustomerDAO` has the highest similarity value in T and it also satisfies the preconditions, our approach precisely suggests moving method $getAllCustomers$ to class `CustomerDAO`.

3.2 Exploratory Study: Similarity Coefficients

In this section, we report an exploratory study aims to answer the following research questions:

- ***RQ #1*** – What is the most suitable similarity coefficient for our approach?

- ***RQ #2*** – What is the impact of the threshold used to filter out recommendations involving very similar classes?

In this study, we evaluated the 18 similarity coefficients described in Table 3.3, where a, b, and c are as previously defined in Section 3.1.2, and d is defined as follows:

$$d = |\ Dep(S) - \ [Dep(m)\ \cup\ Dep(m')]\ | \tag{3.2}$$

where $Dep(S)$ is a set with the dependencies established by all methods in the system S under analysis and $Dep(m)$ is a set with the dependencies established by method m.

Table 3.3: Similarity Coefficients (Extracted from [Terra et al., 2013a])

Coefficient	Definition	Range
1. Jaccard	$a/(a+b+c)$	0–1*
2. Simple matching	$(a+d)/(a+b+c+d)$	0–1*
3. Yule	$(ad-bc)/(ad+bc)$	-1–1*
4. Hamann	$[(a+d)-(b+c)]/[(a+d)+(b+c)]$	-1–1*
5. Sorenson	$2a/(2a+b+c)$	0–1*
6. Rogers and Tanimoto	$(a+d)/[a+2(b+c)+d]$	0–1*
7. Sokal and Sneath	$2(a+d)/[2(a+d)+b+c]$	0–1*
8. Russelll and Rao	$a/(a+b+c+d)$	0–1*
9. Baroni-Urbani and Buser	$[a+(ad)^{\frac{1}{2}}]/[a+b+c+(ad)^{\frac{1}{2}}]$	0–1*
10. Sokal binary distance	$[(b+c)/(a+b+c+d)]^{\frac{1}{2}}$	0*–1
11. Ochiai	$a/[(a+b)(a+c)]^{\frac{1}{2}}$	0–1*
12. Phi	$(ad-bc)/[(a+b)(a+c)(b+d)(c+d)]^{\frac{1}{2}}$	-1–1*
13. PSC	$a^2/[(b+a)(c+a)]$	0–1*
14. Dot-product	$a/(b+c+2a)$	0–1*
15. Kulczynski	$\frac{1}{2}[a/(a+b)+a/(a+c)]$	0–1*
16. Sokal and Sneath 2	$a/[a+2(b+c)]$	0–1*
17. Sokal and Sneath 4	$\frac{1}{4}[a/(a+b)+a/(a+c)+d/(b+d)+d/(c+d)]$	0–1*
18. Relative Matching	$[a+(ad)^{\frac{1}{2}}]/[a+b+c+d+(ad)^{\frac{1}{2}}]$	0–1*

The symbol "$*$" denotes the maximum similarity.

3.2.1 Study Design

We conducted the study using JHotDraw (version 7.6), which is a system commonly used to illustrate object-oriented programming practices and patterns.[1] Its implementation has 80 KLOC, 674 classes, and 6,533 methods. The system has a reputation

[1]http://www.randelshofer.ch/oop/jhotdraw/

of having a well-defined and mature design, proposed and implemented by expert developers [Riehle, 2000]. For this reason, our study is based on the conjecture that *all methods in JHotDraw are implemented in their correct classes*.

We executed Algorithm 1 (described in Section 3.1) multiple times to JHotDraw, considering in each execution a different similarity coefficient. Based on our conjecture, any recommendation should be flagged as a false positive. Therefore, we aim to select the similarity coefficient that generates the lowest number of recommendations in JHotDraw.

3.2.2 Similarity Coefficient Results

Figure 3.2 presents the number of recommendations generated by each similarity coefficient. For the sake of readability, the results for the coefficients that generated more than 100 recommendations (*Hamann*, *Rogers and Tanimoto*, *Sokal and Sneath*, *Simple Matching*, and *Sokal Binary Distance*) are reported separately in Figure 3.3.

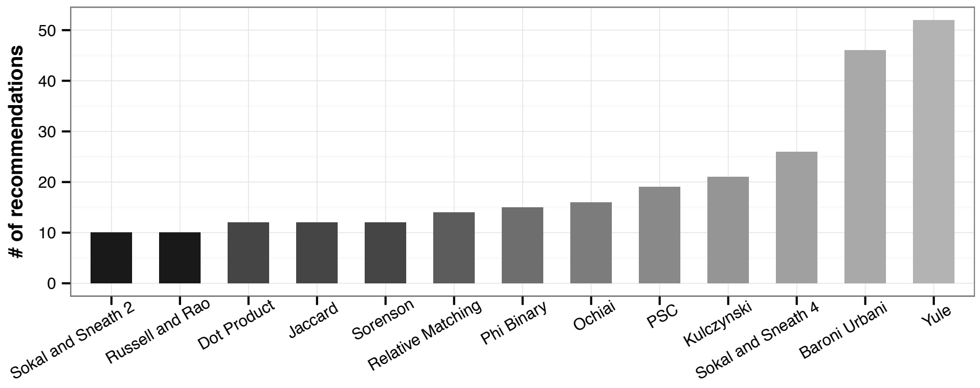


Figure 3.2: Recommendations in JHotDraw for the best coefficients

As reported in Figure 3.2, the best coefficients were *Sokal and Sneath 2* and *Russell and Rao*, both with 10 recommendations. We decided to use *Sokal and Sneath 2* because it is simpler to compute than *Russell and Rao* whose computation requires counting the remaining dependencies in the system that do not occur in a given pair of methods m and m' (variable d in Equation 3.2).

3.2.3 Filter's Threshold Impact

As explained in Section 3.1.3, we proposed the use of a filter to avoid recommendations between two very similar classes. We conducted a preliminary study that showed that 76.28% of methods in JHotDraw are ranked among the three most similar classes of

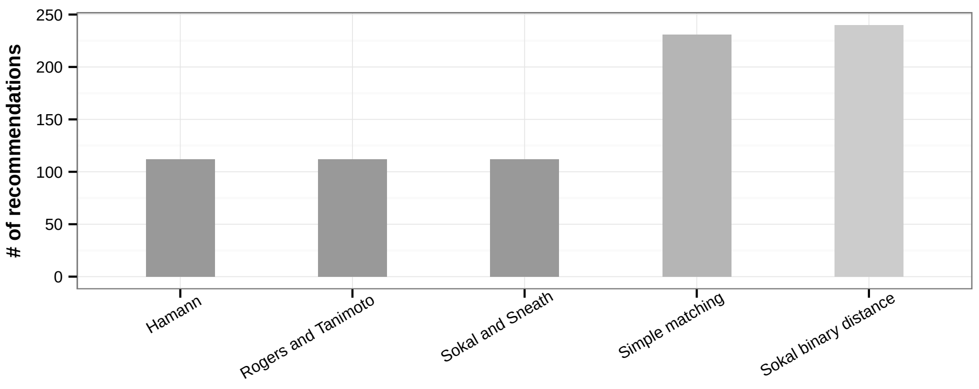


Figure 3.3: Recommendations in JHotDraw for the worst coefficients

the entire system. In other words, if a method is in the most appropriate class, it has 76.28% of chances of being located among the three more similar classes considering all classes in JHotDraw.

However, in order to do not miss valuable recommendations, we restricted the filter with a threshold that represents the maximum acceptable similarity difference when the current class of the method is among the three most similar classes. Figure 3.4 shows the impact of changes on this filter's threshold. Based on the results, we decided to adopt the value of 25% because it is the less restrictive value among the filter's threshold that achieved the best possible result (10 recommendations).

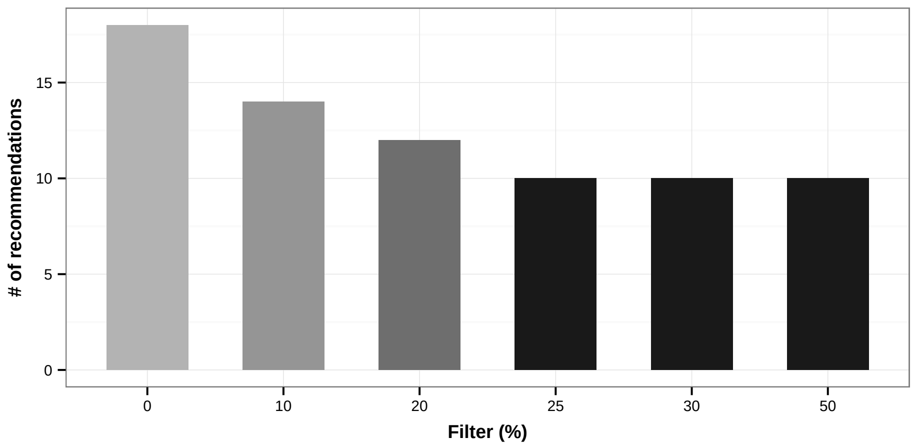


Figure 3.4: Recommendations in JHotDraw using distinct filter's thresholds

3.2.4 Qualitative Analysis

Furthermore, we analyzed each suggested recommendation on JHotDraw, in order to confirm if they were indeed incorrect recommendations. Particularly, we could classify the 10 incorrect recommendations in two well-distinguishing cases: (i) filtering and (ii) semantically unrelated problems. Next, we discuss some recommendations in detail.

Filtering: Most incorrect suggestions (6 out of 10)—are related to small methods. More specifically, these recommendations refer to five methods that handle events and one setter method. In these cases, our approach suggested the move of these methods to classes that represent event or data types, which is notably wrong. As an example, Code 3.4 presents method `selectionChanged`—which is implemented in class `ApplyAttributesAction`—but our approach has suggested to move to class `FigureSelectionEvent`—which is an event type.

```
1:public void selectionChanged(FigureSelectionEvent evt) {
2:  setEnabled(getView().getSelectionCount() == 1);
3:}
```

Code 3.4: A wrong recommendation in a event handler method in JHotDraw

In fact, our filter for small methods has failed to exclude these methods. As explained in Section 3.1.2, to exclude small methods like getters, setters and handlers, we decide to do not recommend *Move Method* refactorings for methods that have less than four dependencies. However, as can be observed in Code 3.4, the method `selectionChanged` has exactly four dependencies: a dependency with the formal parameter `FigureSelectionEvent` (line 1); and three more accessing dependencies (line 2). More specifically, there are accesses to method `setEnabled` from its superclass `AbstractAction`, to method `getView` from another superclass `AbstractSelectedAction`, and to method `getSelectionCount` from class `DrawingView`, which is the type of the object returned by `getView`.

Indeed, the same problem occurred on the other five methods in this classification, i.e., our approach failed to filter these small methods. A potential solution might be to raise the threshold to exclude methods that have less than five dependencies. However, we also noticed in further studies that we could miss correct suggestions if we reduce the threshold. For this reason, we decided to keep the threshold to filter out methods with less than four dependencies since it has presented itself as the best cost benefit solution.

