RECOMMENDING AUTOMATED EXTRACT METHOD REFACTORING

DANILO FERREIRA E SILVA

RECOMMENDING AUTOMATED EXTRACT METHOD REFACTORING

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

Julho de 2014

DANILO FERREIRA E SILVA

RECOMMENDING AUTOMATED EXTRACT METHOD REFACTORING

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

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

Belo Horizonte

July 2014

Silva, Danilo Ferreira e

S586r Recommending automated extract method refactoring / Danilo Ferreira e Silva. — Belo Horizonte, 2014
xxi, 68 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. Engenharia de software - Teses. 3. Sistemas de recomendação - Teses.
I. Orientador. II. Coorientador. III. Título.

CDU 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 automated extract method refactorings

DANILO FERREIRA E SILVA

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

[illegible]

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. NICOLAS ANQUETIL
Universidade de Lille-1 - França

[illegible]

PROF. NIKOLAOS TSANTALIS
Departamento de Ciência da Computação - Concordia University

Belo Horizonte, 29 de julho de 2014.

Agradecimentos

Agradeço a todas as pessoas que contribuiram, de forma direta ou indireta, para que eu chegasse até este ponto do curso.

A família é o alicerce para a nossa educação, e não poderia deixar de agradeçer aos meus pais, Messias e Marilu, tudo que representaram na minha formação como pessoa. Agradeço também de forma especial a minha esposa, Izadora, devido a importância que ela teve nesse período, me dando coragem e apoio para ir em busca de meus sonhos.

Agradeço ao professor Marco Tulio, que foi muito mais que um orientador neste trabalho. Deixo registrados a profunda admiração e respeito que tenho por ele, pois pude perceber o grande profissional e pessoa que é durante nossa convivência no mestrado.

Agradeço ao professor Ricardo Terra, meu coorientador, sua dedicação e energia. Certamente sua contribuição foi inestimável para a qualidade do trabalho desenvolvido.

Agradeço aos amigos do DCC, aos colegas do grupo de pesquisa (ASERG) as dicas e apoio prestados.

Por fim, agradeço ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) um curso de tal nível de excelência e o suporte financeiro e acadêmico.

Resumo

Extração de Método é uma refatoração chave para aprimorar a manutenibilidade de sistemas, visto que métodos pequenos com nomes significativos favorecem legibilidade e reúso de código. Por tais razões, Extração de Método é uma das refatorações mais populares e versáteis. Entretanto, estudos empíricos recentes indicam que ferramentas dedicadas a automação dessa refatoração são frequentemente sub-utilizadas e, mais importante, não oferecem auxílio aos desenvolvedores na identificação de potenciais fragmentos de código a serem extraídos.

Para suprir essa deficiência, propõe-se uma abordagem para identificar oportunidades de Extração de Método que podem ser aplicadas de forma automatizada por ferramentas de refatoração de IDEs. A abordagem utiliza uma heurística para ordenar as oportunidades identificadas, centrada no princípio de *design* de separação de responsabilidades. Especificamente, assume-se que as dependências estruturais estabelecidas por um fragmento de código candidato a extração devem ser bem diferente daquelas estabelecidas pelo código restante no método original.

Em um primeiro estudo envolvendo um conjunto sintetizado de oportunidades de Extração de Método, introduzidas pela expansão de invocações de métodos (*Inline Method*), a abordagem proposta mostrou-se mais efetiva (considerando revocação e precisão) do que uma ferramenta que representa o estado da arte. Em um segundo estudo envolvendo 13 sistemas de código aberto, uma a abordagem proposta alcançou revocação global de 59,1%. Além disso, ao se tolerar pequenas variações em oportunidades de Extração de Método conhecidas (por exemplo, incluindo/excluindo uma única sentença de código), a revocação global sobe para 66,6%.

Palavras-chave: Refatoração Extração de Método, Sistemas de recomendação, Conjunto de dependências.

Abstract

Extract Method is a key refactoring for improving software maintainability. In fact, small methods with meaningful names improve source code readability and also favor reuse. For such reasons, Extract Method is one of the most popular and versatile refactorings. However, recent empirical research shows that refactoring tools designed to automate this refactoring are often underused and, more important, they do not support developers in the identification of potential code fragments for extraction.

To address this issue, we propose an approach to identify Extract Method refactoring opportunities that are directly automated by IDE-based refactoring tools. The proposed approach relies on a heuristic to rank the identified refactoring opportunities based on the design principle of separation of concerns. Specifically, we assume that the structural dependencies established by a Extract Method candidate should be very different from the ones established by the remaining statements in the original method.

In a first study using synthesized Extract Method opportunities introduced by inlining method invocations, shows that our approach is more effective (w.r.t. recall and precision) than a state-of-the-art tool. In a second study, with 13 open-source systems our approach achieved an overall recall of 59.1%. Moreover, when we tolerate suggestions that are slight variations of known Extract Method instances (e.g., including/excluding a single statement), recall rises to 66.6%.

Keywords: Extract Method refactoring, Recommendation systems, Dependency sets.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

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

1.1 Motivation

A well-known property of software systems is their need to evolve, driven by the pressure of changing real-world requirements [Lehman, 1980]. Unfortunately, this evolution process usually leads to deviations from the original design and an increase in the complexity. This phenomenon, usually referred as software aging, makes software maintenance and evolution increasingly costly [Parnas, 1994]. Although it is difficult to completely avoid software aging, developers can tackle this phenomenon with refactoring.

Refactoring is defined as the process of changing a software system to improve its internal structure in such a way that it does not alter the external behavior of the code [Opdyke, 1992; Fowler, 1999]. Thus, refactoring is an important practice to recover from design deviations and to improve the quality of the source code. While refactoring is usually associated with software maintenance, it is also employed in the earlier phases of development. For example, practitioners of Test Driven Development (TDD) advocate the use of refactoring as an essential step in a software development cycle: (i) *write a test*; (ii) *make the test work quickly*; and (iii) *refactor to add design decisions one at a time* [Beck, 2003].

Fowler [1999] also presents a catalog of refactorings, including Extract Method, Inline Method, Move Method, Pull Up Field, Replace Inheritance with Delegation, and many others. Particularly, this dissertation focuses on Extract Method, which we define in the next section.

1.1.1 Extract Method Refactoring

When a method contains a code fragment that can be grouped together and the maintainer decides to turn that fragment into a new method, we call this refactoring Extract Method [Fowler, 1999]. Figures 1.1 and 1.2 present a minimal code example, before and after the refactoring, to illustrate its mechanics. In Figure 1.1, the statements at lines 4 and 5 were extracted into a new method, called `printDetails`, which is shown in Figure 1.2.

```
1 void printOwing(double amount) {
2   printBanner();
3   //print details
4   System.out.println("name:" + _name);
5   System.out.println("amount" + amount);
6 }
```

Figure 1.1. Code before Extract Method

```
1 void printOwing(double amount) {
2   printBanner();
3   printDetails(amount);
4 }
5 void printDetails (double amount) {
6   System.out.println ("name:" + _name);
7   System.out.println ("amount" + amount);
8 }
```

Figure 1.2. Code after Extract Method

One of the main motivations for Extract Method is to decompose a method that is too long or difficult to understand. Short methods with meaningful names improve readability and also favor reuse. Besides, overriding methods in subclasses is easier when they are finely grained. In fact, empirical studies suggest that Extract Method is one of the most popular and versatile refactorings [Murphy et al., 2006; Murphy-Hill et al., 2012; Wilking et al., 2007]. For example, Tsantalis et al. [2013] analyzed history data from three open-source projects and discovered Extract Method instances with nine distinct purposes.

1.1.2 Current Tool Support

Despite the simplicity of the aforementioned example (Figures 1.1 and 1.2), refactoring is usually a complex task, specially considering all possible scenarios and preconditions involved in such process [Schäfer et al., 2009; Borba et al., 2004; Steimann and Thies, 2009; Verbaere et al., 2006; Opdyke, 1992]. Therefore, tool support is crucial to increase productivity and reliability. Currently, most mainstream IDEs offer some kind of refactoring automation. For example, Eclipse provides automated Extract Method application given a selected code fragment, as long as the selection respects some preconditions.

However, despite the availability of supporting tools, recent empirical research shows that these tools, especially those supporting Extract Method refactorings, are most of the times underused [Negara et al., 2013; Kim et al., 2012; Murphy-Hill et al., 2012; Murphy-Hill and Black, 2008b]. For example, in a study including developers working in their natural environment, Negara et al. [2013] found that the number of participants who are aware of the automated support for Extract Method but still apply it manually is higher than the number of participants who predominantly apply it automatically. Moreover, a study from Murphy-Hill and Black [2008a] revealed that some users were discouraged to use the Extract Method tool of Eclipse IDE due to usability problems regarding the manual selection of valid code fragments. More important, current tools focus only on automating refactoring application, but developers expend considerable effort on the manual identification of refactoring opportunities.

In face of this scenario, we advocate that recommendation systems designed to identify Extract Method refactoring opportunities can play an important role in modern IDEs. Particularly, they can be effective instruments to increase the popularity of refactoring tools among IDE users since they avoid the manual step of selecting a valid code fragment. More important, such systems may suggest valuable refactoring opportunities that developers are not aware, alleviating the burden to manually inspect the source code.

1.2 Proposed Approach

We previously discussed how the automated identification of Extract Method refactoring opportunities may leverage existing tool support and contribute to a widespread adoption of refactoring practices. In this dissertation, we propose a novel approach, based on structural dependencies, to identify and rank Extract Method refactoring opportunities that can be directly automated by IDE-based refactoring tools.

1.2.1 Problem Statement

We aim to design, implement, and evaluate a refactoring recommendation approach, which should have the following input and output:

- **Input:** The source code of a method.

- **Output:** Potential Extract Method refactoring opportunities, such that:

  1. They can be automatically extracted using the existing tool support.
  2. They preserve the program's original behavior.
  3. They are ranked by relevance, inspired by the *minimize coupling/maximize cohesion* design guideline, i.e., they should encapsulate a well-defined computation (high cohesion) that is independent from the input method (low coupling).

While the first two properties of the output can be formally specified, the third one is subjective in its essence. The notion of relevance in recommendation systems is subjected to human judgment. However, we desire that the system's notion of relevance is as close as possible from an expert's notion of relevance.

In order to illustrate the proposed recommendation system, we use the method presented in Figure 1.3, from the JHotDraw system. In this method, lines 5–9 are closely related to each other, encapsulating a well-defined computation responsible for the initialization of field `fConnectors`. Specifically, an object of type `Vector` is instantiated and populated. This code can be better organized by extracting these statements into a new method, preferably with a descriptive name (e.g., `initFConnectors`). Thus, a possible output in this case is a recommendation suggesting to extract lines 5–9 to a new method.

```
 1 private void initialize() {
 2   setText("node");
 3   Font fb = new Font("Helvetica", Font.BOLD, 12);
 4   setFont(fb);
 5   fConnectors = new Vector(4);
 6   fConnectors.addElement(new LocatorConnector(this, Locator.north()));
 7   fConnectors.addElement(new LocatorConnector(this, Locator.south()));
 8   fConnectors.addElement(new LocatorConnector(this, Locator.west()));
 9   fConnectors.addElement(new LocatorConnector(this, Locator.east()));
10 }
```

Figure 1.3. An example method from the JHotDraw system

1.2.2 Proposed Solution: JExtract

JExtract, the tool we designed to implement our approach, is an Extract Method refactoring recommendation system. Specifically, given one or more methods, the output of our tool is a ranked list of refactoring opportunities. Moreover, when developers accept a recommendation, JExtract automatically applies it.

Figure 1.4 depicts the main elements involved in our recommendation approach, which can be summarized in the following two subproblems:

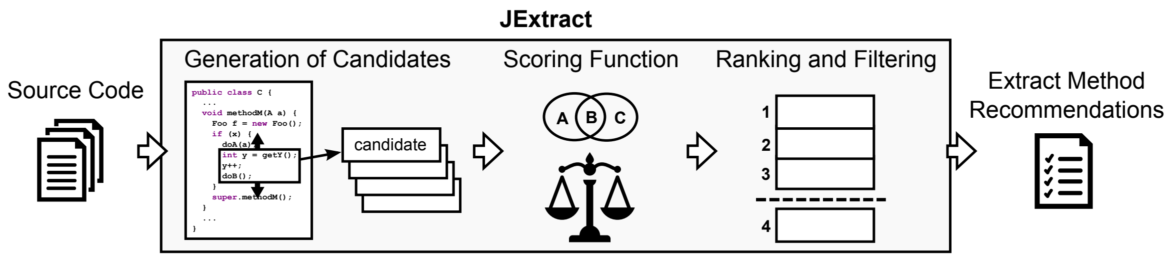


Figure 1.4. Overview of the proposed approach

1. *Candidates Generation:* Given a method, JExtract identifies all viable Extract Method candidates, regardless of their relevance. More specifically, it analyzes the source code and enumerates each Extract Method candidate that respects a set of preconditions necessary to guarantee its viability. Additionally, we enforce a configurable size threshold to avoid suggesting the extraction of very small code fragments (e.g., with one or two statements).

2. *Ranking:* Given a list of Extract Method candidates, JExtract ranks the items by relevance and exclude non-relevant ones. More specifically, it relies on a scoring function to classify the candidates according to their potential to improve the design of the code. This function, which is the key element of our solution, is based on the *minimize coupling/maximize cohesion* design guideline. We assume that *the structural dependencies established by top-ranked Extract Method candidates should be very different from the ones established by the remaining statements in the original method.* When this assumption holds, the extraction tends to encapsulate a well-defined computation with its own set of dependencies (high cohesion) and that is also independent from the original method (low coupling).

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 recommendation systems applied to software engineering. We also present an overview on tools for detecting Extract Method refactoring opportunities and related work.
- **Chapter 3** presents the proposed approach. We detail the two main phases of the approach (*Candidates Generation* and *Ranking*) and illustrate the whole process using an example method. Furthermore, this chapter reports an initial study we designed to calibrate our approach, exploring ranking strategies and threshold parameters. Finally, we present JExtract, the tool that implements our proposed approach.
- **Chapter 4** reports the evaluation of our approach. First, we investigate how our approach performs compared to state-of-the-art ones. Second, we report a study with a sample of 1,182 synthesized Extract Method instances from 13 open-source systems. Last, we investigate alternative ranking strategies to confirm the findings of the exploratory study reported in Chapter 3.
- **Chapter 5** presents the final considerations of this dissertation, outlining its main contributions, limitations, and future work.

1.4 Publications

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

- Danilo Silva, Ricardo Terra, and Marco Tulio Valente. Recommending Automated Extract Method Refactorings. In *22nd IEEE International Conference on Program Comprehension (ICPC)*, pages 146–156, 2014.
- Danilo Silva, Ricardo Terra, and Marco Tulio Valente. JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings. *V Congresso Brasileiro de Software: Teoria e Prática, Sessão de Ferramentas*, pages 1–8, 2014. (Under submission).

Chapter 2

Background

In this chapter, we discuss background and work related to this dissertation. First, Section 2.1 presents a brief introduction on *Recommendation Systems for Software Engineering*, since our approach falls in this category of system. Next, we present previous work in the area of refactoring recommendation systems, dedicating special attention to two tools: JDeodorant (Section 2.2), a state-of-the-art refactoring recommendation system that we compared our approach with, and JMove (Section 2.3), a Move Method recommendation system that our technique for ranking Extract Method candidates is inspired by. Section 2.4 briefly presents other approaches related to the identification of Extract Method refactoring opportunities. Last, Section 2.5 concludes this chapter with some final remarks.

2.1 Recommendation Systems for Software Engineering

Recommendation Systems for Software Engineering (RSSEs) is an emerging research area [Robillard et al., 2010]. A RSSE is a software application that provides potential valuable information for a software engineering task in a given context. For example, the solution we propose in this dissertation provides information to aid the task of decomposing a method into smaller parts, in the form of code fragments recommended for extraction. Moreover, such systems may employ a variety of techniques, e.g., there are approaches based on search algorithms [O'Keeffe and Cinnéide, 2006; Seng et al., 2006], Relational Topic Model (RTM) [Bavota et al., 2014], metrics-based rules [Marinescu, 2004], clustering [Anquetil and Lethbridge, 1999; Mitchell and Mancoridis, 2006], etc.

Finally, RSSEs can help developers in a wide range of activities, besides refactoring-related tasks. For example, current RSSEs can recommend relevant source code fragments to help developers to use frameworks and APIs (CodeBroker [Ye and Fischer, 2005], Strathcona [Holmes et al., 2006], and APIMiner [Montandon et al., 2013]), software artifacts that must be changed together (eRose [Zimmermann et al., 2005]), parts of the software that should be tested more cautiously (Suade [Nagappan et al., 2006]), and tasks for repairing software architectures (ArchFix [Terra et al., 2014]).

2.2 JDeodorant

JDeodorant is a RSSE that suggests and applies refactoring operations on Java systems, aiming to solve common code smells, such as *Feature Envy*, *Long Method*, and *God Class* [Tsantalis and Chatzigeorgiou, 2009, 2011; Fokaefs et al., 2012]. Specifically, JDeodorant employs an automated approach to identify Extract Method refactoring opportunities, proposed by Tsantalis and Chatzigeorgiou [2011]. Since existing IDEs focus only on automating the extraction of statements indicated by the developer, the authors aim to fill the gap by recommending code fragments that could benefit from decomposition.

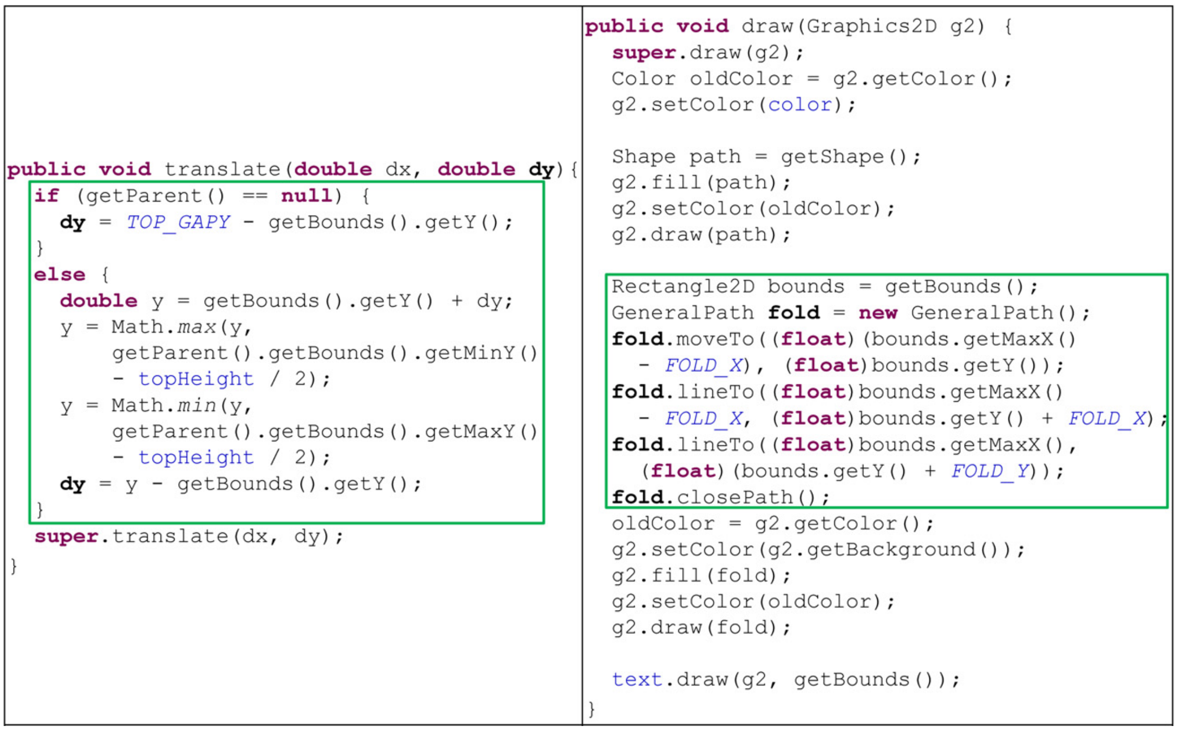
Identification of Code Slices: JDeodorant employs backward slicing, a static analysis technique to identify the slice of code that may affect a variable at a given point. This technique relies on the Program Dependence Graph (PDG) to represent the methods under analysis. In a PDG, nodes represent statements and edges represent dependencies (control or data). Therefore, backward slicing consists on selecting statements connected in the PDG, starting with a set of seed statements.

The identification of data dependencies is crucial to generate precise and correct slices, as well as to preserve program's behavior after code extraction. Thus, the authors employ a variety of program analysis techniques to construct the PDG, considering issues such as polymorphic method calls, unstructured control flow, exceptions, and aliasing.

While traditional slicing algorithms consider the entire method body as a region where the slice may expand, the authors adopt the concept of block-based slicing, introduced by Maruyama [2001]. By employing this concept, it is possible to produce more than one slice for a given slicing criterion, constrained by different block-based regions of a method. The authors propose two algorithms, based on different criteria, to select the slices to be extracted:

- *Complete computation slice:* It generates a slice that modifies the value of a variable. The statements that assign a value to a given variable are initially selected as seeds, which are expanded using backward slicing.
- *Object state slice:* It generates a slice that affects the state of an object reference. The statements that affect the state of the object are selected as seeds, which are also expanded using backward slicing.

Figure 2.1 exemplifies both slicing criteria, the first using variable `dy` and the second using the object reference `fold`. By selecting such slices, it is possible to decompose the method by applying an Extract Method refactoring.



```
public void translate(double dx, double dy){
  if (getParent() == null) {
    dy = TOP_GAPY - getBounds().getY();
  }
  else {
    double y = getBounds().getY() + dy;
    y = Math.max(y,
        getParent().getBounds().getMinY()
        - topHeight / 2);
    y = Math.min(y,
        getParent().getBounds().getMaxY()
        - topHeight / 2);
    dy = y - getBounds().getY();
  }
  super.translate(dx, dy);
}
```

```
public void draw(Graphics2D g2) {
  super.draw(g2);
  Color oldColor = g2.getColor();
  g2.setColor(color);

  Shape path = getShape();
  g2.fill(path);
  g2.setColor(oldColor);
  g2.draw(path);

  Rectangle2D bounds = getBounds();
  GeneralPath fold = new GeneralPath();
  fold.moveTo((float)(bounds.getMaxX()
    - FOLD_X), (float)bounds.getY());
  fold.lineTo((float)bounds.getMaxX()
    - FOLD_X, (float)bounds.getY() + FOLD_X);
  fold.lineTo((float)bounds.getMaxX(),
    (float)(bounds.getY() + FOLD_Y));
  fold.closePath();
  oldColor = g2.getColor();
  g2.setColor(g2.getBackground());
  g2.fill(fold);
  g2.setColor(oldColor);
  g2.draw(fold);

  text.draw(g2, getBounds());
}
```

Figure 2.1. *Complete computation slice* and *Object state slice* examples

It is worth noting that the selected slices may contain statements that cannot be removed from the original method, for example, when they are also required for the computation of other variable or other object states. In this case, these statements must be duplicated, i.e., they should be present both in the extracted method and the original method. However, such duplication may not preserve behavior. For this reason, the authors defined a set of rules to prohibit harmful duplication scenarios. For example, a statement that modifies the state of an object cannot be duplicated.

Evaluation #1: The authors considered several aspects of the recommendations, such as precision, usefulness, behavior-preservation, and impact on source code quality metrics. In a first experiment on JFreeChart system, an independent expert answered the following questions regarding the recommendations triggered by JDeodorant:

1. *Does the code fragment suggested to be extracted as a separate method have a distinct and independent functionality compared to the rest of the original method?*

2. *Does the application of the suggested refactoring solve an existing design flaw (e.g., by decomposing a complex method, removing a code fragment that is duplicated among several methods, or extracting a code fragment suffering from Feature Envy)?*

From the 64 evaluated recommendations, the expert agreed on 57 (89%) in the first question and on 27 (42%) in the second question. From such results, the authors argued that the extracted methods showed to be useful and cohesive. For the behavior preservation concern, the authors relied on the existing automated tests, which covered 41 from the 64 triggered recommendations. All tests ran without errors after the application of the suggested refactoring opportunities, which indicates with a relative certainty that no bugs were introduced by the code transformations, an essential premise for refactoring.

Furthermore, the authors verified that the recommended refactorings had a positive impact on the slice based cohesion metrics proposed by Ott and Thuss [1993], namely *Overlap*, *Tightness*, and *Coverage*. Specifically, the authors applied the refactorings approved by the expert evaluator and compared: (i) metric values before the refactoring and (ii) average metric values for the original and extracted method after the refactoring. The average increase was +0.303 for *Overlap*, +0.319 for *Tightness*, and +0.113 for *Coverage*.[1]

Evaluation #2: In this evaluation, the suggestions triggered by the tool were contrasted with suggestions identified by expert developers. More specifically, two experts analyzed the source code of two systems (they had previously worked on) to identify method decomposition opportunities, which in this case were considered the ideal answer. Thereby, the experts received as input only a list of methods to analyze, which was a sample of the methods that had at least one recommendation triggered by JDeodorant. In this scenario, JDeodorant achieved the precision and recall of 51% and 69%, respectively. Therefore, the authors concluded that the tool is able to identify refactoring opportunities that are usually found by human experts.

[1]Metric values range over the $[0, 1]$ interval.

2.3 JMove

JMove is a RSSE, proposed by Sales et al. [2013], that recommends Move Method refactoring opportunities for Java systems, i.e., it detects methods located in incorrect classes and then suggests moving such methods to more suitable ones. JMove is influenced by the work of Tsantalis and Chatzigeorgiou [2009] that 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, the authors proposed a novel heuristic to detect misplaced methods, based on the similarity of dependency sets.

Dependency Sets: The proposed technique first retrieves the set of structural dependencies established by a given method m located in a class C. The set of structural dependencies, which are used to compute the similarity, includes the types that are referenced in the source code. A method establishes a dependency on a certain type C when:

- Calls a method defined in type C.
- Accesses a field defined in type C.
- Instantiates an object of type C.
- Declares a variable or parameter of type C.
- Declares C as the return type of a method.
- Handles an exception of type C.
- Annotates the code with an annotation type C.

Next, it computes two similarity coefficients: (a) the average similarity between the set of dependencies established by m and by the remaining methods in C; and (b) the average similarity between the dependencies established by m and by the methods in another class C_i. If the similarity measured in step (b) is greater than the similarity measured in (a), it is inferred 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 candidate class to receive m. By computing this similarity for every class C_i of the system, the tool is able to suggest the best class to receive the method.

Similarity Coefficients: To measure the similarity between the sets of dependencies established by two methods the authors rely on similarity coefficients, which are usually employed to measure the similarity between two generic sets. The authors investigated the use of 18 different similarity coefficients, using JHotDraw system, to choose the most suitable one. As the result, they decided for the use of *Sokal and Sneath 2* coefficient.

Evaluation: The authors evaluated the approach using 14 open-source Java systems. They introduced Move Method opportunities in those systems by applying random Move Method operations. More specifically, they randomly selected a sample of 3% of the classes in each system and manually moved a method of them to new classes, also randomly selected. After that, the authors compared the set of suggestions triggered by *JMove* with the set of synthesized Move Method instances, achieving an average precision of 60.63% and an average recall of 81.07%.

2.4 Other Approaches

2.4.1 Slicing and PDG Based Code Extraction

Weiser [1981] formally defines slicing as a method for automatically decomposing programs by analyzing their data and control flow, represented usually in a Program Dependency Graph (PDG). Most studies found in the literature for function (or method) extraction are based on the concept of program slicing. While not a comprehensive list, we may cite the work of Gallagher and Lyle [1991]; Cimitile et al. [1996]; Lanubile and Visaggio [1997]; Lakhotia and Deprez [1998]; Maruyama [2001]; Komondoor and Horwitz [2003]; Harman et al. [2004]; Abadi et al. [2009]; and Ettinger [2012]. Nevertheless, these approaches extract code slices based on some kind of user input (e.g., a certain variable or point of interest in the code). As presented in Section 2.2, JDeodorant [Tsantalis and Chatzigeorgiou, 2011] is also centered on slicing. However, in contrast to the aforementioned approaches, it provides automated identification of Extract Method refactoring opportunities.