Semantically unrelated: The remaining recommendations are related to methods that are structurally similar to a class C', but semantically they should *not* be moved to C'. This classification is frequently associated to some design patterns. In practice, there are design patterns that deliberately violate specific design guidelines, e.g., the methods that belong to a facade are not cohesive since they fundamentally forward incoming requests to other classes [Gamma et al., 1995].

As an example, Code 3.5 presents method `findFiguresWithin` that belongs to class `QuadTreeCompositeFigure`, which follows the Composite design pattern. This pattern describes a group of objects that should be treated in the same way as a single instance of an object [Gamma et al., 1995]. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. The implementation of a composite pattern allows clients to treat both individual objects and compositions in the same way. In this example, the `QuadTreeCompositeFigure` class abstracts all events fired by the `Figure` objects contained in the `Drawing` objects.

```
public List<Figure> findFiguresWithin(Rectangle2D.Double bounds) {
  LinkedList<Figure> contained = new LinkedList<Figure>();
  for (Figure f : children) {
      Rectangle2D r = f.getBounds();
      if (f.get(TRANSFORM) != null) {
          r = f.get(TRANSFORM).createTransShape(r).getBounds2D();
      }
      if (f.isVisible() && bounds.contains(r)) {
          contained.add(f);
      }
  }
  return contained;
}
```

Code 3.5: `findFiguresWithin` method

Method `findFiguresWithin` is responsible for finding all `Figures` objects, among their children, within a given rectangular frontier. Our recommendation algorithm suggests moving this method to class `QuadTree`. Although `findFiguresWithin` is indeed structurally more similar to the methods of class `QuadTree`, the suggestion is semantically incorrect. Class `QuadTree` is a data structure for performing searches by partitioning a two-dimensional space into four square regions. Due to the nature of the `QuadTree` class, most of its methods have a dependency with `Rectangle2D.Double` and `Rectangle2D` classes. As can be observed in Code 3.5, these classes denote two out of the six dependencies established by `findFiguresWithin` and, for this reason, this method is very similar to class `QuadTree`.

A potential solution to address such problem is to define rules to identify and filter methods that are part of design pattern scenarios. Thus, our approach would not trigger any incorrect recommendation for these scenarios. Such a solution is quite similar to the one proposed by Seng et al. [2006] to identify methods that should not be moved. The authors propose an approach to identify the role of design elements by describing their static structure and naming conventions. As an example, static factory methods are described as having the static modifier, having a non-void return type, and containing a call to a constructor that creates objects whose type is equal to the return type of the method.

3.2.5 Threats to Validity

There are two main threats regarding the validity of this initial study. First, as usual, we cannot extrapolate our results to other systems (external validity). However, it is not simple to find systems that confessedly have a mature design, as JHotDraw. Second, even in JHotDraw, it is possible to have methods that are in fact not implemented in the most recommended class (conclusion validity). On the other hand, due to the particular motivation and context behind JHotDraw's implementation, we claim that the number of such misplaced methods is really small.

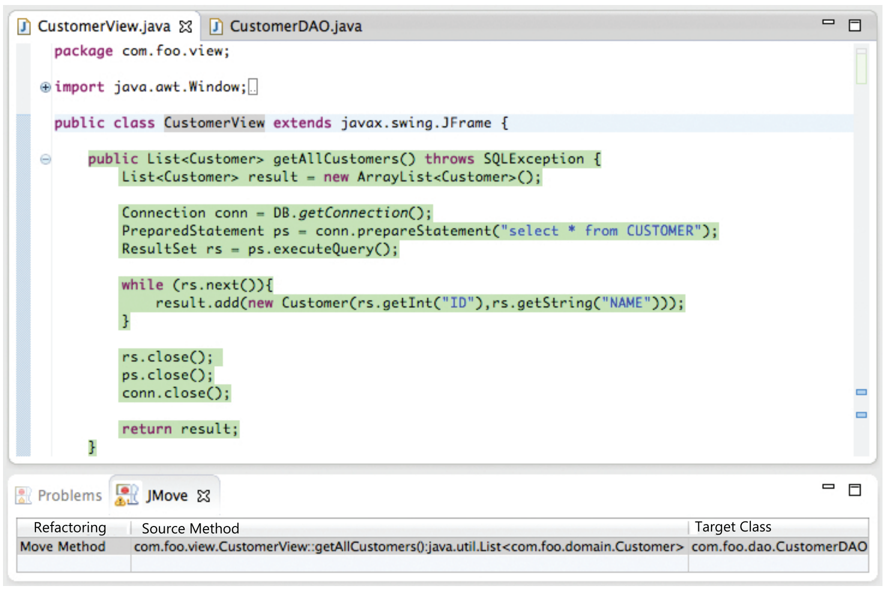
3.3 Tool Support

We implemented a prototype tool, called JMove,[2] that supports the approach proposed in this dissertation. Basically, JMove is an Eclipse plug-in that implements Algorithms 1 and 2 (previously defined in Section 3.1).

Figure 3.5 illustrates the tool's interface using our previously illustrated example (Section 3.1.4) where method *getAllCustomers* from the persistence layer was purposely implemented in class `CustomerView` from the presentation layer. As can be noticed, JMove adds a new report panel to the IDE interface, which is used to display the recommended *Move Method* refactorings. Regarding the tool architecture, Figure 3.6 illustrates the main modules from JMove's implementation. First, module *Dependency Extraction* is used to extract the dependency sets that characterize the methods in our approach. In our running example, method *getAllCustomers* has the following dependency set:

$$\begin{aligned} Dep(getAllCustomers) = \{ & SQLException, Connection, PreparedStatement, DB \\ & ResultSet, Customer\} \end{aligned}$$

[2]JMove is publicly available at: http://aserg.labsoft.dcc.ufmg.br/jmove/



```java
package com.foo.view;

import java.awt.Window;

public class CustomerView extends javax.swing.JFrame {

    public List<Customer> getAllCustomers() throws SQLException {
        List<Customer> result = new ArrayList<Customer>();

        Connection conn = DB.getConnection();
        PreparedStatement ps = conn.prepareStatement("select * from CUSTOMER");
        ResultSet rs = ps.executeQuery();

        while (rs.next()){
            result.add(new Customer(rs.getInt("ID"),rs.getString("NAME")));
        }

        rs.close();
        ps.close();
        conn.close();

        return result;
    }
```

Refactoring	Source Method	Target Class
Move Method	com.foo.view.CustomerView::getAllCustomers():java.util.List<com.foo.domain.Customer>	com.foo.dao.CustomerDAO

Figure 3.5: JMove's interface

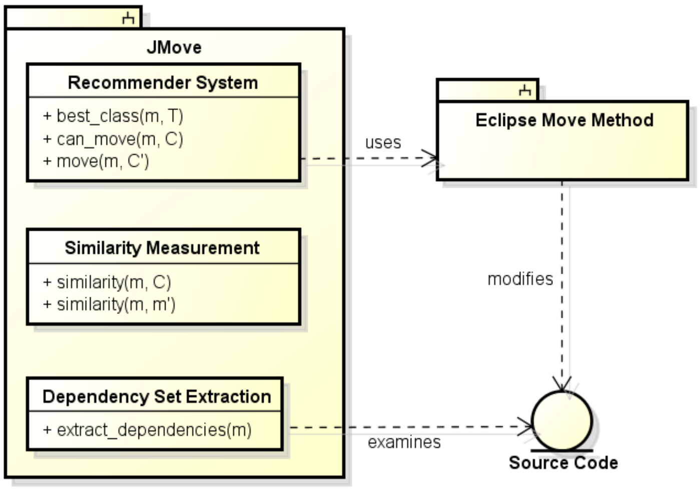


Figure 3.6: JMove's architecture

Second, module *Similarity Measurement* implements Algorithms 1 and 2. In our running example, this module computes a list T with the following candidate target classes to receive *getAllCustomers*:

$$T = \{[\texttt{CustomerDAO}, 0.70],\ [\texttt{ProductDAO}, 0.55],\ [\texttt{DepartmentDAO}, 0.55],\ \ldots\ \}$$

Third, module *Recommender System* implements function $best\_class(m, T)$. Given a list T of candidate classes, this function selects the most suitable one to receive method m, according to the criteria defined in Section 3.1.3. As already reported in

Section 3.1.4, this function returns class `CustomerDAO` since it has the highest number of methods similar to *getAllCustomers*. In other words, *getAllCustomers* is more similar—based on its dependency set—to the methods in `CustomerDAO` than to the methods in its current class `CustomerView`.

3.4 Final Remarks

This chapter presented our recommendation approach to detect methods located in incorrect classes and to suggest moving such methods to more suitable ones. We first evaluate the set of static dependencies established by a given method m located in a class C and then suggest moving to a more suitable class C', if there is one.

In Section 3.1, we presented the proposed recommendation system, including the selection of the most appropriate class for a method, underlying algorithms and similarity functions, and an illustrative example of our recommendation system. In Section 3.2, we reported a quantitative study on JHotDraw that compares 18 similarity coefficients to identify that *SokalandSneath*2 is the most appropriate coefficient to determine where a method should be located. We conclude that the coefficient *SokalandSneath*2 is the most suitable for our approach. We also conducted a qualitative analysis to understand the incorrect recommendations, which were verified filtering and semantic issues. Last, in Section 3.3, we presented JMove, an Eclipse plug-in that implements our proposed approach.

In the next chapter, we describe the evaluation of our recommendation system. In short, we present and discuss results on applying our approach (i) in 14 open-source systems using a synthesized dataset and (ii) in two real-world systems in which the system's experts evaluated the *Move Method* recommendations.

Chapter 4

Evaluation

Chapter 3 presented the proposed recommendation system, including a description of the underlying algorithms and similarity functions, an exploratory study to select the most appropriate coefficient, and a tool—called JMove—that implements the proposed approach.

We divided this chapter into four main sections. In Section 4.1, we compare our approach with JDeodorant and inCode using 14 open-source systems where we have artificially created the methods to be moved. In Section 4.2, we compare our approach with JDeodorant and inCode using two systems where the systems' expert evaluated the triggered recommendations. In Section 4.3, we describe a performance comparison of JMove, JDeodorant, and inCode. Finally, Section 4.4 concludes.

4.1 Evaluation with Open-Source Systems

This section is divided into five parts, Section 4.1.1 states the research questions we aim to answer. Section 4.1.2 presents the dataset used in our evaluation and the methodology we followed to generate our gold sets (for precision and recall measuring). Section 4.1.3 exhibits the precision and recall achieved by the approaches. Section 4.1.4 discusses three examples of *Move Method* refactorings suggested by our approach in the JHotDraw system. Finally, Section 4.1.5 lists threats to validity.