While not directly based on slicing, Sharma [2012] also proposes an approach to identify Extract Method refactoring opportunities. Specifically, the author employ a *longest edge removal algorithm* in a Data and Structure Dependency (DSD) graph, which is a combination of a data flow graph and a structure dependency graph. In a DSD graph, edges may represent: (i) data dependencies or (ii) structural dependencies, i.e., a dependency between a statement and its enclosing block statement (e.g., `if`, `for`, `while`, etc.). The author proposes a technique to partition the DSD graph, forming disconnected sub-graphs that correspond to code fragments candidates for extraction.

Specifically, the approach employs a heuristic to remove edges iteratively from the DSD graph. This heuristic is centered on the idea that two statements connected by a data dependency edge and placed closer in a DSD graph are likely to be related than two connected statements that are placed farther in the DSD graph. Therefore, the longer the edge, the lower the likelihood that the connected nodes should reside in a single method.

2.4.2 Visualization Based Approaches

Visualization techniques may be employed to aid the identification of Extract Method opportunities. For example, Kanemitsu et al. [2011] propose a visual representation of the PDG of a method that includes the notion of the weight of edges, which reflects as the distance between nodes. The authors define the distance between every pair of nodes having a data dependency in terms of three possible scenarios:

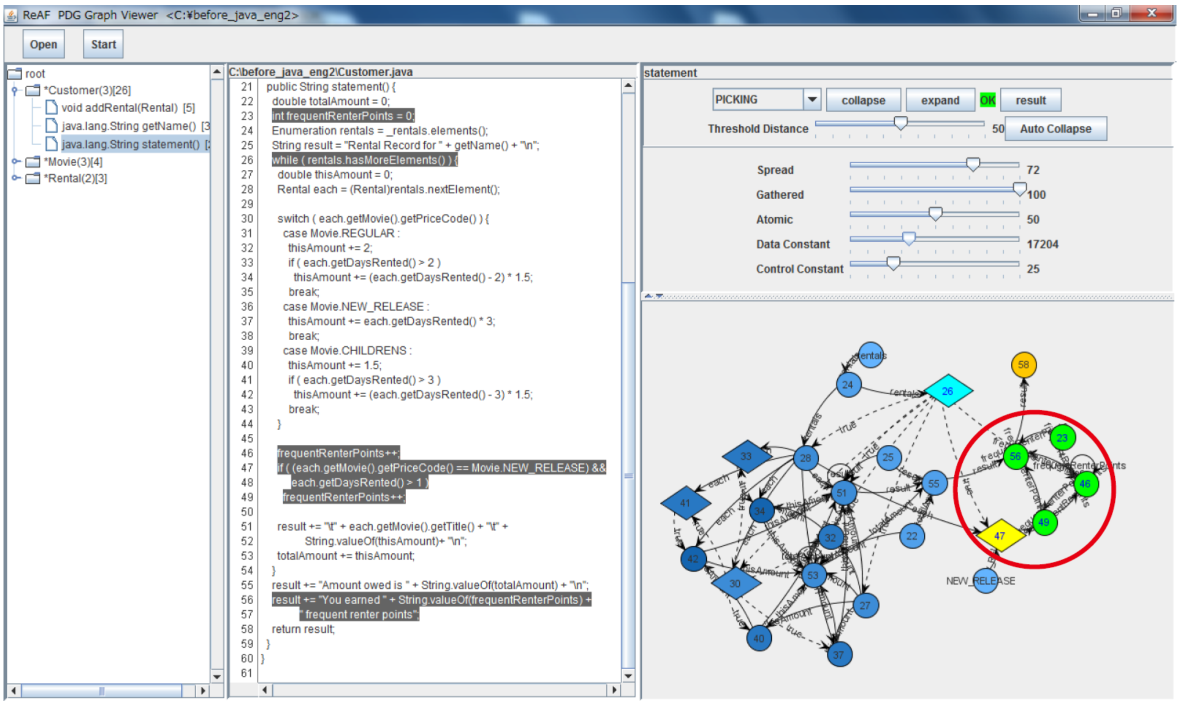
- *Atomic Data Dependency:* A statement defines (initializes) a variable, and a single statement references the variable.

- *Spread Data Dependency:* A statement defines a variable, and many other statements reference the variable.

- *Gathered Data Dependency:* A statement references many variables that are defined in other statements.

A screenshot of their supporting tool, called ReAF, is presented in Figure 2.2. ReAF allows developers to visualize the PDG and select nodes for extraction. ReAF can also automatically merge nodes in the PDG visualization taking as input a threshold value for the weight of the edges. More important, these merged nodes corresponds to Extract Method refactoring opportunities.

Similarly, Kaya and Fawcett [2013] propose an approach, based on a treemap visualization, that uses density of variable references in code fragments to identify main tasks that could be extracted from large methods, composed of several blocks. Their treemap visualization tool helps developers to observe and analyze suggested refactorings. In the proposed technique, an input method is represented with a placement tree where each node represents a different scope. Scopes are defined by all code enclosed within braces. A scope may include children scopes, forming a hierarchical structure. The authors define the notion of dominant variables, which are the most referenced ones inside a particular scope. A single dominant variable is assigned for each scope


ReAF PDG Graph Viewer <C:¥before_java_eng2>
Open
Start
root
*Customer(3)[26]
void addRental(Rental) [5]
java.lang.String getName() [3
java.lang.String statement() [
*Movie(3)[4]
*Rental(2)[3]
statement
PICKING
collapse
expand
OK
result
Auto Collapse

C:\before_java_eng2\Customer.java

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
  while ( rentals.hasMoreElements() ) {
    double thisAmount = 0;
    Rental each = (Rental)rentals.nextElement();

    switch ( each.getMovie().getPriceCode() ) {
      case Movie.REGULAR :
        thisAmount += 2;
        if ( each.getDaysRented() > 2 )
          thisAmount += (each.getDaysRented() - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE :
        thisAmount += each.getDaysRented() * 3;
        break;
      case Movie.CHILDRENS :
        thisAmount += 1.5;
        if ( each.getDaysRented() > 3 )
          thisAmount += (each.getDaysRented() - 3) * 1.5;
        break;
    }

    frequentRenterPoints++;
    if ( (each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
         each.getDaysRented() > 1 )
      frequentRenterPoints++;

    result += "\t" + each.getMovie().getTitle() + "\t" +
              String.valueOf(thisAmount)+ "\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points";
  return result;
 }
}
```

Threshold Distance	50
Spread	72
Gathered	100
Atomic	50
Data Constant	17204
Control Constant	25

Figure 2.2. Screenshot of ReAF tool [Kanemitsu et al., 2011]

using a set of proposed rules to cover the cases of ties. In the proposed visualization, illustrated in Figure 2.3, the nodes of the treemap are colored according to their dominant variable.

Moreover, the authors propose the extraction of code when any of the two following patterns are observed: (i) a large code fragment with a color different from its parent and (ii) consecutive sibling nodes with the same color. The authors evaluated their tool on C++ systems and reported cases of code fragments extracted from long methods with the aid of their visualization approach.

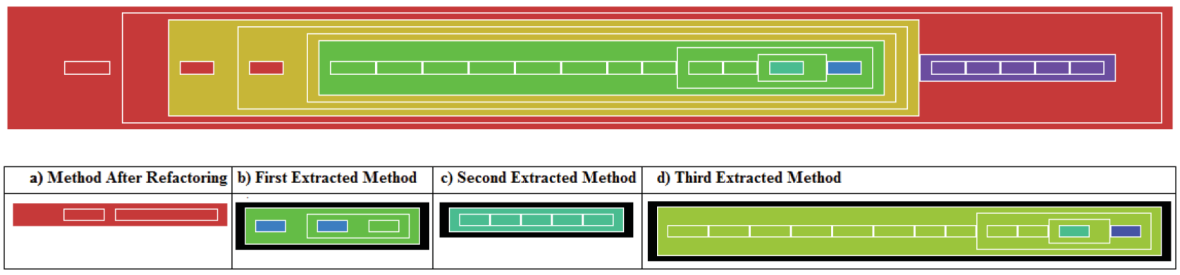

a) Method After Refactoring
b) First Extracted Method
c) Second Extracted Method
d) Third Extracted Method

Figure 2.3. Treemap visualization proposed by Kaya and Fawcett [2013]

2.4.3 Combining Structural and Linguistic Information

Another line of research aims to combine structural and linguistic information extracted from source code to identify groups of related statements. Sridhara et al. [2011] propose an approach to: (i) automatically identify code fragments that represent a high level action and (ii) describe it using natural language. Their technique may be applied, for example, to automatically generate descriptive source code comments. The authors propose a set of heuristics, considering structural and linguistic information, to group related statements that fall into one of the following patterns:

1. A sequence of similar statements that represents a single action.
2. A conditional block that performs an action with subtle variations based on its condition.
3. Loop constructs that implement a set of predefined actions, e.g., find an element in a collection.

For each recognized pattern, the authors define specific rules to generate a phrase, in natural language, describing the action. The authors evaluated their approach with a sample of 75 summarized code fragments and 15 human evaluators, reporting that: (i) in 94.6% of the cases evaluators agreed that the identified code fragments actually represented a high level action; and (ii) in 89.7% of the cases evaluators agreed that the description generated by the approach was appropriate.

Although the authors proposed their technique originally for automatic documentation generation, they also mention other usage scenarios, including the identification of Extract Method opportunities. However, its applicability is limited to the patterns the technique implements.

Wang et al. [2014] presents the SEGMENT tool, which also combines structural and linguistic information. SEGMENT separates meaningful blocks within a method, by inserting blank lines, to increase readability. The tool uses a heuristic based on both program structure and naming information to identify meaningful blocks, i.e., consecutive statements that logically implement a high level action. The proposed approach is composed by the following phases:

1. *Identify Initial Blocks:* It groups related statements to form blocks of code. These statements may be related by: (i) syntactical similarities (SynS blocks); (ii) a data-flow chain (DFC blocks); and (iii) compound statements including {`switch`, `while`, `if`, `for`, `try`, `do`, `synchronized`} (E-SWIFT blocks).

2. *Remove Block Overlap:* It joins consecutive overlapping blocks according to a set of proposed rules.

3. *Refine SynS Blocks:* It splits large SynS blocks using additional information extracted from word usage and naming conventions.

4. *Merge Short Blocks:* It merges very small blocks or individual statements with other blocks, also using linguistic information.

In the final output, blank lines are inserted between each identified block, aiding the comprehension of the code. The authors report that SEGMENT's effectiveness from a code reader's perspective achieved a precision of 66% and a recall of 87.6%. Although their approach does not focus on the identification of Extract Method opportunities, their technique can be adapted to it.

2.4.4 Leveraging Current Refactoring Tools

Besides automated identification of Extract Method opportunities, some studies focus on issues of current refactoring tools. For example, Murphy-Hill and Black [2008a] reported a study to investigate the usability of current refactoring tools. The authors observed several usability issues in a study where 11 experienced programmers performed a number of Extract Method refactoring tasks using Eclipse IDE. Specifically, the study revealed that: (i) programmers need support in making a valid selection to prevent errors; and (ii) programmers need expressive and understandable feedback to recover from violated preconditions. As result of this study, the authors defined a set of guidelines that could improve the user experience using refactoring tools. Moreover, the authors designed three refactoring assisting tools according to the proposed guidelines to demonstrate how speed, accuracy, and user satisfaction can be significantly increased.

On the other hand, Ge et al. [2012] tackle the automatic refactoring tools underuse problem. The authors found that one cause of this problem is that developers sometimes fail to recognize that they are going to refactor. For this reason, the authors propose BeneFactor, a refactoring tool that detects developers' manual refactoring, reminds them that automatic refactoring is available, and can automatically complete their refactoring. Specifically, they rely on a set of manual refactoring workflow patterns, including those frequently adopted by developers when performing Extract Method, to detect such scenarios.

2.5 Final Remarks

This chapter presented an overview of the state-of-the-art in refactoring recommendations system, focusing on Extract Method refactorings. Even though most approaches are based on the concept of slicing and PDGs, such as JDeodorant, recent studies also explore structural and linguistic information, visualization techniques, and density of variable references.

In the next chapter, we detail our Extract Method refactoring recommendation approach. However, we can already outline two main differences from the state-of-the-art approaches:

1. We do not rely on slicing techniques to identify related statements to be extracted. Rather, we rely on the similarity between dependency sets, influenced by JMove's technique to measure similarity between methods. Nevertheless, we extended JMove's notion of dependencies to better fit our problem.

2. We propose a heuristic for ranking Extract Method opportunities, i.e., we define a scoring function that aims to express the relevance of each opportunity.

Chapter 3

Proposed Approach

This chapter explains in details the proposed approach for recommending Extract Method refactoring opportunities, previously introduced in Section 1.2. We start by discussing each of the phases involved in the recommendation process, namely the candidates generation phase (Section 3.1) and the ranking phase (Section 3.2), we also provide a full example to illustrate the approach (Section 3.3). Following, we present a preliminary exploratory study (Section 3.4) conducted when designing the solution and the tool we built to implement the approach (Section 3.5). Finally, we end this chapter with some final remarks (Section 3.6).

3.1 Candidates Generation

This phase is responsible for identifying all Extract Method refactoring possibilities for a method, which we refer as Extract Method candidates. In this context, a candidate is a range of statements that could be extracted into a new method, without introducing any compilation error or behavior modification. The highlighted code fragment shown in Figure 3.1 is an example of an Extract Method candidate. Note that many other candidates can be identified in that method. For example, lines `09–13` or lines `09–14` are also potential candidates, just to cite a few.

3.1.1 Candidates Generation Algorithm

Before the identification of candidates, we must first extract from the source code the relevant information our algorithm needs; thus creating a model representing the source code of the method. The main entities represented in this model are statements and blocks, which are series of sequential statements that follow a linear control flow in the

```
01 public void mouseReleased(MouseEvent me) {
02           for (Button btn : buttons) {
03           2/01 int cx = btn.fig.getX() + btn.fig.getWidth() - btn.icon.getIconWidth();
04           2/02 int cy = btn.fig.getY();
05           2/03 int cw = btn.icon.getIconWidth();
06           2/04 int ch = btn.icon.getIconHeight();
07           2/05 Rectangle rect = new Rectangle(cx, cy, cw, ch);
08                if (rect.contains(me.getX(), me.getY())) {
09                3/01 Object metaType = btn.metaType;
10   1/01         3/02 FigClassifierBox fcb = (FigClassifierBox) getContent();
11                3/03 FigCompartment fc = fcb.getCompartment(metaType);
12           2/06 3/04 fc.setEditOnRedraw(true);
13                3/05 fc.createModelElement();
14                3/06 me.consume();
15                3/07 return;
16                }
17           }
18   1/02    super.mouseReleased(me);
19 }
```

Figure 3.1. An Extract Method candidate from ArgoUML system (lines 10–13)

Control Flow Graph (CFG).[1] The structure of such model is represented in Figure 3.1, where we can notice that each statement is labeled using the `X/Y` pattern. In this case, `X` and `Y` denote the block and the statement, respectively. For example, `2/03` is the third statement of the second block, which declares variable `cw`.

Using the model aforementioned described as input, we generate all Extract Method candidates using Algorithm 1.

Algorithm 1 Candidates generation algorithm

Input: A method M
Output: List with Extract Method candidates

```
 1: $Candidates \leftarrow \emptyset$
 2: for all block $B \in M$ do
 3:     $n \leftarrow statements(B)$
 4:     for $i \leftarrow 1, n$ do
 5:         for $j \leftarrow i, n$ do
 6:             $C \leftarrow subset(B, i, j)$
 7:             if $isValid(C)$ then
 8:                 $Candidates \leftarrow Candidates + C$
 9:             end if
10:         end for
11:     end for
12: end for
```

The three nested loops in Algorithm 1 (lines 2, 4, and 5) iterates through each subsequence of statements, of each block, to consider them for extraction. More important, this algorithm enforces the following properties:

[1]A CFG is a directed graph in which nodes represent basic blocks and edges represent control flow paths [Allen, 1970].

- Only contiguous statements inside a block are selected, as the selection is defined by the range i to j. In Figure 3.1, for example, it is not possible to select a candidate with `3/02` and `3/04` without including `3/03`.

- It is not possible to select only part of a statement. That is, when a statement is selected, the respective children statements are also included. In Figure 3.1, for example, when statement `2/06` is selected, its children statements `3/01` to `3/07` are also included.

- The selected statements are direct children of a single block of statements. In Figure 3.1, for example, it is not possible to generate a candidate with both `2/06` and `3/01` since they belong to distinct blocks.

3.1.2 Preconditions Checking

It is important to note that not every possible subsequence turns into a candidate because Algorithm 1 relies on the $isValid$ function (line 7) to check two kinds of preconditions: Viability Preconditions and Quality Preconditions.

3.1.2.1 Viability Preconditions

We check whether the candidates are applicable and preserve the program's behavior. For example, when two or more variables are assigned by a code fragment selected for extraction and they are further used by other statements, we cannot apply the Extract Method. Fundamentally, there is no way to define a method with multiple return values in Java, unless an auxiliary data structure is used.[2] The code fragment enclosed at lines `03`–`05` in Figure 3.1 is an example of such scenario since variables `cx`, `cy`, and `cw` are assigned in these statements and further used at line `07`.

Another example of an invalid candidate is the code fragment enclosed by lines `08`–`16`. In this case, extracting the code into a new method would change the behavior of the exiting jump introduced by the `return` statement at line `15`. More specifically, instead of exiting the `mouseReleased` method, it would only exit the new method created by the extraction. Currently, our prototype implementation relies on the preconditions defined by the Extract Method refactoring tool provided by the Eclipse IDE.

[2]Currently, we implemented our approach for Java. However, it is straightforward to adapt it to other object-oriented languages.

3.1.2.2 Quality Preconditions

We also check whether the candidates to extraction follow minimal design quality preconditions. Thus, we propose a size threshold to filter out the extreme cases in which a candidate would result in a poor-quality recommendation because of its small size (e.g., a candidate with a single statement) and also when it is too large (e.g., a candidate encompassing almost all statements of the method).

This size threshold is defined as the minimum number of statements K. More specifically, assuming that C is the set of statements to be extracted and M' are the remaining statements in the method, we check the following condition:

$$|C| \geq K \wedge |M'| \geq K$$

Therefore, a valid candidate must have at least K statements and its extraction must keep the original method with at least K statements. This condition implies that methods with less than $2 \times K$ statements do not produce candidates.

It is important to note that when a certain statement contains child statements, they are also considered in the counting. For example, the `if` statement at line `09` in Figure 3.1 has a size of 8, because it contains 7 children of size 1. On the other hand, the method call at line `18` only count as 1, as it is a single statement with no children.

3.1.3 Computational Complexity

We define the computational complexity of Algorithm 1 in terms of the number of times the $isValid$ function is called. In the worst case scenario, when all statements of the method belongs to a single block, the complexity is $O(n^2)$, where n is the number of statements of the method. The computational complexity of the $isValid$ function itself is hard to determine in practice because its implementation relies on operations provided by the refactoring tools of the underlying IDE. However, we can speculate that the complexity is at least proportional to the size of the method, i.e., $\Omega(n)$. On the other hand, as the generation of candidates for a method is independent from other methods of the system, when we consider the generation of candidates for N methods, the cost would increase linearly.

3.2 Ranking

After the candidates generation step, a scoring function is used to rank the candidates and show the most relevant ones as Extract Method recommendations. The intuition is that Extract Method recommendations should be as independent as possible from the original method, in terms of dependencies, in order to hide an autonomous and well-defined computation. In this section, we describe the kinds of structural dependencies considered for the computation of the function (Section 3.2.1) and how the information extracted from the structural dependencies is used to compute the final score of a candidate (Section 3.2.2).

3.2.1 Structural Dependencies

Statements may read variables, invoke methods, and perform other operations that induce dependencies on other source code entities of the system (or external libraries). This information may be used to characterize and discriminate concerns in the source code. More specifically, considering a selection of statements S, we may compute its set of dependencies, denoted by $Dep(S)$, by extracting the distinct entities referenced in S. Therefore, we represent the structural dependencies of a code fragment using dependency sets. Besides, our approach considers three kinds of dependencies, with distinct levels of granularity: (i) variables, (ii) types, and (iii) packages.

3.2.1.1 Variables

The dependency set of S, considering variables, is denoted by $Dep_v(S)$. If a statement s from a selection of statements S declares, assigns, or reads a variable v, then v is added to $Dep_v(S)$. In a similar way, reads from and writes to formal parameters and fields are also considered when computing $Dep_v(S)$.

For example, the statement at line `03` of Figure 3.1 writes to variable `cx`, which denotes a dependency on `cx`. Furthermore, the statement at line `08` reads from parameter `me`, which denotes a dependency on `me`.

3.2.1.2 Types

The dependency set of S, considering types, is denoted by $Dep_t(S)$. If a statement s from a selection of statements S depends on a type (class or interface) T, then T is added to $Dep_t(S)$. Specifically, the following scenarios denote the dependency on a type:

- If s calls a method m, the type T that declares m is included in $Dep_t(S)$.
- If s reads from or writes to a field f, the type T that declares f is included in $Dep_t(S)$.

- If s creates an object of type T, then T is included in $Dep_t(S)$.
- If s declares a variable v, the type T of v is included in $Dep_t(S)$.
- If s handles an exception of type T, then T is included in $Dep_t(S)$.
- If s casts an object to type T, then T is included in $Dep_t(S)$.
- If s contains a type literal of type T, then T is included in $Dep_t(S)$.

Considering Figure 3.1, the statements at lines `11` and `12` depend on type `FigCompartment`, but for distinct reasons. The former declares a variable of type `FigCompartment`, while the latter invokes method `setEditOnRedraw`, which is defined on type `FigCompartment`.

When handling parameterized types, we include the main type and all its type parameters in $Dep_t(S)$. This rule is applied recursively in the case of nested parameterized types. For example, when we declare a variable of type `Map<String, List<FigCompartment>>`, types `Map`, `String`, `List`, and `FigCompartment` are included in $Dep_t(S)$.

It is worth noting that we may optionally ignore a list of known common types, such as basic types of the language (e.g., `String`, `Object`, etc.). More specifically, our implementing tool is preconfigured with the same filters used in JMove, which ignores types from the core Java API. We assume that these types are not relevant when characterizing the dependencies established by methods, analogous to stop words in natural language processing systems. However, further work is necessary to confirm that assumption, as we discuss in Section 5.3.

3.2.1.3 Packages

The dependency set of S, considering packages, is denoted by $Dep_p(S)$. For each type T included in $Dep_t(S)$, as described in the previous section, the package where T is implemented is included in $Dep_p(S)$. Additionally, all its parent packages are included in $Dep_p(S)$. However, similarly to the list of types to ignore, we may optionally ignore some common root packages, such as `com`, `org`, or `java`, as they do not denote meaningful modules.

Considering Figure 3.1, the statement at line `11` depends on type `FigCompartment`, thus it also depends on `org.argouml.uml.diagram.ui`, which is the package that contains `FigCompartment`. Besides, we also consider that the statement depends on `org.argouml.uml.diagram`, `org.argouml.uml`, and `org.argouml`.

The package `org` is not in this list because it is ignored when using the default settings of our tool.

3.2.2 Computing the Score of Candidates

Let m' be the selection of statements of an Extract Method candidate for method m and m'' the remaining statements in m. We can compute two dependency sets from the dependencies in m' and m'', which we denote by Dep' and Dep'', respectively.

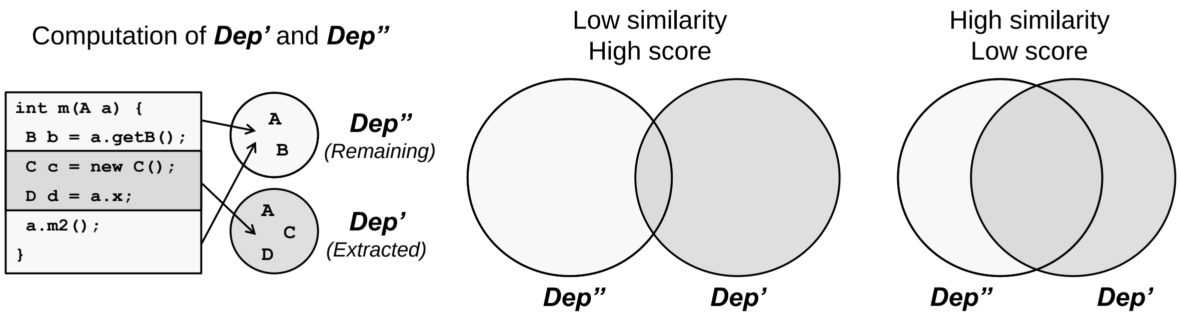


Figure 3.2. Overview of the scoring strategy

The reasoning behind our scoring strategy is that the sets Dep' and Dep'' should be as dissimilar as possible. As illustrated in Figure 3.2, the lower the intersection (similarity) between Dep' and Dep'', the higher the score since there will be a low coupling between the extracted method and the remaining code fragment. In this scenario, we are segregating distinct concerns by extracting the method. On the other hand, the higher the intersection (similarity), the lower the score since there will be a high coupling between the extracted method and the remaining code fragment.

The same rationale can be used to the considered kinds of dependencies: (i) *variables*, where methods should encapsulate their use (and therefore the partial computation they are used for); (ii) *types*, where methods should preferably hide the services provided by them, reducing the impact of changes in their members; and (iii) *packages*, which is analogous to *types*, but at a higher level of abstraction.

3.2.2.1 Distance Between Sets

To compute the score, we need to define the notion of distance between the aforementioned sets Dep' and Dep'', which is defined as:

$$dist(Dep', Dep'') = 1 - \frac{1}{2}\left[\frac{a}{(a+b)} + \frac{a}{(a+c)}\right]$$

where $a = |Dep' \bigcap Dep''|$, $b = |Dep' \setminus Dep''|$, and $c = |Dep'' \setminus Dep'|$.

This distance is based on the Kulczynski set similarity coefficient [Everitt et al., 2011; Sales et al., 2013; Terra et al., 2013a]. Kulczynski measures the similarity between two sets, returning a value between 0 (lowest similarity) and 1 (highest similarity). The Kulczynski distance between two sets is obtained by subtracting the Kulczynski coefficient from 1. Therefore, the highest such distance, the highest the dissimilarity of the sets. Section 3.4 presents an exploratory study we conducted with other five set similarity coefficients in order to decide to use Kulczynski.

3.2.2.2 Scoring Function

Let m' be the selection of statements of an Extract Method candidate for method m and m'' the remaining statements in m. The score of m' is defined as:

$$s(m') = \frac{w_v \times dist({Dep'}_v, {Dep''}_v) + w_t \times dist({Dep'}_t, {Dep''}_t) + w_p \times dist({Dep'}_p, {Dep''}_p)}{w_v + w_t + w_p}$$

where w_v, w_t, and w_p are weight factors for each kind of dependency (variables, types, and packages, respectively). The denominator is used to normalize the score so that the value ranges from 0 to 1.

In the exploratory study conducted in Section 3.4, we investigate the use of weighting schemes that enable/disable each level of dependency. However, unless otherwise noted, when we refer to the scoring function we are using its simplest form, where the weight of each component is equal to 1:

$$s(m') = \frac{1}{3}\Big[dist({Dep'}_v, {Dep''}_v) + dist({Dep'}_t, {Dep''}_t) + dist({Dep'}_p, {Dep''}_p)\Big]$$

In the special case where $Dep' \setminus Dep''$ is empty for all kinds of dependencies (variables, types, and packages) we assign zero to the score of the candidate. The reason for this rule is that there is no merit in extracting code that does not encapsulate any concern. Likewise, the score is also zero when the remaining statements do not encapsulate any concern. In summary, the score of an Extract Method candidate is zero when any of the following conditions hold:

$$|{Dep'}_v \setminus {Dep''}_v| + |{Dep'}_t \setminus {Dep''}_t| + |{Dep'}_p \setminus {Dep''}_p| = 0 \quad \textbf{(condition 1)}$$
$$|{Dep''}_v \setminus {Dep'}_v| + |{Dep''}_t \setminus {Dep'}_t| + |{Dep''}_p \setminus {Dep'}_p| = 0 \quad \textbf{(condition 2)}$$

3.2.3 Filtering the Rank of Candidates

By the exhaustive nature of our candidates generation algorithm, there are usually dozens of candidates for each method. Since users are usually interested on receiving just a few good recommendations, we filter the list of candidates according to the following parameters:

- *Maximum Recommendations per Method:* Limits the number of recommendations for a single method. This parameter is preset to 3 (changeable), which means that only the top three ranked recommendations of each method are considered.
- *Minimum Score Value:* Filters out recommendations with score value lower than a threshold parameter (ranging from 0 to 1). This parameter is preset to 0 (changeable), which means that no recommendation is filtered.

In Chapter 4, we discuss the influence of these parameters on the results of our approach, considering their impact on precision and recall.

3.2.4 Computational Complexity

To compute the dependency sets Dep' and Dep'', we need to perform a single traversal through each statement of the method, which is done in linear time to the number of statements. Besides, we must compute the intersection between these sets, which can also be done in linear time, assuming a $O(1)$ set look-up operation. Finally, the ranking requires sorting the list of candidates, which can be done in $O(n\,lg(n))$ time, where n is the number of candidates. Similarly to the first phase, as the ranking of candidates of a method is independent from other methods of the system, when we consider the generation of candidates for N methods, the cost increases linearly.

3.3 The Approach in Action

This section presents an illustrative example, walking through each step of our recommendation approach applied to a given method. Figure 3.3 illustrates method `execute` from class `LoginAction` of the MyWebMarket system. Throughout this section, we always refer to such method since we aim to demonstrate how our approach provides Extract Method refactoring opportunities for it.

This example method is responsible for handling a request for logging in a user in a Web application. The method uses the Hibernate API to retrieve the user record corresponding to the provided information from a database. When it succeeds, the