4.1.1 Research Questions

This first evaluation aims to answer the following research questions:

- ***RQ #1*** – How does our approach compare with JDeodorant and inCode in terms of precision?

- ***RQ #2*** – How does our approach compare with JDeodorant and inCode in terms of recall?

Both questions compare our approach with JDeodorant and inCode for three reasons: (a) JDeodorant is a well-known solution for identifying *Move Method* refactoring opportunities; (b) inCode is a commercial solution for identifying design flaws. It is worth noting that inCode is a lightweight and more affordable version of inFusion (refer to Section 2.2.4), which behaves exactly like inFusion to detect Feature Envy instances; and (c) both tools are robust and friendly enough to be used in real-world systems.

4.1.2 Study Design

In this section, we present the dataset used in our evaluation and the methodology we followed to generate our gold sets (i.e., the methods implemented in incorrect classes).

Dataset: We evaluate our approach using the systems in the Qualitas.*class* Corpus [Terra et al., 2013b]. This corpus is a compiled version of 111 systems originally included in the Qualitas Corpus [Tempero et al., 2010], which solely provides the source code of the systems. However, for evaluating tools that depend on the Abstract Syntax Tree (AST) provided by a given IDE, as the one considered in this paper, we need to import and compile the source code. However, this effort is not trivial in the case of systems with many external dependencies. For this reason, we decided to use the aforementioned parallel project which aimed to create a compiled variant of the Qualitas Corpus.

Particularly, our evaluation considers only a sample of the systems in Qualitas.*class* Corpus. We selected the systems according to the following criteria: (a) we only considered active projects to avoid outdated systems; (b) we only considered systems having between 500 and 5,000 classes to filter out small and huge systems; (c) we only considered systems whose implementation consists of a single Eclipse project (otherwise we would have to execute the tools multiple times for each project). Among the systems attending our criteria, we selected the first 15 systems, sorted according to their release date from newest to oldest, as stated in the original Qualitas Corpus documentation. For our initial systems selection, JDeodorant did not raise recommendations for two systems: Lucene and iReport. We therefore decided to remove these systems from our sample. Finally, we included JHotDraw to the sample, due to the same reasons that motivated its use in our first study (Section 3.2).

Table 4.1 presents the final 14 systems considered in the study, including their names, version, number of classes (NOC), number of methods (NOM), and size in terms of lines of code (LOC).

Table 4.1: Target Systems

System	**Version**	**NOC**	**NOM**	**LOC**
Ant	1.8.2	1,474	12,318	127,507
ArgoUML	0.34	1,291	8,077	67,514
Cayenne	3.0.1	2,795	17,070	192,431
DrJava	r5387	788	7,156	89,477
FreeCol	0.10.3	809	7,134	106,412
FreeMind	0.9.0	658	4,885	52,757
JMeter	2.5.1	940	7,990	94,778
JRuby	1.7.3	1,925	18,153	243,984
JTOpen	7.8	1,812	21,630	342,032
Maven	3.0.5	647	4,888	65,685
Megamek	0.35.18	1,775	11369	242,836
WCT	1.5.2	539	5,130	48,191
Weka	3.6.9	1,535	17,851	272,611
JHotDraw	7.6	674	6,533	80,536

Gold Sets: To evaluate precision and recall, it is crucial to identify the methods implemented in the wrong classes, which we refer as the gold sets [Dit et al., 2013]. Typically, the task of generating such sets would require the participation of expert developers on the target systems in order to manually analyze and classify each method. However, in the context of open-source systems, it is not straightforward to establish a contact with the key project developers. For this reason, inspired by the evaluation proposed by Moghadam and Ó Cinnéide [2012] in a work on design-level refactoring, we manually synthesized a version of each system with well-known methods implemented (at least a high probability) in incorrect classes.

More specifically, we randomly selected a sample including 3% of the classes in each system and manually moved a method from them to new classes, also randomly selected. Before each manual move, we verified the following preconditions: (a) the method selected to be moved must have at least four dependencies in its dependency set because our approach automatically filters out methods not satisfying this condition (as described in Section 3.1.2); (b) a given class (source or target) can be selected at most once in order to avoid multiple moves to or from the same class;[1] and (c) given a

[1]In practice, multiple moves to or from the same class reduce the chances of returning the moved

method m in a class C and a target class C', it must be possible to move m to C' and also back to C. This last precondition is important because otherwise our approach—and also JDeodorant—would never make a recommendation about m in the synthesized system. On the other hand, inCode is not affected by this last precondition because his Feature Envy detection technique does not verify *Move Method* preconditions. Finally, when a given method and a target class do not attend the proposed preconditions, a new candidate is randomly generated, until reaching 3% of the classes in the system.

By following this procedure, we synthesized for each system S a modified system S' with a well-known *GoldSet* of methods with a high probability to be located in the wrong class for the reasons described next:

- In the case of JHotDraw, as claimed in Section 3.2, it is reasonable to consider that *all methods in the original system are in their correct class* since JHotDraw was developed and maintained by a small number of expert developers. Therefore, we argue that in the modified version of the system, the methods in the *GoldSet* are the only ones located in wrong classes. Due to this special condition, for JHotDraw, we generated five instances of the system with well-known *GoldSet*.

- In the case of the other systems, it is not reasonable to assume that all methods are in the correct classes because such systems are maintained and evolved by developers with different levels of proficiency (i.e., ranging from beginners to experts). However, we argue that it is reasonable to assume that at least *most methods are in the correct classes.* More specifically, the number of methods in such systems range from 4,885 methods (FreeMind) to 21,630 methods (JTOpen). As a consequence, in a sample with such considerable number of methods, the probability of randomly selecting one located in the correct class is much higher than otherwise.

In fact, the proposed procedure only inserts an invalid method in a synthesized *GoldSet* when the following two unlikely conditions hold for a randomly selected method m and a randomly selected target class C': (a) m is originally implemented in a wrong class in the original system; and (b) C' is exactly the class where this method should have been implemented. For example, when condition (a) holds, but condition (b) does not hold, we still have a valid method in the *GoldSet* because we are basically moving a method implemented in a wrong class to another class that is also not correct.

methods to their original class. For instance, assume a class in which all methods were moved to another class. In this case, it would be unlikely to recommend the return of such methods to the original one, an empty class.

Table 4.2 shows the number of methods in the gold sets generated for each system. In total, using the Eclipse support to *Move Method* refactorings, we moved 475 methods, including 100 methods in the five JHotDraw instances.

Table 4.2: Gold Sets size

System	$\mid GoldSet \mid$	**System**	$\mid GoldSet \mid$
Ant	25	JRuby	41
ArgoUML	32	JTOpen	39
Cayenne	47	Maven	24
DrJava	18	Megamek	35
FreeCol	17	WCT	29
FreeMind	12	Weka	31
JMeter	25	JHotDraw	20

Calculating Precision and Recall: We executed JMove, JDeodorant, and inCode in the modified systems. Each execution generated a list of recommendations *Rec*, whose elements are triples (m, C, C') expressing a suggestion to move m from C to C'. A recommendation (m, C, C') is classified as a true recommendation when it matches a method in the *GoldSet*. Particularly, for a list of recommendations *Rec* generated by JMove, JDeodorant, or inCode, and a given *GoldSet*, the set of true recommendations is defined as:

$$TrueRec = \{ \ (m, C, C') \in Rec \mid \exists (m, C, C') \in GoldSet \ \}$$

It is worth noting that suggestions from JMove and JDeodorant contains only one destination class for a method, whereas suggestions from inCode are in format (m, C, T), where T is a list of classes $C_1, C_2, \ldots, C_n$. In others words, inCode can suggest moving a method to several classes in the same suggestion. It occurs because inCode shows, for a Feature Envy code smell, attributes from one or more external classes used by method m. For this reason, we consider a suggestion (m, C, T) from inCode as correct when there is any $C' \in T$ matching a method in the *GoldSet*.

Furthermore, in the case of the Qualitas.*class* systems, we cannot assume that the methods in the gold sets are the only ones implemented in incorrect classes. For this reason, we measure precision only for the five instances of JHotDraw, as follows:

$$Precision = \frac{\mid TrueRec \mid}{\mid Rec \mid}$$

In all other systems, we calculated a first recall measure defined as the ratio of the methods covered with the evaluated tools by the number of methods in the *GoldSet*, as follows:

$$Recall_1 = \frac{\mid TrueRec \mid}{\mid GoldSet \mid}$$

We also calculated a second recall, defined as:

$$Recall_2 = \frac{\mid \{ \ (m, C, *) \in Rec \mid \exists (m, C, *) \in GoldSet \ \} \mid}{\mid GoldSet \mid}$$

This second definition considers the ratio of methods covered by recommendations suggesting moving m to any other class (denoted by a $*$), which is not necessarily the correct one. Thus, we always have $Recall_1 \ \leq \ Recall_2$.

4.1.3 Quantitative Results

In this section, we provide answers for the proposed research questions.

RQ #1 (Precision): Table 4.3 shows the precision results for the five instances of JHotDraw. Our approach—as implemented by the JMove tool—achieved an average precision of 60.63% against 26.47% achieved by JDeodorant and 12.68% by inCode. As reported in Table 4.4, JMove generated less recommendations than JDeodorant (26.2 x 39.2 recommendations on average). Besides that, we generated more true recommendations (15.8 x 10.4 true recommendations on average). In fact, inCode produced the lowest number of recommendations (11.60 recommendations on average), but achieved the worst precision (1.6 true recommendations on average). On the other hand, in the best case (instance #4), JMove achieved a precision of 66.67%.

RQ #2 (Recall): Table 4.5 shows $Recall_1$ and $Recall_2$ for each system. Regarding the JHotDraw system, the results presented in this table are the average of the recall considering the five instances we generated for this system.

Considering $Recall_1$, JMove achieved a recall of 81.07 ± 5.88 (average plus/minus standard deviation), JDeodorant achieved a result of 54.36 ± 11.83, and inCode achieved 12.35 ± 10.35. Therefore, on average, our results are 49.13% better than JDeodorant and 556.43% better than inCode in terms of recall. Moreover, our minimal recall was 70.73% (JRuby) and our maximal recall was 91.67% (FreeMind). On the

Table 4.3: Precision Results for JHotDraw

System	*Precision* (%)		
	JMove	**JDeodorant**	**inCode**
JHotDraw #1	57.14	25.64	28.57
JHotDraw #2	59.26	25.64	16.67
JHotDraw #3	57.14	21.05	0
JHotDraw #4	66.67	29.27	9.09
JHotDraw #5	62.96	30.77	9.09
Average	60.63	26.47	12.68
Std Dev	4.13	3.78	10.67
Median	59.26	25.64	9.09
Max	66.67	30.77	28.57
Min	57.14	21.05	0

Table 4.4: Number of Recommendations for JHotDraw

System	| *Recommendations* |			| *TrueRec* |		
	JMove	**JDeo***	**inCode**	**JMove**	**JDeo***	**inCode**
JHotDraw #1	28	39	14	16	10	4
JHotDraw #2	27	39	12	16	10	2
JHotDraw #3	28	38	10	16	8	0
JHotDraw #4	21	41	11	14	12	1
JHotDraw #5	27	39	11	17	12	1
Average	26.20	39.20	11.60	15.80	10.40	1.60
Std Dev	2.95	1.10	1.52	1.10	1.67	1.52
Median	27	39	11	16	10	1
Max	28	41	14	17	12	4
Min	21	38	10	14	8	0

JDeo* stands for **JDeodorant**

other hand, JDeodorant achieved a minimal recall of 27.66% (Cayenne) and a maximal recall of 72.22% (DrJava) while inCode achieved a minimal recall of 0.00% (FreeCol and Maven) and maximal recall of 38.71% (Weka).

Considering $Recall_2$, JMove achieved a recall of 84.11 ± 4.35, JDeodorant achieved a result of 59.58 ± 11.27, and inCode achieved 12.63 ± 10.51. When comparing the results of $Recall_1$ and $Recall_2$, we can observe that $Recall_2$ is 3.74% better than $Recall_1$ in our approach, 9.60% better in JDeodorant, and 2.26% better in inCode, on average. In other words, when the tools detect a method in the wrong class, usually they are also able to infer the correct class to receive the method.

Additional Result: Table 4.6 presents the number of recommendations triggered by

Table 4.5: Recall Results

System	**Recall$_1$ (%)**			**Recall$_2$ (%)**		
	JMove	**JDeo***	**inCode**	**JMove**	**JDeo***	**inCode**
Ant	84.00	72.00	16.00	84.00	84.00	16.00
ArgoUML	84.38	56.25	9.38	84.38	56.25	9.38
Cayenne	72.34	27.66	8.51	78.72	38.30	8.51
DrJava	83.33	72.22	5.56	83.33	72.22	5.56
FreeCol	76.47	41.18	0.00	76.47	58.82	0.00
FreeMind	91.67	58.33	25.00	91.67	58.33	25.00
JMeter	84.00	60.00	16.00	84.00	60.00	20.00
JRuby	70.73	58.54	9.76	85.37	58.54	9.76
JTOpen	89.74	53.85	20.51	89.74	58.97	20.51
Maven	79.17	45.83	0.00	87.50	54,51	0.00
Megamek	80.00	51.43	8.57	80.00	60.00	8.57
WCT	82.76	48.28	6.90	86.21	48.28	6.90
Weka	77.42	64.52	38.71	87.10	74.19	38.71
JHotDraw #1-#5	79.00	51.00	8.0	79.00	52.00	8.0
Average	81.07	54.36	12.35	84.11	59.58	12.63
Std Dev	5.88	11.83	10.35	4.35	11.27	10.51
Median	81.38	55.05	8.97	84.19	58.68	8.97
Max	91.67	72.22	38.71	91.67	84.00	38.71
Min	70.73	27.66	0	76.47	38.30	0

JDeo* stands for **JDeodorant**

Table 4.6: Number of Recommendations

System	**Number of Recommendations**		
	JMove	**JDeodorant**	**inCode**
Ant	21 + 118	21 + 156	4 + 16
ArgoUML	27 + 41	18 + 30	3 + 10
Cayenne	37 + 121	18 + 105	4 + 43
DrJava	15 + 81	13 + 293	1 + 10
FreeCol	13 + 162	10 + 281	0 + 24
FreeMind	11 + 44	7 + 60	3 + 5
JMeter	21 + 50	15 + 102	4 + 31
JRuby	35 + 310	24 + 399	4 + 40
JTOpen	35 + 90	23 + 427	8 + 31
Maven	21 + 32	13 + 56	0 + 15
Megamek	28 + 224	21 + 243	3 + 285
WCT	25 + 31	14 + 72	2 + 46
Weka	27 + 175	23 + 327	12 + 31

JMove, JDeodorant, and inCode for the systems in the Qualitas.*class* Corpus. The results are in the format $tr + r$, where tr are the recommendations that matched a method in the generated gold sets and r are the remaining recommendations, which is not safe to infer whether they represent true recommendations or not. On one hand, inCode produces less recommendations than JMove and JDeodorant in virtually all systems—except the Megamek system. When comparing JMove to JDeodorant, we observe that in 11 out of the 13 systems, JDeodorant produced more recommendations than our approach.

4.1.4 Qualitative Discussion

In this section, we discuss our results in qualitative terms. More specifically, we present three examples of *Move Method* refactorings suggested by our approach for the first modified version of JHotDraw, as described in Section 4.1.2.

Example #1: Code 4.1 shows our first example in which method *calculateLayout*2—moved from class `LocatorLayouter`—does not access any service from its class `AttributeKey`. Thus, its dependency set is very different from the dependency sets of the other methods in this class, as follows:

$$Similarity(calculateLayout2, AttributeKey) = 0.02$$

On the other hand, this method is more similar to the methods in class `LocatorLayouter`. To illustrate, we report the similarity between *calculateLayout*2 and the three methods in `LocatorLayouter`:

$$\begin{array}{lcl} meth\_sim(calculateLayout2, layout) & = & 1.00 \\ meth\_sim(calculateLayout2, calculateLayout) & = & 0.33 \\ meth\_sim(calculateLayout2, getLocator) & = & 0.27 \end{array}$$

As result, we have that:

$$Similarity(calculateLayout2, LocatorLayouter) = 0.53$$

Due to this high similarity, our approach has correctly recommended moving method *calculateLayout*2 back to its original class `LocatorLayouter`.

On the other hand, JDeodorant does not make a recommendation moving this method back. Basically, in the case of `getter` methods, as the `getLocator` call in line 6, JDeodorant considers that the method envies not the target type (`LocatorLayouter`) but the type returned by the call (`Locator`). However, it is not possible to move *calculateLayout*2 to `Locator` because JDeodorant's refactoring preconditions fail in this particular case.

```
public class AttributeKey<T> implements Serializable {
  ... piece of code
  Double calculateLayout2(LocatorLayouter locLayouter,
      CompositeFigure compositeFigure, Double anchor, Double lead) {
      Double bounds = null;
      for (Figure child: compositeFigure.getChildren()){
          Locator loc = locLayouter.getLocator(child);
          Double r;
          if (loc == null) {
              r = child.getBounds();
          } else {
              Double p = loc.locate(compositeFigure);
              Dimension2DDouble d = child.getPreferredSize();
              r = new Double(p.x, p.y, d.width, d.height);
          }
          if (!r.isEmpty()) {
              if (bounds == null) {
                  bounds = r;
              } else {
                  bounds.add(r);
              }
          }
      }
      return (bounds == null) ? new Double() : bounds;
   }
 }
```

Code 4.1: First Move Method Example (JHotDraw)

In order to identify the code smell Feature Envy, inCode uses a strategy described in Section 2.2.4. InCode fails in suggesting to move method *calculateLayout*2 because the metric ATFD (Access to Foreign Data) considers only accesses to external data (e.g., attributes from external classes handled directly or via accessors methods). However, in *calculateLayout*2, only methods from the original class `LocatorLayouter` are called.

Example #2: Code 4.2 shows the second example discussed here. Similar the previous example, method *fireAreaInvalidated*2 does not access any service from class `DrawingEditorProxy`. However, it calls three methods from `AbstractTool` (lines 4-6) and the *Move Method* preconditions are satisfied. For this reason, JDeodorant infers that `AbstractTool` is the most suitable class for the method.

Our approach also suggests to move *fireAreaInvalidated*2 back to `AbstractTool` because the method is more similar to this class than to its current class, as indicated by the following similarity function calls:

```
public class DrawingEditorProxy extends AbstractBean
  implements DrawingEditor {
  ... piece of code
  void fireAreaInvalidated2(AbstractTool abt, Double r){
      Point p1 = abt.getView().drawingToView(...);
      Point p2 = abt.getView().drawingToView(...);
      abt.fireAreaInvalidated(
      new Rectangle(p1.x, p1.y, p2.x - p1.x, p2.y - p1.y));
  }
}
```

Code 4.2: Second Move Method Example (JHotDraw)

$$\begin{aligned} &Similarity(fireAreaInvalidated2,\ DrawingEditorProxy) &= 0.00 \\ &Similarity(fireAreaInvalidated2,\ AbstractTool) &= 0.10 \end{aligned}$$

On the other hand, inCode fails in recommending to move $fireAreaInvalidated2$ because only methods are accessed from its original class `AbstractTool`.

Example #3: Code 4.3 shows an example in which all evaluated approaches successfully recommend moving a method to its original class. As in the previous examples, method *setEditor*— moved from `SVGDrawingPanel`—does not access any service from its class `ViewToolBar`. However, the method accesses 13 different attributes from class `SVGDrawingPanel`.

For JDeodorant a method m envies a class C' when m accesses more services (attributes and methods) from C' than from its own class. Therefore, JDeodorant correctly infers that class `SVGDrawingPanel` is the most suitable class for the method.

InCode also successfully identified *setEditor* as a Feature Envy instance. Particularly, the three rules that compose the detection strategy described in Equation 2.2 are satisfied in this case: (i) the method accesses 13 different attributes from external class`SVGDrawingPanel` and therefore the first rule for the ATFD metric holds; (ii) the method does not access any service from its current class and therefore the second rule for the LAA metric holds; and (iii) only attributes of class `SVGDrawingPanel` are accessed in the body of the method and therefore the third rule for the FDP metric holds.

Our approach also suggests to move *setEditor* back to `SVGDrawingPanel` because the method is more similar to `SVGDrawingPanel` than any other class in the system, including its current class `ViewToolBar`.

Example #4: Code 4.4 shows the last example discussed here where, besides inCode,

```
1:public class SVGDrawingPanel extends JPanel implements Disposable {
    ... piece of code
2:  public void setEditor(SVGDrawingPanel svg, DrawingEditor newValue) {
3:      DrawingEditor oldValue = svg.editor;
4:      if (oldValue != null) {
5:          oldValue.remove(svg.view);
6:      }
7:      svg.editor = newValue;
8:      if (newValue != null) {
9:          newValue.add(svg.view);
10:     }
11:     svg.creationToolBar.setEditor(svg.editor);
12:     svg.fillToolBar.setEditor(svg.editor);
13:     svg.strokeToolBar.setEditor(svg.editor);
14:     svg.actionToolBar.setUndoManager(svg.undoManager);
15:     svg.actionToolBar.setEditor(svg.editor);
16:     svg.alignToolBar.setEditor(svg.editor);
17:     svg.arrangeToolBar.setEditor(svg.editor);
18:     svg.fontToolBar.setEditor(svg.editor);
19:     svg.figureToolBar.setEditor(svg.editor);
20:     svg.linkToolBar.setEditor(svg.editor);
21:     DrawingView temp =
22:             (svg.editor == null) ? null : svg.editor.getActiveView();
23:     if (svg.editor != null) {
24:         svg.editor.setActiveView(svg.view);
25:     }
26:     svg.canvasToolBar.setEditor(svg.editor);
27:     setEditor(svg.editor);
28:     if (svg.editor != null) {
29:         svg.editor.setActiveView(temp);
30:     }
31: }
32:}
```

Code 4.3: Third Move Method Example (JHotDraw)

JMove also fails. Similar the previous examples, method *loadDrawing* does not access any service from its new current class `ProgressIndicator`. However, it calls four methods from its original class `SVGApplet` (lines 3-6) and the *Move Method* preconditions are satisfied. For this reason, JDeodorant correctly infers that `SVGApplet` is the most suitable class for the method. Once again, inCode fails in recommending to move *loadDrawing* because only methods are accessed from its original class `SVGApplet`.