```
01 public String execute() throws Exception {
02 1/01 logger.info("Starting execute()");
03 1/02 Session sess = HibernateUtil.getSessionFactory().openSession();
04 1/03 Transaction t = sess.beginTransaction();
05 1/04 Criteria criteria = sess.createCriteria(User.class);
06 1/05 criteria.add(Restrictions.idEq(this.user.getUsername()));
07 1/06 criteria.add(Restrictions.eq("password", this.user.getPassword()));
08 1/07 User user = (User) criteria.uniqueResult();
09 1/08 t.commit();
10 1/09 sess.close();
11      if (user != null) {
12        2/01 ActionContext.getContext().getSession().put(AUTHENTICATED_USER, user);
13 1/10   2/02 logger.info("Finishing execute() -- Success");
14        2/03 return SUCCESS;
15      }
16 1/11 this.addActionError(this.getText("login.failure"));
17 1/12 logger.info("Finishing execute() -- Failure");
18 1/13 return INPUT;
19 }
```

Figure 3.3. An example method extracted from the MyWebMarket system

user session is set up and a success indicator is returned. When the user record is not found, a failure message is displayed and the method returns a failure indicator. The method also logs execution information, using the Log4j API.

3.3.1 Candidates Generation

Our approach first analyzes the method to construct its corresponding model (refer to Section 3.1.1). As we can note by the labeling of the statements in Figure 3.3, two blocks of statements are identified. The first block is the top level block of the method, containing 13 statements (lines `02` to `18`). The second block is enclosed by the `if` statement at line `11`, containing only three statements (lines `12` to `14`).

Next, for each of these two blocks, we iterate through every possible subsequence of statements, checking whether they are valid Extract Method candidates according to the viability and size threshold preconditions (refer to Algorithm 1). Table 3.1 represents each of these candidates, grouping them by their corresponding block (*Block 1* or *Block 2*). Inside a particular block, a cell at row i and column j—which we denote by the tuple (i, j)—corresponds to the Extract Method candidate enclosed by the statements i to j of the block. Therefore, we display in each cell:[3]

- The – symbol when it does not satisfy the size threshold precondition, which in this example we assume to be 3 statements, i.e., $K = 3$.

[3]Cells below the main diagonal are empty because they correspond to cases when $j < i$.

- The × symbol when it does not satisfy the viability precondition, although it satisfies the size threshold precondition.
- The score of the candidate when it is valid.

Table 3.1. Candidates generated from the example method

Block 1													
$i \backslash j$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
1	–	–	×	×	×	×	×	×	0.61	×	–	–	–
2		–	–	×	×	×	×	×	0.81	×	×	–	–
3			–	–	×	×	×	0.63	0.63	×	×	×	–
4				–	–	0.50	0.61	0.54	0.54	×	×	×	0.36
5					–	–	0.53	0.47	0.42	×	×	×	0.30
6						–	–	0.00	0.00	×	×	×	0.24
7							–	–	0.00	×	×	×	0.35
8								–	–	×	×	×	0.42
9									–	×	×	×	0.49
10										×	×	×	0.61
11											–	–	0.46
12												–	–
13													–

Block 2			
$i \backslash j$	**1**	**2**	**3**
1	–	–	0.48
2		–	–
3			–

From 91 distinct (i, j) tuples in Block 1, 28 do not satisfy the size threshold precondition (i.e., contain less than 3 statements) and 41 do not satisfy the viability preconditions (i.e., causes syntactical errors or behavior changes). As an example, tuple $(3, 7)$ is not viable since two variables (`t` and `user`) are declared inside the extracted code fragment and further used in the code. As another example, tuple $(10, 10)$ is also not viable since by extracting a code fragment with a `return` statement inside a conditional the behavior of the program would change. In summary, 22 out of the 91 tuples from Block 1 are valid candidates.

In Block 2, on the other hand, only 1 out of the 6 possible tuples is a valid candidate, the other 5 tuples are discarded due to the size threshold. In total, this phase yields a list of 23 candidates to be ranked in the second phase of the approach.

3.3.2 Ranking

Next, we need to compute the score for each candidate identified at the first phase. Therefore, we need to identify the dependencies present in the source code and build the sets *Dep*′ and *Dep*″, as described in Section 3.2.2.

In order to illustrate the computation of the scoring function, we will consider Extract Method candidate `1/02–09`, i.e., the extraction of the statements 2 to 9 of *Block 1*. We will focus on the first extracted statement (`1/02`). It declares variable `sess`, which is added to Dep'_v. The type of variable `sess` is `Session`, which is added to Dep'_t. Besides, there are two method invocations (`getSessionFactory` and `openSession`) which triggers the addition of their declaring types (`HibernateUtil` and `SessionFactory`, respectively) to Dep'_t. By adding types `Session` and `SessionFactory`, we also add their package `org.hibernate` to Dep''_p. Although package `org.hibernate` has a parent package `org`, it is disregarded due to the ignore list of common root packages. Last, package `classes` of type `HibernateUtil` is added to Dep''_p.

The sets Dep'_v, Dep'_t, and Dep'_p are built by repeating this procedure for every statement to be extracted. Similarly, the sets Dep''_v, Dep''_t, and Dep''_p are built by repeating this procedure for the remaining statements. Table 3.2 presents the dependencies identified for candidate `1/02–09`. The left column lists the dependencies only present in the remaining statements ($Dep'' \setminus Dep'$), while the right column lists the dependencies only present in the extracted statements ($Dep' \setminus Dep''$). The center column contains the common dependencies, found both in the extracted and remaining statements ($Dep' \bigcap Dep''$).

Table 3.2. Dependency sets computed for the example candidate

Variables		
$c = \lvert Dep''_v \setminus Dep'_v \rvert = 4$	$a = \lvert Dep'_v \bigcap Dep''_v \rvert = 1$	$b = \lvert Dep'_v \setminus Dep''_v \rvert = 4$
`INPUT`	`user`	`criteria`
`AUTHENTICATED_USER`		`sess`
`SUCCESS`		`t`
`logger`		`this.user`

Types		
$c = \lvert Dep''_t \setminus Dep'_t \rvert = 6$	$a = \lvert Dep'_t \bigcap Dep''_t \rvert = 1$	$b = \lvert Dep'_t \setminus Dep''_t \rvert = 7$
`SystemConstants`	`LoginAction`	`HibernateUtil`
`Action`		`User`
`ActionContext`		`Criteria`
`ActionSupport`		`Session`
`Map`		`SessionFactory`
`Category`		`Transaction`
		`Restrictions`

Packages		
$c = \lvert Dep''_p \setminus Dep'_p \rvert = 5$	$a = \lvert Dep'_p \bigcap Dep''_p \rvert = 1$	$b = \lvert Dep'_p \setminus Dep''_p \rvert = 2$
`com.opensymphony`	`classes`	`org.hibernate`
`com.opensymphony.xwork2`		`org.hibernate.criterion`
`java.util`		
`org.apache`		
`org.apache.log4j`		

The values from Table 3.2 allow us to compute the distance between the dependency sets, using the formula presented at Section 3.2.2.1:

$$
\begin{aligned}
dist(Dep'_v, Dep''_v) &= 1 - \frac{1}{2}\Big[\frac{1}{(1+4)} + \frac{1}{(1+4)}\Big] = 0.8 \\
dist(Dep'_t, Dep''_t) &= 1 - \frac{1}{2}\Big[\frac{1}{(1+7)} + \frac{1}{(1+6)}\Big] = 0.866 \\
dist(Dep'_p, Dep''_p) &= 1 - \frac{1}{2}\Big[\frac{1}{(1+2)} + \frac{1}{(1+5)}\Big] = 0.75
\end{aligned}
$$

Thus, the score is calculated by replacing the distance values computed above in the score equation presented in Section 3.2.2.2. In this case, the score value for candidate `1/02–09` is:

$$s(m') = \frac{1}{3}\Big[0.8 + 0.866 + 0.75\Big] = 0.805$$

The score values calculated for all 23 candidates are presented in Table 3.1. Assuming parameter *Maximum Recommendations per Method* as 3 and *Minimum Score Value* as 0, the final list of recommendations is:

1. Extract statements `1/02–09` (score of 0.805)

2. Extract statements `1/03–09` (score of 0.63)

3. Extract statements `1/03–08` (score of 0.63)

The higher score of the first recommendation (0.805) is due the fact that such Extract Method candidate encapsulates all the code responsible for the database operation, isolating variables, types, and packages related to module Hibernate. In contrast, the second recommendation (score of 0.63) does not include statement `1/02` and therefore fails to encapsulate the entire database operation concern, i.e., entities such as variable `sess`, type `Session`, and package `org.hibernate` are not referenced only by the code fragment suggested to be extracted.

3.4 Exploratory Study

The final design of our solution required a preliminary evaluation in order to calibrate alternative strategies to rank the refactoring opportunities. In this section, we report an exploratory study—wich was conducted in a controlled environment—to address the following overarching research questions:

RQ #1 – How many known Extract Method instances can be found by the proposed approach?

RQ #2 – What is the best set similarity coefficient and dependency set strategy to rank Extract Method candidates?

RQ #3 – What is the impact of the minimal number of statements threshold on the provided Extract Method recommendations?

RQ #4 – What is the precision of the proposed approach?

3.4.1 Target System

This first study relies on a simple web-based e-commerce system, called MyWebMarket, which includes functions to manage customers and products, handle purchase orders, generate reports, etc. This system was implemented two years ago by Terra et al. [2012] to evaluate refactoring recommendations to repair software architectural violations. Despite its small size—1,016 LOC and 116 methods—MyWebMarket was carefully designed to resemble on a smaller scale the architecture of a large real-world human resource management system [Terra and Valente, 2009].[4]

MyWebMarket was first implemented as a monolithic system (version used in this study) and has evolved to more modularized versions. More important, we identified 25 Extract Method instances that were applied over the system evolution, which we used in the study setup.

3.4.2 Study Setup

To evaluate the recommendations triggered by our tool, we rely on an oracle, which is a set of well-known Extract Method instances. A recommendation is considered relevant when the oracle contains it, otherwise it is not relevant. Therefore, given a set of recommendations R and a set of known Extract Method instances O, the recall and precision of our approach is computed as follows:

$$recall = \frac{|R \bigcap O|}{|O|}$$
$$precision = \frac{|R \bigcap O|}{|R|}$$

[4]MyWebMarket source is publicly available at: `http://aserg.labsoft.dcc.ufmg.br/jextract`

Our oracle contains 25 well-known Extract Method opportunities, as previously mentioned. We define an Extract Method opportunity as a sequence of statements to be extracted into a new method. It is worth noting that we only consider equal recommendations that suggest the extraction of exactly the same sequence of statements.

We used the minimal size threshold of 3 statements (i.e., $K = 3$) and the canonical formula for the ranking function described in Section 3.2.2.2 (i.e., using a uniform weighting scheme and the Kulczynski coefficient) to run this experiment. Besides, we ignored dependencies on types of packages `java.lang` and `java.util`. Similarly, we ignore dependencies on the root packages `com`, `org`, `java` and `javax`.

In particular cases, the results are referred as Top n recall or precision, which means that we are computing the recall or precision using n as the parameter *Maximum Recommendations per Method*, introduced in Section 3.2.3.

3.4.3 RQ #1: How many known Extract Method instances can be found by the proposed approach?

This research question investigates if our approach is able to detect the known Extract Method instances, regardless of their positions in the rank. More specifically, we check whether such instances are identified as valid candidates in the first phase of the approach (refer to Section 3.1).

Our approach was able to detect 14 Extract Method instances (56%) as valid candidates. The remaining 11 instances require some kind of code transformation to attend the preconditions of the candidates generation phase. Most of them (10) require the extraction of non-contiguous code. As an example, Figure 3.4 presents an Extract Method instance that our approach did not detect as a valid candidate. Since such method should implement only functional concerns, the persistence-related code (lines 3–4 and 12–14) should be extracted into a new method. However, this code fragment is non-contiguous. As a matter of fact, if we move lines 3–4 to after line 11 (which preserves behavior in this case), our approach would suggest the desired Extract Method recommendation. Likewise, all failing instances fit in this pattern, except for one instance in which the repetition of some statements were required prior to the method extraction.

In the following research questions, we considered only the 14 contiguous Extract Method instances since it would be pointless to measure the effectiveness of the ranking when the expected candidate is not even returned from the candidates generation phase.