Finally, our approach fails to recommend to move *loadDrawing* back to `SVGApplet` because the method is not structurally similar to the other methods in this class. As can be observed in Code 4.4, due to its specific task of loading a `Drawing`, the set of dependencies of the method contains many particular classes, such as `URLConnection`, `URL`, `IOException`, and `BufferedInputStream`. For this reason, *loadDrawing* has a low similarity to the analyzed classes, as follows:

$$
\begin{aligned}
&\mathit{Similarity}(\mathit{loadDrawing},\ \mathit{ProgressIndicator}) &&= 0.038\\
&\mathit{Similarity}(\mathit{loadDrawing},\ \mathit{SVGApplet}) &&= 0.037
\end{aligned}
$$

In spite of the small difference of similarity between the two classes (only 2.7%), method *loadDrawing* is more similar to its current class(`ProgressIndicator`) and therefore JMove does not provide recommendations for moving the method.

Limitations: Our approach does not provide recommendations for methods that have less than four dependencies and also for methods that are the single methods in their classes (as explained in Section 3.1.2). Regarding the JHotDraw system, we found only 17 classes (2.5%) having a single method. On the other hand, among the 6,533 methods in the system, 4,250 methods (65%) have less than four dependencies. More specifically, 2,173 of such methods (51.1%) are `getters` or `setters`. We also found 1,064 methods (25%) that are graphical user interface listeners or utility methods, such as `toString`, `equals`, etc. By their very nature, such methods are typically implemented in the correct classes. They are also not considered by other *Move Method* recommendation systems, including JDeodorant and the search-based approach proposed by Seng et al. [2006].

Furthermore, we do not recommend to move methods that do not attend the refactoring preconditions. This decision is based on the fact that the moving operation, in this case, typically requires a more complex restructuring both in the source and in the target classes. Finally, we do not provide suggestions to move fields since it is rare to observe fields declared in wrong classes [Tsantalis and Chatzigeorgiou, 2009].

Final Remarks: The distinguishing characteristic of our approach is the fact that we depart from the traditional heuristics for detecting *Move Method* refactorings. Essentially, such heuristics consider that a method m should be moved from C to C' when it accesses more data from C' than from its current class C. Instead, we consider that m should be moved when it is more similar to the methods in C' than to the methods in C. Moreover, we assumed that the dependencies established by a method are good estimators of its *identity*. Therefore, our notion of similarity relies on a similarity coefficient applied over dependency sets, calculated at the level of methods.

Besides checking the traditional heuristic for detecting Feature Envy, JDeodorant only makes a recommendation when the refactoring improves a system-wide metric, which combines cohesion and coupling. Basically, this metric is based on the Jaccard distance between all entities in the system and the original class with the Feature Envy instance. On the other hand, we evaluate the gains of a *Move Method* refactoring by comparing only the original class of the method with the target class. In fact, recent work has questioned whether high-cohesion/low-coupling—when measured at the level of packages—is able to explain the design decisions behind real remodularization

```java
public class ProgressIndicator extends javax.swing.JPanel {
  ... piece of code
public Drawing loadDrawing(SVGApplet svgApplet) throws IOException {
    Drawing drawing = svgApplet.createDrawing();
    if (svgApplet.getParameter("datafile") != null) {
        URL url = new URL(svgApplet.getDocumentBase(),
                            svgApplet.getParameter("datafile"));
        URLConnection uc = url.openConnection();

        if (uc instanceof HttpURLConnection) {
            ((HttpURLConnection) uc).setUseCaches(false);
        }

        int contentLength = uc.getContentLength();
        InputStream in = uc.getInputStream();
        try {
            if (contentLength != -1) {
                in = new BoundedRangeInputStream(in);
                ((BoundedRangeInputStream) in)
                        .setMaximum(contentLength + 1);
                setProgressModel((BoundedRangeModel) in);
                setIndeterminate(false);
            }
            BufferedInputStream bin = new BufferedInputStream(in);
            bin.mark(512);

            IOException formatException = null;
            for (InputFormat format : drawing.getInputFormats()) {
                try {
                    bin.reset();
                } catch (IOException e) {
                    uc = url.openConnection();
                    in = uc.getInputStream();
                    in = new BoundedRangeInputStream(in);
                    ((BoundedRangeInputStream) in)
                            .setMaximum(contentLength + 1);
                    setProgressModel((BoundedRangeModel) in);
                    bin = new BufferedInputStream(in);
                    bin.mark(512);
                }
                try {
                    bin.reset();
                    format.read(bin, drawing, true);
                    formatException = null;
                    break;
                } catch (IOException e) {
                    formatException = e;
                }
            }
            if (formatException != null) {
                throw formatException;
            }
        } finally {
            in.close();
        }
    }
    return drawing;
  }
}
```

Code 4.4: Fourth Move Method Example (JHotDraw) tasks [Anquetil and Laval, 2011].

Since inCode considers only accesses to attributes (not methods), the tool fails in many cases. Therefore, a potential improvement to inCode would be changing the AFTD metric to consider all dependencies defined by a given method (similar to JDeodorant and our strategy). On the other hand, this improvement may lead to an explosion in the number of false positives, and therefore new preconditions (e.g., *Move Method* ones) or new strategies (e.g., JDeodorant's Entity Placement metric) need to be considered.

4.1.5 Threats to Validity

Quite similar to the study reported in Section 3.2, there are three main threats regarding the validity of this study. First, as usual, we cannot extrapolate our results to other systems (external validity). However, we argue that we evaluated a credible sample including 14 real-world systems. Second, we acknowledge that the strategy we follow to generate the gold sets can lead to incorrect classifications in rare circumstances, as already discussed in Section 4.1.2 (conclusion validity). However, despite having a low probability of being synthesized, invalid entries in our gold sets would affect equally our results and the results generated by JDeodorant and inCode, which at least makes our comparison of the tools fair. Third, we cannot claim that our approach outperforms JDeodorant and inCode in a real scenario since our evaluation is based on artificial moves (conclusion validity). However, in order to address this threat, we perform a field study involving real systems and assertions of experts, described in the next section.

4.2 Evaluation with Experts

This section is divided into five parts, Section 4.2.1 states the research questions we aim to answer. Section 4.2.2 presents the evaluated systems and our methodology. Sections 4.2.4 and 4.2.3 report and discuss the results achieved in the two systems evaluated in the section. Finally, Section 4.2.5 lists threats to validity.

4.2.1 Research Questions

This second study aims to answer the following research questions:

- ***RQ #1***– How does our approach compare with JDeodorant and inCode in real instances of methods implemented in incorrect classes from industrial-strength systems?
- ***RQ #2***– How relevant are our recommendations from a developers' point of view?

4.2.2 Study Design

In this section, we present the evaluated systems and the methodology we followed in this second study.

Target systems: This study relies on two real-world systems: (i) Geplanes, an open-source strategic management Java web-based system designed to handle strategic management activities, including management plans, goals, performance indicators, actions, etc.; and (ii) SGA,[2] an EJB-based information system used by a major Brazilian university, which includes functionalities for human resource management, finance, accounting, and material management. Table 4.7 presents the main characteristics of both systems, in terms of their sizes.

Table 4.7: Real-world systems evaluated by experts

System	**NOC**	**NOM**	**LOC**
Geplanes	340	3,101	29,046
SGA	1,056	11,556	27,045

We chose these systems for two main reasons: (i) both systems are mature projects, facing a continuous process of maintenance due to constant requirement changes. Therefrom, it is reasonable to expect that they offer several refactoring opportunities; and (ii) we have accesses to an expert in each system, who could dedicate a significant amount of time to studying and commenting the refactoring suggestions provided by the considered approaches.

Methodology: We executed the JMove, JDeodorant, and inCode on the SGA and Geplanes systems. As already detailed in Section 4.1.2, each execution generated a list of recommendations, whose elements are triples (m, C, C') expressing a suggestion to

[2]Due to confidentiality reasons, we omit the real name of this system.

move m from C to C'. On the other hand, a suggestion from inCode is a triple (m, C, T), where T is a list of classes.

After generating the list of recommendations provided by the three approaches, we invited the expert to evaluate the recommendations using the Likert scale [Jamieson, 2004]. Specifically, we asked the expert the following question: *How do you rank this Move Method recommendation?* The answers could be: (1) Strongly not recommended, (2) Not recommended, (3) Neither recommended nor not recommended, (4) Recommended, (5) Strongly Recommended.

4.2.3 SGA Results

We executed the three refactoring recommendation tools (JMove, JDeodorant, and inCode) in the SGA system. JMove could not trigger any recommendation, while JDeodorant and inCode triggered 43 and 50 recommendations, respectively. More important, the expert classified every recommendation triggered by JDeodorant and inCode as *strongly not recommended*.

In order to explain the reasons behind too many incorrect recommendations, Figure 4.1 presents a high-level component model of the SGA's architecture, as defined by its expert developer. As can be observed, SGA follows a layered architectural pattern where the *ManagedBean* layer is the bridge between Graphical User Interface (GUI) components and business-related components. The *Service* layer is the core business processes, *Persistence Layer* is responsible for database operations, and the *BusinessEntity* layer implements system domain types (e.g., `Professor`, `Student`, etc.).