```
 1 public String save() throws Exception {
 2   logger.info( "Starting save()" );
 3   Session s= HibernateUtil.getSessionFactory().openSession();
 4   Transaction t = sess.beginTransaction();
 5
 6   purchaseOrder.setOrderDate(new Date());
 7   for (PurchaseOrderItem item :
 8     purchaseOrder.getPurchaseOrderItems()){
 9     item.setPurchaseOrder(this.purchaseOrder);
10   }
11
12   s.save(this.purchaseOrder);
13   t.commit();
14   s.close();
15
16   this.task = SystemConstants.UD_MODE;
17   ....
18 }
```

Figure 3.4. A non-contiguous Extract Method instance (lines 3–4 and 12–14)

3.4.4 RQ #2: What is the best set similarity coefficient and dependency set strategy to rank Extract Method candidates?

This research question investigates different set similarity coefficients to compute the distance between sets of dependencies, as presented in Table 3.3. To measure the similarity between *Dep*′ and *Dep*″, the considered coefficients rely on variables a (the number of dependencies on both sets), b (the number of dependencies on *Dep*′ only), and c (the number of dependencies on *Dep*″ only), as described in Section 3.2.2.1.

Table 3.3. General Purpose Similarity Coefficients

Coefficient	Definition	Range
a. Jaccard	$a/(a+b+c)$	0–1*
b. Sorenson	$2a/(2a+b+c)$	0–1*
c. Sokal and Sneath 2	$a/[a+2(b+c)]$	0–1*
d. PSC	$a^2/[(b+a)(c+a)]$	0–1*
e. Kulczynski	$\frac{1}{2}[a/(a+b)+a/(a+c)]$	0–1*
f. Ochiai	$a/[(a+b)(a+c)]^{\frac{1}{2}}$	0–1*

The symbol "∗" denotes the maximum similarity.

The six evaluated similarity coefficients were drawn from an initial list of 18 coefficients studied in a previous work of Terra et al. [2013a]. We have not included the other 12 coefficients because they require an additional parameter, which is not applicable in our context. Specifically, they required a parameter d, which corresponds to the size of the set of all entities on neither *Dep*$'$ nor *Dep*$''$. However, since we only consider the dependencies in a method, they are always on either *Dep*$'$ or *Dep*$''$ (or both).

Additionally, we also investigated the impact of excluding some kinds of dependencies in our scoring function by assigning a value of zero for one or more of the weight factors w_v, w_t, or w_p (refer to Section 3.2.2.2). For example, $w_v = 0$ excludes variable dependencies information.

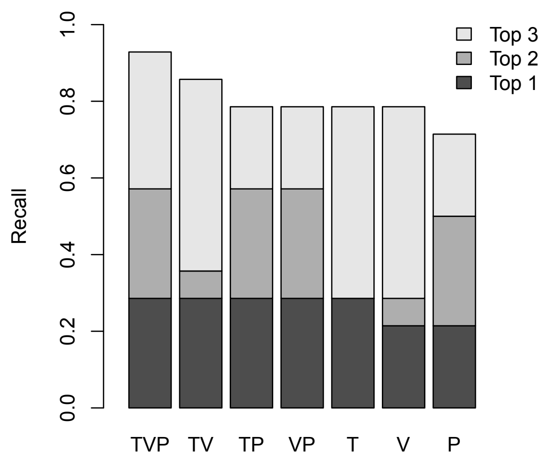
For each set similarity coefficient reported in Table 3.3, we tested the recommendations produced by each possible combination of dependency sets, as presented in Figure 3.5. In this figure, `V` stands for *Variables*, `T` stands for *Types* , and `P` for *Packages*. For example, `TV` stands for a compound score of *Types* and *Variables* dependency sets, but excluding *Packages*.

Figure 3.5 reports the Top n recall for the tested combinations of similarity coefficients and dependency sets, regarding only the 14 Extract Method instances respecting the proposed preconditions (as discussed in the answer to RQ #1). Recall, in this case, is defined as the percentage of instances covered by the first n recommendations triggered for each method in the system. For example, using *Jaccard* and only *Type* dependencies we were able to cover 4 (28.6%) Extract Method instances relying solely on the first recommendation (Top 1). Thus, the results reported in Figure 3.5 supports three central findings:

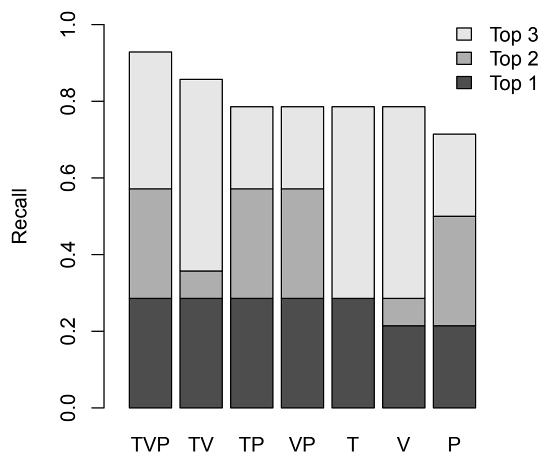
- There are two well-defined groups of similarity coefficients: (i) ineffective coefficients—namely *Jaccard*, *Sorenson*, and *Sokal and Sneath 2*—that, independently of the considered dependency sets, achieved the maximum Top 1 recall of 28.7%; and (ii) effective coefficients—namely *Ochiai*, *PSC*, and *Kulczynski*—that achieved Top 1 recall values above 80%. It is worth noting that, interestingly, *Sokal and Sneath 2* was the most effective coefficient in the study of Sales et al. [2013].

- If we consider a single strategy to compute dependency sets (i.e., just `T`, just `V`, or just `P`), we can infer that dependency sets considering just *Types* (`T`) are more effective.

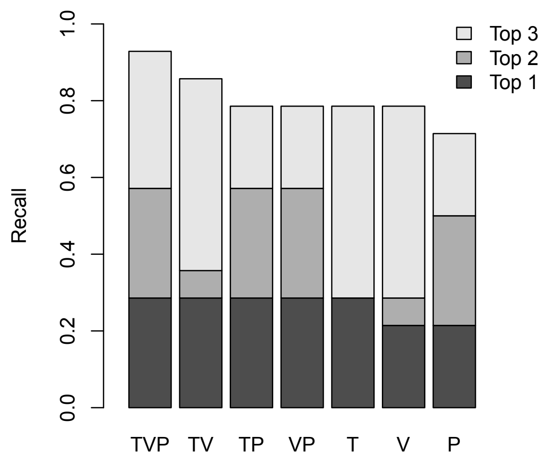
- Combining dependency sets can slightly improve the results. For example, when we use the `TV` strategy (i.e., dependency sets for types and variables), Top 1 recall is always greater than or equal to the ones achieved using just `T` or `V`. Moreover,



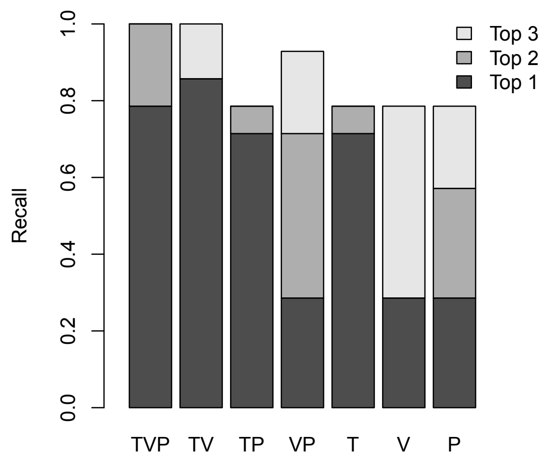
(a) Jaccard



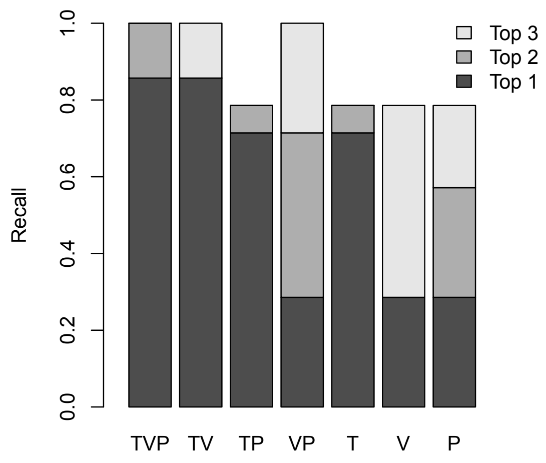
(b) Sorenson



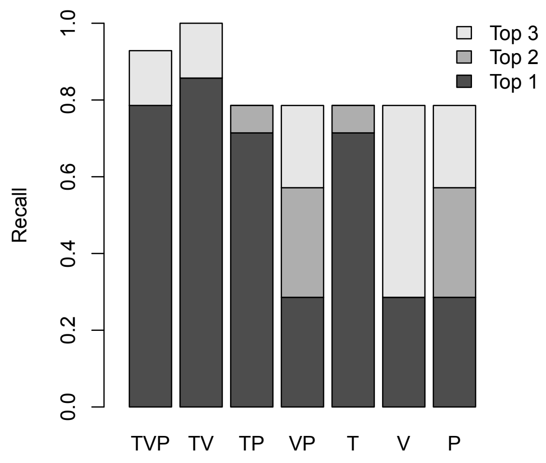
(c) Sokal and Sneath 2



(d) PSC



(e) Kulczynski



(f) Ochiai

Figure 3.5. Top n recall using different scoring strategies (MyWebMarket)

when we compare `TVP` against `TV`, Top 2 and Top 3 recall are higher in five of the six coefficients (*Ochiai* is the exception).

In summary, the combination of the *Kulczynski* coefficient and the `TVP` strategy achieved the maximal recall values. Using such combination, we could provide correct recommendations for 12 out of 14 Extract Method instances (85.7%) using only the first recommendation (Top 1). Moreover, the recall increases to 100% on Top 2. Although other configurations were only slightly inferior, we decided to rely on the *Kulczynski* coefficient and the `TVP` configuration in our approach. In Chapter 4, we further investigate ranking strategies on a larger dataset.

3.4.5 RQ #3: What is the impact of the minimal number of statements threshold on the provided Extract Method recommendations?

The results from RQ #2 supported the choice of the *Kulczynski* coefficient and the `TVP` dependency sets for our approach. This research question, on the other hand, investigates the impact of the minimal threshold parameter (K) on the recall. Figure 3.6 shows the Top n recall values with the minimal thresholds ranging from 1 to 6. As can be inferred, the optimal choice for the minimal threshold is indeed 3 statements, which was the value used when investigating the RQ #2. On one hand, when using values higher than 3, the Top 2 recall decreases from 100% (K=3) to 22% (K=6). On the other hand, when using values lower than 3, our approach detects many spurious results that achieves a high score but are meaningless due to their size (e.g., one or two statements).

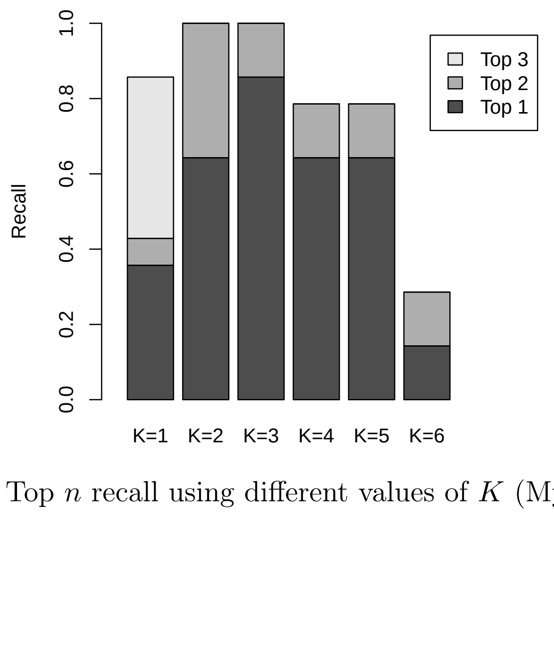


Figure 3.6. Top n recall using different values of K (MyWebMarket)

3.4.6 RQ #4: What is the precision of the proposed approach?