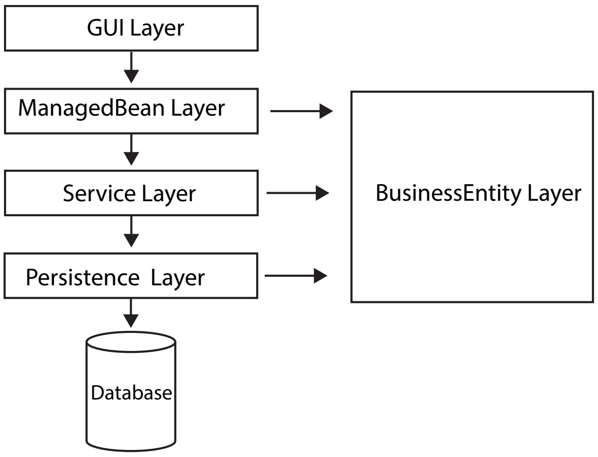


Figure 4.1: SGA's architecture

According to SGA expert, the *BusinessEntity* layer should contain only entity classes, which have only fields and accessor methods (i.e., *getters* and *setters*). Nevertheless, all recommendations given by JDeodorant and inCode suggest to move methods to such entity classes. As an example, we present a *strongly not recommended* suggestion provided by JDeodorant. In this particular case, JDeodorant recommended to move method `find` (Code 4.5) to class `Outsourcing` (Code 4.6).[3]

```java
public class OutsourcingDAO implements IOutsourcingDAO {
    ...piece of code
    public List<Outsourcing> find(Outsourcing service) {
        Query query = this.entityManager.createNamedQuery(
            OutsourcingDAO.FIND_OUTSOURCING);
        return (List<Outsourcing>) query.setParameter("id",
                service.getId()).getResultList();
    }
}
```

Code 4.5: Example to illustrate a typical recommendation on SGA

```java
public class Outsourcing extends AuditInfo implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id = 0;
    ...remainder attributes

    public long getId() {
        return id;
    }
    public void setId(long idOutsourcing) {
        this.id = idOutsourcing;
    }
    ...remainder accessor methods
}
```

Code 4.6: Class *Outsourcing*

The `find` method is currently implemented in class `OutsourcingDAO` located on the *Persistence* layer, which is the unique layer that can access the database according to the SGA's architecture. Therefore, the expert classified such *Move Method* refactoring recommendation as *strongly not recommended* for the following two reasons: (i) `Outsourcing` is an entity class, which must contain only fields and accessor methods; and (ii) classes in the *BusinessEntity* layer cannot access the database, wich represents an architectural violation.

In fact, JDeodorant and inCode triggered too many wrong recommendations because they rely on the number of accessed members and, by their nature, entity classes are widely accessed. On the other side, JMove is more robust to such false

[3]The complete source code of class *OutsourcingDAO* is available in Appendix B.1

positives because it does not rely on the number of members accesses of each class, but on the set of static dependencies established by the method. In other words, an access to a class is transformed into a single dependency regardless of the number of times it happens. Therefore, a single dependency is not able to attract a method to an entity class, because JMove does not make recommendations for methods whose dependency set has less than four dependencies.

In summary, this study provides empirical evidence that—in systems with a strict and well-defined architecture—it is quite rare that developers implement methods in inappropriate classes. In fact, none of approaches was able to identify true *Move Method* refactoring opportunities.

4.2.4 Geplanes Results

JMove triggered 72 recommendations of *Move Method* refactorings for the Geplanes system. In a first analysis, we found that 31 out of 72 recommendations are in accessor methods, which by their very nature are rarely implemented in incorrect locations. As mentioned in Section 3.1.2, to exclude accessors methods, we do not recommend *Move Method* refactorings for methods that have less than four dependencies. However, Geplanes makes massive use of Java annotations. For instance, Code 4.7 illustrates a typical *getter* method from a persistence class in Geplanes.

```
1:  @Required
2:  @DisplayName("Data da auditoria")
3:  public Date getDataAuditoria() {
4:      return this.dataAuditoria;
5:  }
```

Code 4.7: Typical getter method in Geplanes

The method `getDataAuditoria` has four dependencies: a dependency with the return type `Date` (line 3); a dependency with its owner class `AuditoriaGestao` (line 4), and two dependencies due to the annotations `Required` (line 1) and `DisplayName` (line 2). Due to the frequent use of annotations in accessors methods, our filter failed on excluding these *getters* methods in Geplanes.

To address this shortcoming, we decide to manually remove from the analysis methods that have no parameters and have only one statement—which is a *return* statement. This solution can be automated and it is based on the one proposed by Seng et al. [2006] to identify methods that should not be moved. Using this filter, 41 JMove's recommendations were given to the expert. Table 4.8 presents the number of recommendations triggered by JMove, and also by JDeodorant and inCode on Geplanes system.

Table 4.8: Evaluation of the Geplanes' recommendations by an expert

How do you rank this Move Method recommendation?	**JMove**	**JDeodorant**	**inCode**
(5) Strongly recommended	7 (17%)	6 (3%)	1 (3%)
(4) Recommended	3 (7%)	6 (3%)	1 (3%)
(3) Neither recommended nor not recommended	2 (5%)	18 (9%)	0 (0%)
(2) Not recommended	9 (22%)	28 (15%)	5(13%)
(1) Strongly not recommended	20 (49%)	136 (70%)	31(81%)
Total	**41**	**194**	**38**

Our approach could trigger three recommendations scored as *recommended* and seven as *strongly recommended.* Stated differently, 10 out of 41 recommendations triggered by JMove would improve the system quality according to the expert judgment.

In a detailed analysis for the recommendations triggered by JMove we observed that 10 out of 20 recommendations scored by the expert as *strongly not recommended* are methods having a parameter `WebRequestContext`, which should never be moved as explained by the architect because this refactoring violates the system architecture.

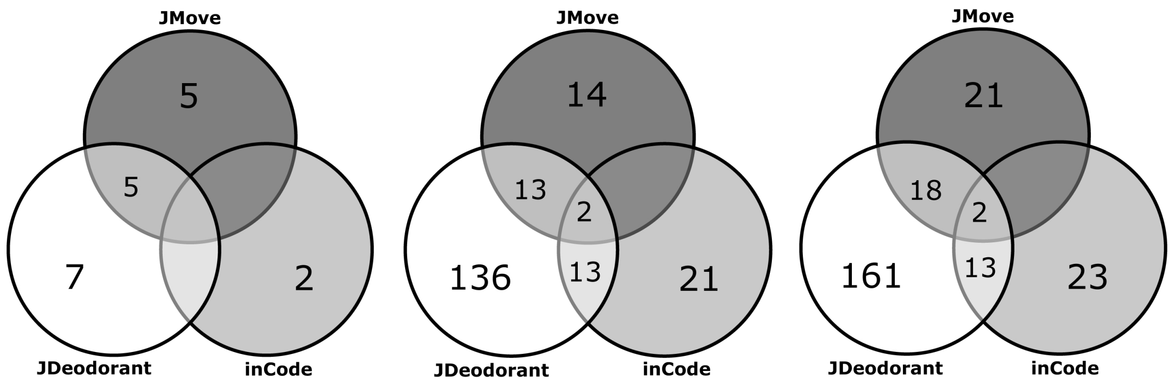
Moreover, for the ten remaining *strongly not recommended* JMove's suggestions, the current and the recommended classes are structurally very similar to the method. For example, 5 out of 10 recommendations suggest to move methods between DAO (Data Access Object) classes, which by their nature are very similar. More specifically, these five recommendations indicate to move methods from class `AnomaliaDAO` to class `UnidadeGerencialDAO`. Although these methods are structurally similar to class `UnidadeGerencialDAO`, semantically they should not be moved. Similarly, the five remaining recommendations occurred between methods of *Service* layer classes, which are structurally very similar.

For the *not recommended* suggestions triggered by JMove, we observed that 7 out of 9 recommendations also refer to the case in which the current and the recommended classes are mostly structurally very similar to the method. Regarding the suggestions classified as *neither recommended nor not recommended*, the expert argued that the recommendation could be applied, but it would not significantly improve the system quality.

When comparing JMove with JDeodorant and inCode, we observed that JDeodorant was able to detect more correct suggestions than JMove and inCode in absolute values. As reported in Table 4.8, JDeodorant could detect 12 *strongly recommended* or *recommended* refactorings, whereas JMove and inCode could detect 10 and 2, respectively. However, JDeodorant triggered 194 recommendations, which is 373% and 410% more recommendations than JMove and inCode, respectively. As a result, 164 recommendations (85%) triggered by JDeodorant are *not recommended* or *strongly not recommended*. Furthermore, it is possible to observe that similarly to the evaluation with open-source systems (Section 4.1), inCode produced the lowest number of 38 recommendations, but achieved the worst precision of 5.2%, i.e., 36 out of 38 recommendations triggered by inCode are *not recommended* or *strongly not recommended*.

Intersection of the results: Finally, we checked the intersection between the recommendations triggered by JMove, JDeodorant, and inCode which is presented in Figure 4.2. Regarding the correct recommendations (scores 4 and 5), we can observe that there is a relevant intersection between JDeodorant and JMove (half of the correct recommendations provided by JMove are also provided by JDeodorant). On the other side, we found no intersection between JMove and inCode and between JDedorant and inCode. Regarding the incorrect recommendations (scores 1 and 2) the results basically reinforce our previous observation on the massive number of false positives provided only by JDeodorant (136 recommendations) and, to some extent, by inCode only (21 recommendations). Essentially, this finding undermines the possibility of running the three tools and combining their results. In fact, this scenario would only be feasible after investigating a heuristic to reduce the false positives provided by JDeodorant, without affecting the true positives detected by such tool.

Figure 4.2: Intersection between the JMove, JDeodorant, and inCode



(a) Correct recommnedations (b) Wrong recommnedations (c) All recommnedations

Lessons Learned: Two main lessons were learned after this study:

- The results indicate that a more sophisticated mechanism is necessary to decide when recommendations involving two very similar classes should be triggered. Currently, as explained in Section 3.1.3, a move recommendation is *not* created when: (i) T has less than three classes, and (ii) the difference between the similarity coefficient value of C_1 (the first class in the list) and C (the original class of m) is less than or equal to 25%. A potential improvement to address this shortcoming is to introduce a mechanism to measure the amount of uncertainty in each *Move Method* recommendation. This solution is similar to the one recently proposed by Bavota et al. [2014] who complements *Move Method* refactoring recommendations with a *confidence level* that indicates the reliability of the proposed refactoring.

- The recommendations should be filtered by the system's architect, possibly using an automatic filtering based on straightforward regular expressions, to remove recommendations that represent violations in design rules. For example, the Geplanes' architect could hide all recommendations involving methods from the *Controller* layer having a `WebRequestContext` parameter. By following this filter, we could achieved a precision of 38.5% in Geplanes.

4.2.5 Threats to Validity

The subject systems Geplanes and SGA are Java EE web applications, thus our results might be in part due to the specific nature of these systems. As usual, we cannot extrapolate our results to others types of systems (external validity). The experts could be responsible for some bad design choices and consequently they could not have recognized a correct *Move Method* refactoring opportunity as meaningful. However, we conducted a deep discussion with them on the results. Moreover, the feedback we received about the correct recommendations demonstrated that the experts provided an objective evaluation of the *Move Method* operations.

4.3 Performance Analysis

We executed JMove, JDeodorant, and inCode in the original version of JHotDraw (described in Section 3.2.1) to measure the tool's runtime performance. The system configuration used was an Intel Core2 Duo CPU E8400 @3.00GHz with 16GB RAM, operating system LinuxMint 14 (Nadia), and Java SE Runtime Environment 1.6.0_45.