In the previous research questions, we rely on recall to guide central design decisions in our approach (RQ #2) and evaluate the relevance of the minimal size threshold parameter (RQ #3). However, as usual, recall measures should be complemented by precision measures. Regarding precision, it is worth noting that our approach generates a large number of candidates in the first phase. Specifically for MyWebMarket, we generate 951 valid candidates, which correspond to 35.2 candidates/method on average. For that reason, the approach heavily depends on the ranking phase to filter out non-relevant candidates. Therefore, parameters *Maximum Recommendations per Method* and *Minimum Score Value* (described in Section 3.2.3) influence the precision of the results. Therefore, this research question investigates the impact of the aforementioned parameters on the precision of our approach.

Figure 3.7 shows the overall precision and recall of our approach for variations of *Minimum Score Value*, i.e., we removed from the rank all recommendations with a ranking score less than a given threshold (x-axis). Moreover, we did not restrict the number of recommendations per method (i.e., *Maximum Recommendations per Method* $= \infty$). The results indicate that our overall precision is very low, usually less than 20%, when we set up a minimum threshold below 0.8. However, when we set up a minimum score threshold above 0.8, recall faces a significant decrease, falling below 30%. In fact, Figure 3.7 shows that there is no single threshold alone that provides high precision without sacrificing recall or vice-versa. This fact is mainly due to the existence of a large number of similar recommendations (with also similar scores) for the same method on the rank.

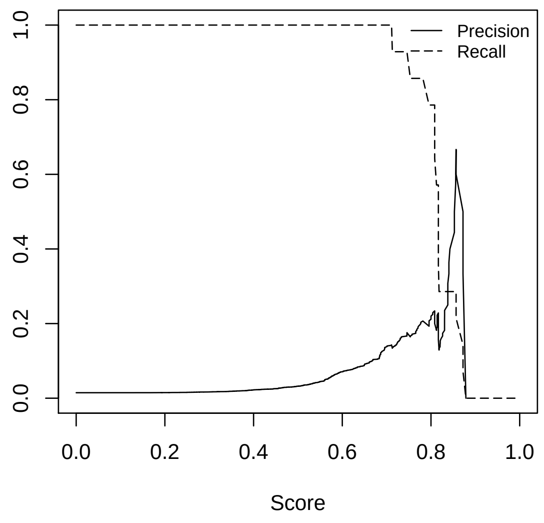


Figure 3.7. Precision and recall by *Minimum Score Value* (MyWebMarket)

The aforementioned finding leads us to limit the number of recommendations for a single method (parameter *Maximum Recommendations per Method*). Figure 3.8a shows the precision-recall curves when we consider only the Top n recommendations of each method. In other words, the curves show how precision and recall vary as the maintainer proceeds with the examination of the recommendations in our ranking, starting from the top-ranked recommendation. A relevant observation is that the Top 1 strategy can achieve an overall precision of 50.0%, while preserving a high recall (85.7%).

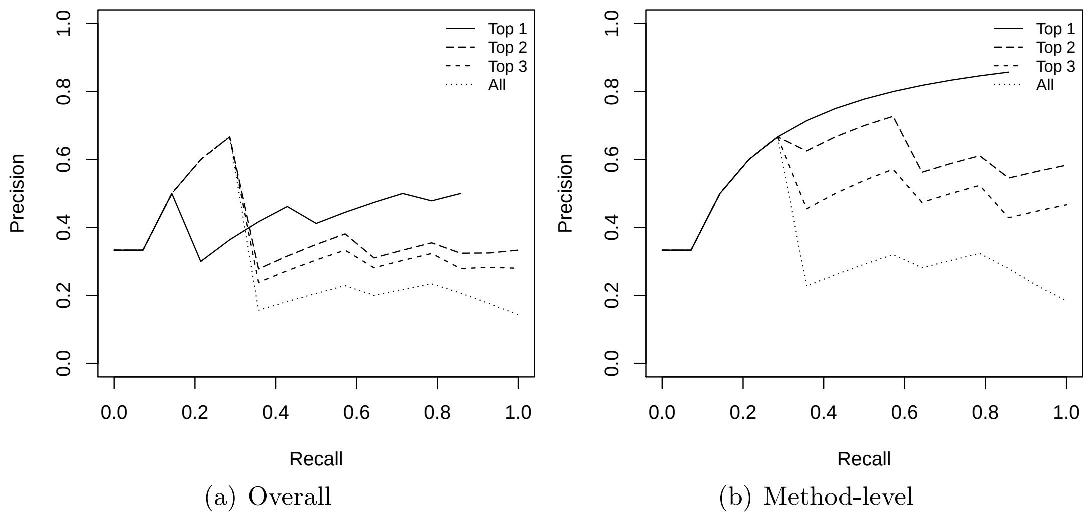


Figure 3.8. Precision vs. Recall (MyWebMarket)

Moreover, we also computed a second type of precision, considering just the recommendations related to methods that have a valid Extract Method instance in the oracle. This precision, which we referred to as method-level precision, would correspond the scenario where developers initially manifest interest on receiving automated Extract Method refactoring recommendations for a particular method they are maintaining or trying to comprehend. In this scenario, as presented in Figure 3.8b, we could achieve a precision of 85.7% and a recall of 85.7%.[5]

On the other hand, the Top 2 and Top 3 strategies trade precision for a higher recall. More specifically, the precision decreases even when considering two recommendations per method (from 85.7% for Top 1 to 58% for Top 2). However, it is possible to achieve a recall of 100% with two recommendations per method.

In summary, this research question provides the following insights:

[5]By construction, considering one recommendation per valid method, precision will be equal to recall, when evaluating the whole set of recommendations

- On one hand, it is not feasible to present all potential candidates, since our approach would generate a massive number of recommendations, which causes very low precision rates.
- On the other hand, it is better to present only one recommendation per method. We claim that the proposed score function is most effective when used to rank recommendations confined to the scope of methods. In practical terms, a supporting tool for our approach should suggest few recommendations per method, preferably just only the best one.

3.4.7 Threats to Validity

We must state at least one major threat to the external validity of the reported study. Since we considered only a small web-based system, we cannot claim that our approach will provide equivalent results in other systems (as usual in empirical studies in software engineering). On the other hand, MyWebMarket was carefully implemented and designed to resemble on a smaller scale the architecture of a large-scale and long-lived real-world human resource management system [Terra and Valente, 2009]. Besides, this is only a preliminary case study. Chapter 4 reports an evaluation with other Java systems of different domains.

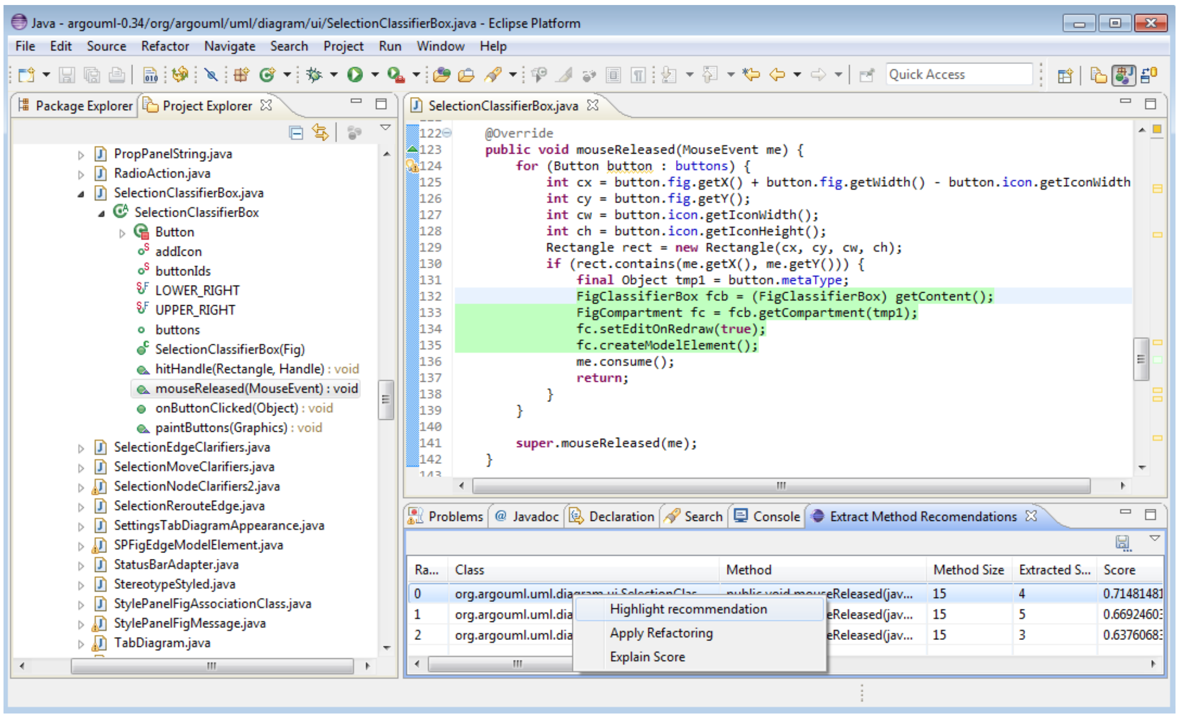
3.5 Tool Support

This section presents JExtract, the tool we built to implement the proposed Extract Method refactoring recommendation approach. More specifically, we discuss its basic functionality, configuration, and internal architecture.[6]

3.5.1 Functionality and User Interface

JExtract is a plug-in for the Eclipse IDE that allows developers to request Extract Method recommendations for a particular method. Figure 3.9 shows a screenshot of the Eclipse IDE workbench with JExtract plug-in. We can observe the *Extract Method Recommendations* view at the bottom, displaying a list of recommendations. In this case, we triggered the *Find Extract Method Opportunities* action for method `mouseReleased`, previously presented in Figure 3.1. The best candidate for the method is highlighted (lines 132–135), which corresponds to the extraction of statements `3/02`–`05`, achieving a score of 0.7148.

[6]JExtract is publicly available at: `http://aserg.labsoft.dcc.ufmg.br/jextract`


Java - argouml-0.34/org/argouml/uml/diagram/ui/SelectionClassifierBox.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
Project Explorer
PropPanelString.java
RadioAction.java
SelectionClassifierBox.java
SelectionClassifierBox
Button
addIcon
buttonIds
LOWER_RIGHT
UPPER_RIGHT
buttons
SelectionClassifierBox(Fig)
hitHandle(Rectangle, Handle) : void
mouseReleased(MouseEvent) : void
onButtonClicked(Object) : void
paintButtons(Graphics) : void
SelectionEdgeClarifiers.java
SelectionMoveClarifiers.java
SelectionNodeClarifiers2.java
SelectionRerouteEdge.java
SettingsTabDiagramAppearance.java
SPFigEdgeModelElement.java
StatusBarAdapter.java
StereotypeStyled.java
StylePanelFigAssociationClass.java
StylePanelFigMessage.java
TabDiagram.java
SelectionClassifierBox.java
Problems
Javadoc
Declaration
Search
Console
Extract Method Recomendations
Highlight recommendation
Apply Refactoring
Explain Score