Table 4.9 reports the time needed by the approaches to provide their recommendations. JMove required 17:56 minutes to provide *Move Method* recommendations on JHotDraw, while JDeodorant required 2:49, and inCode 1:27 minutes. InCode required less time to provide the recommendations, but it is important to emphasize that this tool does not check the *Move Method* refactoring preconditions and therefore some recommendations could not have a trivial application.

Table 4.9: Execution time for JHotDraw (minutes)

JMove	**JDeodorant**	**inCode**
17:56	2:49	1:27

To understand our low performance, it is necessary to present some details of JMove's current implementation. JMove is implemented as an Eclipse plug-in and relies on the Eclipse JDT parser to build the Abstract Syntax Tree (AST). Furthermore, JMove relies on preconditions checked by the *Move Method* automatic refactoring engine provided by the Eclipse IDE. More specifically, the process we follow to identify *Move Method* refactoring opportunities is as follows.

1. Parsing the system into an Abstract Syntax Tree (AST).
2. Computing the set of dependencies (for every single method).
3. Measuring the similarity between each method/class pair of the system.
4. Verifying *Move Method* refactoring preconditions.

As observed, JMove creates dependency sets for every method and also measures the similarity between these methods and all classes in the system. To save computational resources, JMove only calculates the similarity between a given method m and a class C when the operation satisfies the *Move Method* refactoring preconditions. However, JMove depends of an external API to check the *Move Method* refactoring preconditions, which is not very efficient. Basically, this checking procedure demanded 13:41 minutes, i.e., 76.4% of the total executing time.

In summary, since our tool is a prototype, it requires further performance improvements. For instance, we could implement the checking of the *Move Method* refactoring preconditions in the same way as JDeodorant, rather than using an external API. However, it is important to acknowledge that, although our prototype needs performance improvements, it is robust enough to be used in real-world systems.

4.4 Final Remarks

This chapter reported the evaluation of our approach. In Section 4.1, we manually synthesized versions of 14 open-source systems with well-known methods implemented in incorrect classes. We achieved an average precision of 60.63% and an average recall of 81.07%. On average, our results were 49.13% better than JDeodorant and 556.43% better than inCode in terms of recall, and 129% better than JDeodorant and 378% better than inCode in terms of precision.

In Section 4.2, we used two real-world systems in which the systems' experts evaluated the recommendations triggered by the approaches. Regarding the SGA study, we learned that in systems with a well-defined architecture it is quite rare that developers implement methods in inappropriate classes. Regarding the Geplanes study, we conclude that some improvements can be made on JMove to make it more robust, including a filter for accessor methods (especially, *getters*) and a more sophisticated mechanism to decide when recommendations between two very similar classes should be triggered. In spite of these findings, JMove was more precise than the other approaches in both evaluated systems. In the Geplanes system, for instance, JMove achieved a precision of 25.6% against 6.8% achieved by JDeodorant and 5.2% by inCode.

Chapter 5

Conclusion

We organized this chapter as follows. First, Section 5.1 provides a brief description of the problem and the approach we proposed. Second, Section 5.2 reviews the contributions of our research. Next, Section 5.3 points the limitations of our work. Finally, Section 5.4 suggests further work.

5.1 Summary

During software evolution, developers may inadvertently implement methods in incorrect classes, creating instances of the Feature Envy code smell. Basically, the actions to remove a Feature Envy are well-documented. A *Move Method* refactoring must be applied to move the method from its current class to the class that it envies [Fowler, 1999; Tsantalis and Chatzigeorgiou, 2009]. However, maintainers must first detect the Feature Envy instances in the source code, and determine the correct classes to receive the methods, which are two non-trivial program comprehension tasks. Despite that, currently there are few tools intended to assist in identifying opportunities to take advantage of *Move Method* refactorings.

To address this shortcoming, we proposed in this master's dissertation a solution based on recommendation system principles to recommend *Move Method* refactorings. Basically, our recommendation approach detects methods located in incorrect classes and then suggests moving them to more suitable classes. Moreover, we conducted an evaluation where experts in two industrial-strength systems provided us encouraging feedback on the applicability of our recommendations. We also conducted an evaluation where we manually synthesized versions of 14 open-source systems with well-known methods implemented in incorrect classes. In this case, we achieved an average precision of 60.63% and an average recall of 81.07%. On average, our results are 49.13% better than JDeodorant and 556.43% better than inCode in terms of recall, and 129% better than JDeodorant and 378% better than inCode in terms of precision.

5.2 Contributions

This research makes the following contributions:

- The design of an algorithm that provides *Move Method* recommendations for developers and maintainers to remove the Feature Envy code smell and therefore to improve the system quality (Chapter 3);
- An empirical study that supported the implementation decisions (coefficients and strategies) related to our heuristic to calculate the similarity between methods and classes (Section 3.2);
- A prototype tool called JMove that implements our approach and hence detects methods located in incorrect classes and then suggests moving them to more suitable classes. Furthermore, JMove applies the *Move Method* when requested. JMove is publicly available at: http://aserg.labsoft.dcc.ufmg.br/jmove/ (Section 3.3);
- An evaluation of the correctness of our recommendations using two industrial-strength systems and synthesized versions of 14 open-source systems (Chapter 4);
- A dataset with 475 well-defined Feature Envy instances distributed among 14 open-source systems, including 100 methods in five JHotDraw instances.This dataset is publicly available at JMove's website (Chapter 4.1.2).

5.3 Limitations

Our work has the following limitations:

- Our heuristic is based only on structural similarity, even though the way that developers decide a most suitable class for a method might also consider semantic and architectural aspects;
- We do *not* handle static methods;
- We do *not* provide recommendations about method that do *not* satisfy the *Move Method* preconditions, i.e., methods that can not be automatically moved;

- Our prototype tool has some performance issues that can be tackled by implementing its own precondition checking functions instead of relying on the Eclipse functions.

- We did *not* evaluate with experts whether our approach provides equivalent results in contexts other than web-based systems;

5.4 Future Work

We consider that this work can be complemented with the following future work:

- *Proposed Approach*: (i) a new mechanism to decide when recommendations between similar classes should be triggered. A potential improvement to address this problem is to investigate a mechanism to measure the uncertainty inherent to each *Move Method* recommendation, similar to the one proposed by Bavota et al. [2014]. Basically, in this work they supplement a suggestion of envied class with a *confidence level* that indicates the reliability of the proposed refactoring; (ii) extend our approach to search opportunities for *Move Method* refactorings in methods' fragments, to extract fragments into a new method and then to recommend the move; (iii) conduct an evaluation of our approach using experts in non web-based systems; and (iv) compare our approach with MethodBook, an approach that considers structural and semantic aspects to provide *Move Method* recommendations [Bavota et al., 2014].

- *JMove* Tool: (i) the implementation of a module to check the *Move Method* refactoring preconditions; and (ii) work on performance improvements to reach for example a performance similar to the one achieved by JDeodorant.

Bibliography

Anquetil, N. and Laval, J. (2011). Legacy software restructuring: Analyzing a concrete case. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–286.

Anquetil, N. and Lethbridge, T. (1999). Experiments with clustering as a software remodularization method. In *6th Working Conference on Reverse Engineering (WCRE)*, pages 235–255.

Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and Lucia, A. D. (2014). Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 99:1–10.

Bieman, J. M. and Kang, B.-K. (1995). Cohesion and reuse in an object-oriented system. *Software Engineering Notes*, 20:259–262.

Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52:53–100.

Briand, L. C., Daly, J. W., and Wüst, J. (1998). A unified framework for cohesion measurement in object-orientedsystems. *Empirical Software Engineering*, 3(1):65–117.

Casais, E. (1994). Automatic reorganization of object-oriented hierarchies: a case study. *Object Oriented Systems*, 1:95–115.

Chang, J. and M. Blei, D. (2010). Hierarchical relational models for document networks. In *11th Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 124–150.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

D'Ambros, M., Bacchelli, A., and Lanza, M. (2010). On the impact of design flaws on software defects. In *10th International Conference on Quality Software (QSIC)*, pages 23–31.

Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, pages 53–95.

Everitt, B. S., Landau, S., Leese, M., and Stahl, D. (2011). *Cluster Analysis*. Wiley, 5th edition.

Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. (2012). Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260.

Fowler, M. (1999). *Refactoring: Improving the design of existing code*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

Glorie, M., Zaidman, A., van Deursen, A., and Hofland, L. (2009). Splitting a large software repository for easing future software evolution - an industrial experience report. *Journal of Software Maintenance*, 21(2):113–141.

Holmes, R., Walker, R. J., and Murphy, G. C. (2006). Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970.

Jamieson, S. (2004). Likert scales: how to (ab)use them. *Medical Education*, 38(12):1217–1218.

Kataoka, Y., Notkin, D., Ernst, M. D., and Griswold, W. G. (2001). Automated support for program refactoring using invariants. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 736--744.

Lanza, M., Marinescu, R., and Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Springer.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

Maffort, C., Valente, M. T., Anquetil, N., Hora, A., and Bigonha, M. (2013). Heuristics for discovering architectural violations. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 222–231.

Marinescu, C., Marinescu, R., Mihancea, P. F., and Wettel, R. (2005). iPlasma: An integrated platform for quality assessment of object-oriented design. In *International Conference on Software Maintenance (ICSM) (Industrial and Tool Volume)*, pages 77–80.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSM)*, pages 350–359.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Mihancea, P. F. and Marinescu, R. (2005). Towards the optimization of automatic detection of design flaws in object-oriented software systems. In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 92–101.

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.

Moghadam, I. H. and Ó Cinnéide, M. (2011). Code-imp: a tool for automated search-based refactoring. In *4th Workshop on Refactoring Tools (WRT)*, pages 41–44.

Moghadam, I. H. and Ó Cinnéide, M. (2012). Automated refactoring using design differencing. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52.

Montandon, J. E., Borges, H., Felix, D., and Valente, M. T. (2013). Documenting apis with examples: Lessons learned with the apiminer platform. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 401–408.

Moore, I. (1996). Automatic inheritance hierarchy restructuring and method refactoring. *SIGPLAN Not.*, 31(10):235--250.

Oliveira, P., Valente, M. T., and Lima, F. (2014). Extracting relative thresholds for source code metrics. In *1st CSMR-WCRE Software Evolution Week*, pages 1–10.

Oliveto, R., Gethers, M., Bavota, G., Poshyvanyk, D., and De Lucia, A. (2011). Identifying method friendships to remove the feature envy bad smell. In *33rd International Conference on Software Engineering (ICSE), NIER track*, pages 820–823.

Opdyke, W. (1992). *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

Riehle, D. (2000). *Framework Design: A Role Modeling Approach*. PhD thesis, Swiss Federal Institute of Technology.

Riel, A. J. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley.

Robillard, M., Walker, R., and Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Software*, 27:80–86.

Santos, G., Valente, M. T., and Anquetil, N. (2014). Remodularization analysis using semantic clustering. In *1st CSMR-WCRE Software Evolution Week*, pages 1–10.

Schäefer, M. and de Moor, O. (2010). Specifying and implementing refactorings. In *25th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 286–301.

Schäfer, M., Ekman, T., and de Moor, O. (2009). Challenge proposal: verification of refactorings. In *3rd Workshop on Programming Languages meets Program Verification (PLPV)*, pages 67–72.

Seng, O., Stammel, J., and Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In *8th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1909–1916.

Sjoberg, D. I., Yamashita, A., Anda, B., Mockus, A., and Dyba, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, pages 1–15.

Soares, G., Gheyi, R., and Massoni, T. (2013). Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162.

Soares, G., Gheyi, R., Serey, D., and Massoni, T. (2010). Making program refactoring safer. *IEEE Software*, 27(4):52–57.

Sokal, R. R. and Sneath, P. H. A. (1963). *Principles of Numerical Taxonomy*. Freeman.

Sokal, R. R. and Sneath, P. H. A. (1973). *Numerical Taxonomy: The Principles and Practice of Numerical Classification*. Freeman.

Steimann, F. and Thies, A. (2009). From public to private to absent: Refactoring Java programs under constrained accessibility. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The Qualitas Corpus: A curated collection of Java code for empirical studies. In *17th Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345.

Terra, R., ao Brunet, J., Miranda, L., Valente, M. T., Serey, D., Castilho, D., and Bigonha, R. (2013a). Measuring the structural similarity between source code entities. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 753–758.

Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. (2013b). Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, pages 1–4.

Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software - Practice and Experience*, 32(12):1073–1094.

Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. (2013c). A recommendation system for repairing violations detected by static architecture conformance checking. *Software - Practice and Experience*, pages 1–36.

Tokuda, L. and Batory, D. (2001). Evolving object-oriented designs with refactorings. *16th Automated Software Engineering (ASE)*, 8:89–120.

Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of Move Method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367.

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

Verbaere, M., Ettinger, R., and de Moor, O. (2006). JunGL: a scripting language for refactoring. In *28th International Conference on Software Engineering (ICSE)*, pages 172–181.

Xing, Z. and Stroulia, E. (2008). The JDEvAn tool suite in support of object-oriented evolutionary development. In *30th International Conference on Software Engineering (ICSE)*, pages 951–952.

Yamashita, A. and Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *28th International Conference on Software Maintenance (ICSM)*, pages 306–315.

Ye, Y. and Fischer, G. (2005). Reuse-conducive development environments. *20th Automated Software Engineering (ASE)*, 12(2):199–235.

Appendix A

Running Example Classes

In this appendix, we include the complete source code of the following classes used in the running example presented in Section 3.1.4: `CustomerView` (Section A.1) and `CustomerDAO` (Section A.2).

A.1 Class CustomerView

```
package com.foo.view;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JTextPane;
import com.foo.dao.CustomerDAO;
import com.foo.database.DB;
import com.foo.domain.Customer;

public class CustomerView extends JFrame {

  private JPanel contentPane;
  private JTextField textField_Name;
  private JTextField textField_SocialNumber;
```

```java
private Customer customer;
private CustomerDAO city;

public List<Customer> getAllCustomers() throws SQLException {
  List<Customer> result = new ArrayList<Customer>();
  Connection conn = DB.getConnection();
  PreparedStatement ps =
    conn.prepareStatement("select * from CUSTOMER");
  ResultSet rs = ps.executeQuery();

  while (rs.next()) {
    result.add(new Customer(rs.getInt("ID"),rs.getString("NAME")));
  }
  rs.close();
  ps.close();
  conn.close();
  return result;
}

public CustomerView() {
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setBounds(100, 100, 450, 300);

  JMenuBar menuBar = new JMenuBar();
  setJMenuBar(menuBar);

  JMenu mnNewMenu = new JMenu("File");
  menuBar.add(mnNewMenu);

  JMenuItem item1 = new JMenuItem("Open");
  mnNewMenu.add(item1);

  contentPane = new JPanel();
  setContentPane(contentPane);
  contentPane.setLayout(null);

  JTextPane txtpnName = new JTextPane();
  txtpnName.setText("Name:");
  txtpnName.setBounds(25, 151, 68, 20);
  contentPane.add(txtpnName);

  textField_SocialNumber = new JTextField();
  textField_SocialNumber.setColumns(10);
  textField_SocialNumber.setBounds(139, 11, 193, 20);
  contentPane.add(textField_SocialNumber);

  textField_Name = new JTextField();
  textField_Name.setBounds(105, 151, 193, 20);
  contentPane.add(textField_Name);

  textField_Name.setColumns(10);
  JTextPane txtpnCpf = new JTextPane();
```

```
   txtpnCpf.setText("SocialNumber");
   txtpnCpf.setBounds(25, 11, 98, 20);
   contentPane.add(txtpnCpf);

   JMenuItem mntmNewMenuItem = new JMenuItem("Save");
   mnNewMenu.add(mntmNewMenuItem);

   JButton btnNewButton = new JButton("Find");
   btnNewButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         customer = new Customer(textField_SocialNumber.getText());
         textField_Name.setText(customer.getName());
         }
     });

     btnNewButton.setBounds(157, 57, 89, 23);
     contentPane.add(btnNewButton);
 }
}
```

Code A.1: Class *CustomerView*

A.2 Class CustomerDAO

```
package com.foo.dao;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import com.foo.database.DB;
import com.foo.domain.Customer;
import com.foo.domain.Ticket;

public final class CustomerDAO {

  public List<Customer> getCities(DB db) throws SQLException {
     String sql = "select * from CITY";
     List<Customer> listaDeCidades = new ArrayList<Customer>();
     Customer tmp = null;
     PreparedStatement pstmt;
     pstmt = db.getConnection().prepareStatement(sql);

     ResultSet rs = pstmt.executeQuery();

     while (rs.next()) {
         tmp = new Customer();
         tmp.setCityId(rs.getString("CityID"));
         tmp.setCityName(rs.getString("CityName"));
```

```
       listaDeCidades.add(tmp);
    }

    return listaDeCidades;
}

public String getCityOfOrigin(DB db, Ticket ticket)
throws SQLException {
   String instanceStatement;
   PreparedStatement pmts = db.getConnection().
       prepareStatement("select min(IDINSTANCE) as instance from"
       +" ADM.TICKETINSTANCE where idTicket = ?");
   pmts.setString(1, ticket.getIdPassagem());
   ResultSet rs = pmts.executeQuery();

   if (rs.next()) {
      instanceStatement = rs.getString("instance");
   } else {
      return null;
   }

   pmts = db.getConnection().
       prepareStatement("select CITYNAME as name from ADM.INSTANCE "
       + "natural join PATH natural join CITY"
       + "where IDCITYOFORIGIN =IDCITY and IDINSTANCE =?");
   pmts.setString(1, instanceStatement);
   rs = pmts.executeQuery();

   if (rs.next()) {
       return rs.getString("name");
   } else {
      return null;
   }
}

public String getDestinyCity(DB db, Ticket ticket)
throws SQLException {

   String instanceStatement;
   PreparedStatement pmts = db.getConnection().
       prepareStatement("select max(IDINSTANCE) as instance"
       +" from ADM.TICKETINSTANCE where idTicket = ?");
   pmts.setString(1, ticket.getIdPassagem());
   ResultSet rs = pmts.executeQuery();
   if (rs.next()) {
       instanceStatement = rs.getString("instance");
   } else {
       return null;
   }

   pmts = db.getConnection().
       prepareStatement("select CITYNAME as name from ADM.INSTANCE"
       + "natural join PATH natural join CITIES"
```

```
            + "where IDDESTINYCITY =IDCITY and IDINSTANCE=?");
        pmts.setString(1, instanceStatement);
        rs = pmts.executeQuery();

        if (rs.next()) {
            return rs.getString("name");
        } else {
            return null;
        }
    }
}
```

Code A.2: Class *CustomerView*

Appendix B

SGA running Example Classes

In this appendix, we include the complete source code of the class `Outsourcing` used in the SGA example presented in Section 4.2.3.

B.1 Class Outsourcing

```java
package *.ext.nucleo.dominio;

import java.io.Serializable;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;

import *.nucleo.dominio.AuditoriaInfo;

@Entity
@NamedQueries({...})

public class Outsourcing extends AuditInfo
 implements Serializable{

 private static final long serialVersionUID = 1L;

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private long id = 0;
 private String activity = "";
```

```
@ManyToOne(cascade = CascadeType.PERSIST)
@JoinColumn(name = "id_projectActivity")
private ProjectActivity projectActivity;

private String terceiro = "";
private boolean pessoaFisica = false;
private boolean pessoaJuridica = false;
private long quantity = 0;
private long unitary = 0;
private double unitaryValue = 0.0;

@ManyToOne(cascade = { CascadeType.PERSIST })
@JoinColumn(name = "id_engagedResource",
    referencedColumnName = "ID", nullable = true)
private EngagedResource engagedResource;

public Outsourcing() {
    engagedResource = new EngagedResource();
}

public long getId() {
    return id;
}

public void setId(long idOutsourcing) {
    this.id = idOutsourcing;
}

public void setActivity(String activity) {
    this.activity = activity;
}

public String getActivity() {
    return activity;
}

public void setTerceiro(String terceiro) {
    this.terceiro = terceiro;
}

public String getTerceiro() {
    return terceiro;
}

public void setQuantity(long quantity) {
    this.quantity = quantity;
}

public long getQuantity() {
    return quantity;
}
```

```java
public void setUnitary(long unitary) {
    this.unitary = unitary;
}

public long getUnitary() {
    return unitary;
}

public void setUnitaryValue(double unitaryValue) {
    this.unitaryValue = unitaryValue;
}

public double getUnitaryValue() {
    return unitaryValue;
}

public void setEngagedResource(EngagedResource engagedResource) {
    this.engagedResource = engagedResource;
}

public EngagedResource getEngagedResource() {
    return engagedResource;
}

public void setPessoaFisica(boolean pessoaFisica) {
    this.pessoaFisica = pessoaFisica;
}

public boolean isPessoaFisica() {
    return pessoaFisica;
}

public void setPessoaJuridica(boolean pessoaJuridica) {
    this.pessoaJuridica = pessoaJuridica;
}

public boolean isPessoaJuridica() {
    return pessoaJuridica;
}

public boolean equals(Object obj) {
    if (!(obj instanceof Outsourcing)) {
        return false;
    }
    return this.getActivity().equals(
            ((Outsourcing) obj).getActivity());
}

public void setProjectActivity(ProjectActivity projectActivity) {
    this.projectActivity = projectActivity;
}

public ProjectActivity getProjectActivity() {
    return projectActivity;
}
}
```

Code B.1: Example to illustrate a typical recommendation on SGA