```
    @Override
    public void mouseReleased(MouseEvent me) {
        for (Button button : buttons) {
            int cx = button.fig.getX() + button.fig.getWidth() - button.icon.getIconWidth
            int cy = button.fig.getY();
            int cw = button.icon.getIconWidth();
            int ch = button.icon.getIconHeight();
            Rectangle rect = new Rectangle(cx, cy, cw, ch);
            if (rect.contains(me.getX(), me.getY())) {
                final Object tmp1 = button.metaType;
                FigClassifierBox fcb = (FigClassifierBox) getContent();
                FigCompartment fc = fcb.getCompartment(tmp1);
                fc.setEditOnRedraw(true);
                fc.createModelElement();
                me.consume();
                return;
            }
        }

        super.mouseReleased(me);
    }
```

Ra...	Class	Method	Method Size	Extracted S...	Score
0	org.argouml.uml.diagram.ui.SelectionClas	public void mouseReleased(jav...	15	4	0.71481481
1	org.argouml.uml.dia	eReleased(jav...	15	5	0.6692460
2	org.argouml.uml.dia	eReleased(jav...	15	3	0.6376068

Figure 3.9. Screenshot of JExtract Plug-in

Besides inspecting the recommendations, the tool enables developers to apply the operation automatically, taking advantage of the underlying Eclipse refactoring support. In this case, the Extract Method refactoring wizard is opened, where developers can fill the name of the new method and preview the change.

Before requesting recommendations, developers may change the following parameters of the tool:

- *Minimum Statements:* Minimum number of statements that can be extracted and also the minimum number of statements that can be left in the original method. It corresponds to the size threshold K described in Section 3.1.2.2.

- *Maximum Recommendations per Method:* Maximum number of recommendations that can be given for a single method, as described in Section 3.2.3.

- *Minimum Score Value:* Score threshold to filter recommendations, as described in Section 3.2.3.

- *Ignore Type List:* List of types that should be ignored when extracting dependencies. Values are specified as a comma separated list of qualified type names. A name prefix followed by the wildcard character ($*$) is also supported. This parameter is preset to `"java.lang.*,java.util.*"` (changeable) to ignore types from the core Java API.

- *Ignore Package List:* List of packages that should be ignored when extracting dependencies. Values are specified as a comma separated list of qualified package names. A name prefix followed by the wildcard character (`*`) is also supported. This parameter is preset to `"java,javax,com,org,br"` (changeable), to ignore common root package names that are not meaningful modules.

3.5.2 Architecture

We designed JExtract centered on the plug-in architecture provided by the Eclipse IDE. We also take advantage of the available native infrastructure, such as the Java Development Tools (JDT) and Language Toolkit (LTK).

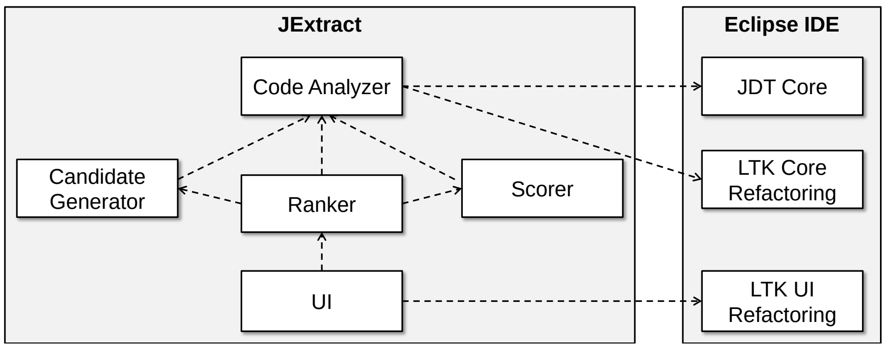


Figure 3.10. JExtract's architecture

Figure 3.10 illustrates the architecture of the current implementation of JExtract. Internally, JExtract is subdivided in the following five main modules:

1. *Code Analyzer*: This module provides the following services to other modules: (a) it builds the structure of block and statements (refer to Subsection 3.1.1); (b) it extracts the structural dependencies (refer to Subsection 3.2.1); and (c) it checks if an Extract Method candidate satisfies the underlying Eclipse Extract Method refactoring preconditions. In fact, this module contains most communication between JExtract and Eclipse APIs (e.g., `org.eclipse.jdt.core` and `org.eclipse.ltk.core.refactoring`).

2. *Candidate Generator*: This module generates all Extract Method candidates based on Algorithm 1 and hence depends on service (a) of module *Code Analyzer*.

3. *Scorer*: This module calculates the score of the Extract Method candidates generated by module *Candidate Generator* (refer to Subsection 3.2.2) and hence depends on service (b) of module *Code Analyzer*.

4. *Ranker*: This module is responsible for sorting and filtering the list of Extract Method candidates generated by module *Candidate Generator* and scored by module *Scorer*. It depends on service (c) of module *Code Analyzer* to filter candidates that do not satisfy preconditions.

5. *UI*: This module consists of the front-end of the tool, which relies on the Eclipse UI API (`org.eclipse.ui`) to implement two menu extensions, six actions, and one main view. Moreover, it depends on module UI from LTK (`org.eclipse.ltk.ui.refactoring`) to delegate the refactoring appliance to the underlying Eclipse Extract Method refactoring tool.

3.6 Final Remarks

This chapter presented our approach to identify and recommend Extract Method refactoring opportunities. The proposed approach consists of two phases: first it identifies code fragments viable for extraction, which are later ranked. We rely on the ranking to select the best refactoring opportunities, centered on the design principle of separation of concerns. Next, we presented an illustrative example of the application of our approach, demonstrating that it was able to isolate a code fragment related to a database operation from its original method.

We also reported an initial exploratory study that guided the design of our approach. We compared 6 similarity coefficients, which led the decision to use the Kulczynski coefficient, although 3 of the 6 coefficients showed to be effective. Besides, we explored a number of possible weighting schemes to identify which of the three components of the scoring function (variables, types, and packages) is more relevant to rank the candidates. This experiment led to the decision of using the same weight for each component.

Last, we presented JExtract, an Eclipse plug-in that implements our approach, describing its design and basic functionalities. Our tool allows developers to request Extract Method recommendations, which can be inspected and automatically applied. Moreover, we described parameter settings that can be used to control the results of the tool.

In the next chapter, we present the evaluation of our approach. In a first study, we compare our approach with JDeodorant, a state-of-the-art recommendation system, using three systems. In a second study, we evaluate our approach in 13 open-source systems using a synthesized dataset. We also return to some of the research questions from the exploratory study to investigate whether the same observations can be made on other systems.

Chapter 4

Evaluation

In Chapter 3, we presented our Extract Method refactoring recommendation approach and reported an initial exploratory study to calibrate our technique. In this chapter, we evaluate the proposed approach by investigating two additional research questions:

RQ #5 – How does our approach perform when compared to state-of-the-art ones?

RQ #6 – How does our approach perform when evaluated with other systems?

To answer ***RQ #5***, we compare the results of our tool with JDeodorant and discuss their differences (Section 4.1). Besides MyWebMarket, we included two other systems in this comparison (JUnit and JHotDraw). To answer ***RQ #6***, we present a quantitative study using a set of 1,182 well-known Extract Method instances from 13 open-source systems (Section 4.2). Last, we conclude this chapter with final remarks (Section 4.3).

4.1 Study 1: Comparative Evaluation

In this study, we investigate how our approach performs in comparison with existing ones, focusing on recall and precision. In order to address this question, we conducted a study with three systems: MyWebMarket, the system we previously used in the exploratory study, and two well-known open-source systems, JUnit and JHotDraw. Unlike MyWebMarket, we did not have an oracle of Extract Method instances for JUnit and JHotDraw. Therefore, we adopted a strategy to synthesize such oracle by applying *Inline Method* refactoring operations. The remainder of this section describes the study design (Section 4.1.1) and discusses the results (Section 4.1.2).

4.1.1 Study Design

4.1.1.1 Selected Tools for Comparison

Even though the large number of approaches in the literature related to code extraction, we compare our approach with those that satisfy the following requirements:

- *Fully Automated:* The approach should provide Extract Method refactoring suggestions with no user input or interaction.
- *Tool Availability:* There must be a publicly available supporting tool.
- *Java Support:* Since our evaluation relies on Java systems, the tool must support such language.

We only considered JDeodorant in this study because it is the only tool that fulfills such requirements.

4.1.1.2 Target Systems

We included three systems in this study: MyWebMarket, JUnit (version 3.8), and JHotDraw (version 5.2). In order to measure recall and precision, we check the suggestions reported by each tool against an oracle, which includes all relevant *Extract Method* opportunities for a given method. For this reason, we considered again MyWebMarket to take advantage of the existing oracle. On the other hand, for JUnit and JHotDraw, we synthesized the oracles, as detailed in the next section.

It is worth noting that we specially chose JUnit and JHotDraw for a specific reason. Since they have been designed and implemented by expert developers (especially in the earlier versions we selected), we may assume, with relative certainty, that there is no relevant *Extract Method* instances in these systems. This property is important because existing *Extract Method* instances would make our oracle incomplete.

4.1.1.3 Oracle Construction

To construct an oracle for JUnit and JHotDraw, we judiciously applied *Inline Method* refactoring operations in order to create system's versions with well-known *Extract Method* instances. This strategy is based on the assumption that *when a certain method m invokes m' and the computation performed by m' is inlined into m, an Extract Method opportunity is introduced in m.* In other words, we assume that the inlined code fragment is expected to be found by tools that identify *Extract Method* refactoring opportunities.

We followed these steps to construct the oracle:

1. We retrieved all methods of the target systems with more than three statements (*minimum size threshold* K, as recommended by the exploratory study described in Section 3.4).

2. For each method m retrieved in Step 1, we retrieved all methods it invokes. Each of these methods, which we denote by m', should satisfy the following preconditions:

   - The size of m' must be at least three statements, i.e., $|m'| \geq 3$, in conformance to the *minimum size threshold* value. Moreover, this rule avoids methods that do not contain complex logic (e.g., getter, setter, and delegate methods).
   - The size ratio between both methods (i.e., $|m'|/|m|$) should lie in the range $[0.1, 2]$. This rule avoids some extreme cases where the inlined method is very small ($|m'|/|m| < 1/10$) or very large ($|m'|/|m| > 2$) when compared to the invoker. Thereupon, we increase the chance of significant Extract Method refactoring opportunities.
   - The *Inline Method* preconditions of the IDE refactoring tool must be respected when inlining the invocation of m'.
   - Method m' cannot be one of the methods previously modified by this process, i.e., if a method m_1 is inlined into a method m_2, then m_2 cannot be inlined into another method.

3. From the list of method invocations computed at Step 2, we selected a single invocation to apply an *Inline Method* refactoring. We gave precedence to methods implemented in other classes and, as a second criterion, to larger methods.

4. The selected invocation was inlined using the IDE refactoring tool and the corresponding Extract Method instance (that reverts the inline) is registered in the oracle.

Table 4.1 presents the number of Extract Method instances in our oracle (25 for JUnit and 56 for JHotDraw). As can be observed, we generated valid instances for 26.0% and 25.2% of candidate methods in JUnit and JHotDraw, respectively (i.e., methods with at least six statements, assuming the minimum size threshold $K = 3$).

Table 4.1. Study 1: Target systems

System	Oracle size	Total methods	$\geq 2{\times}K$
JUnit 3.8	25 (26.0%)	470	96
JHotDraw 5.2	56 (25.2%)	1,478	222

Method size must be at least $2 \times K$ to produce candidates.

4.1.1.4 JExtract Setup

In this study, we used the default settings of JExtract (as presented in Table 4.2), except for parameter *Maximum Recommendations per Method*, for which we compared three different settings, namely Top 1, Top 2, and Top 3. Top-n means that in each valid candidate method JExtract triggers n recommendations at maximum.

Table 4.2. Study 1: JExtract Settings

Parameter	Value
Minimum Statements	3
Maximum Recommendations per Method	1, 2, and 3
Minimum Score Value	0.0
Ignore Type List	`"java.lang.*,java.util.*"`
Ignore Package List	`"java,javax,com,org,br"`

4.1.1.5 JDeodorant Setup

In this study, we used the default settings of JDeodorant, except for the following parameters:

- *Minimum Number of Slice Statements*, which was set to 3 to be consistent with the setup of our approach (*minimum size threshold* K).
- *Minimum Number of Statements in Method*, which was set to 6 because, when $K = 3$, our approach does not generate suggestions for methods with less than $2 \times K$ statements.
- *Maximum Number of Duplicated Statements*, which was set to 0 because: (i) we knew in advance that our oracle does not include any suggestion that involves duplicating statements and (ii) our approach does not suggest duplicating statements.

It is important to mention that, in accordance to JDeodorant's documentation, we attached the source code of all APIs the target systems depend on. For instance, we even included the source code of Java API.

4.1.2 Results and Discussion

We ran both tools on the three aforementioned systems to collect their recommendations. Table 4.3 reports recall and precision values achieved by JExtract (separated into three configurations: Top 1, Top 2, and Top 3) and JDeodorant. In the first and second columns of the table, we display the system and the number of Extract Method instances in the oracle, respectively. In the other columns, we repeat for each tool: the number of relevant recommendations found (#), recall (Rc.), and precision (Pr.).

Table 4.3. Study 1: Comparing JExtract and JDeodorant

		JExtract									JDeodorant		
		Top 1			Top 2			Top 3					
System	#	#	Rc.	Pr.	#	Rc.	Pr.	#	Rc.	Pr.	#	Rc.	Pr.
JHotDraw 5.2	56	19	0.339	0.339	26	0.464	0.236	32	0.571	0.198	2	0.036	0.045
JUnit 3.8	25	13	0.520	0.520	16	0.640	0.327	18	0.720	0.250	0	0.000	0.000
MyWebMarket	14	12	0.857	0.857	14	1.000	0.500	14	1.000	0.333	2	0.143	0.333
Overall	95	44	0.463	0.463	56	0.589	0.299	64	0.674	0.232	4	0.042	0.062

We can draw the following observations from these results:

- Our approach achieves the best results in MyWebMarket, i.e., 85.7% recall using the Top 1 recommendation strategy. In JUnit and JHotDraw, on the other hand, we achieved a recall of 52% and 33.4%, respectively. This difference is somewhat expected since we based on the findings of the exploratory study with MyWebMarket to design and calibrate our approach.

- JDeodorant could find 4 of the 95 Extract Method instances in the oracle. Therefore, its overall recall and precision were 4.2% and 6.2%.

While analyzing the suggestions provided by JExtract and JDeodorant, we observed that some differ from the oracle's suggestion by including/excluding a single statement. To investigate how frequently such scenario occurs, Table 4.4 reports a second set of recall and precision values, using an oracle that tolerates a difference of one single statement from the expected answer. In this scenario, there is a significant improvement in the results, specially for JDeodorant. For example, overall recall improves 34% for JExtract Top 1 and more than 300% for JDeodorant.

Table 4.4. Study 1: Comparing JExtract and JDeodorant (tolerance of one statement)

		JExtract									JDeodorant		
		Top 1			Top 2			Top 3					
System	#	#	Rc.	Pr.	#	Rc.	Pr.	#	Rc.	Pr.	#	Rc.	Pr.
JHotDraw 5.2	56	30	0.536	0.536	36	0.643	0.445	40	0.714	0.401	11	0.196	0.250
JUnit 3.8	25	15	0.600	0.600	20	0.800	0.571	21	0.840	0.569	5	0.200	0.333
MyWebMarket	14	14	1.000	1.000	14	1.000	0.857	14	1.000	0.595	2	0.143	0.500
Overall	95	59	0.621	0.621	70	0.737	0.540	75	0.789	0.475	18	0.189	0.292

In summary, we conclude that, in this particular study, JExtract was significantly superior to JDeodorant, regardless of the configuration used (Top 1, Top 2, or Top 3). The precise reason why JDeodorant was not able to find most of the recommendations in the oracle requires further investigation. However, it is worth noting that, in contrast to our approach, JDeodorant is able to extract non-contiguous code fragments, reordering and duplicating statements when necessary. Such scenarios are not contemplated in this study, but they are listed as future work in Section5.3.

4.1.3 Threats to Validity

There are two main threats regarding the validity of this study. First, as usual, we cannot extrapolate our results to other systems (external validity). Second, we cannot claim that the Extract Method instances we have based the evaluation on represent the whole spectrum of real refactoring instances normally performed by maintainers. However, we can at least assume that the earlier versions of JUnit and JHotDraw have a fairly good internal design and therefore most methods encapsulate a precise design decision. Therefore, we claim that our inlined Extract Methods denote code that should be refactored with a high confidence.

4.2 Study 2: Quantitative Evaluation

In this study, we investigate how our approach performs in a wider range of systems. In order to address this question, we selected 13 well-known open-source Java systems. Similar to the strategy we used for JUnit and JHotDraw, we synthesize a new oracle, which now includes a larger sample of 1,182 Extract Method instances. The remainder of this section describes the study design (Section 4.2.1), the main results (Section 4.2.2), and investigates the impact of alternative ranking strategies (Section 4.2.3).

4.2.1 Study Design

4.2.1.1 Target Systems

We selected a sample of 13 systems from a corpus created by Terra et al. [2013b], which is a compiled version of the Qualitas Corpus [Tempero et al., 2010]. Table 4.5 lists the selected systems, which we chose favoring well-known and active projects. Besides, we aimed to cover a diversity of domains (e.g., XML processing, UML modeling, database, code analysis tools, etc.).

Table 4.5. Study 2: Selected systems

System	Instances	Diferent Classes	Source Files	Avg. Method Size
Ant 1.8.2	99	52 (52.5%)	57	22.1
ArgoUML 0.34	98	39 (39.8%)	65	23.8
Checkstyle 5.6	100	23 (23.0%)	62	17.3
Findbugs 1.3.9	99	51 (51.5%)	58	22.0
Freemind 0.9.0	100	46 (46.0%)	58	19.5
Jasper Reports 3.7.4	100	45 (45.0%)	54	25.0
JEdit 4.3.2	99	51 (51.5%)	48	24.9
JFreechart 1.0.13	100	82 (82.0%)	52	23.1
Log4j 2.0-beta	88	38 (43.2%)	62	20.0
Quartz 1.8.3	70	36 (51.4%)	41	18.1
Squirrel SQL 3.1.2	30	12 (40.0%)	14	19.1
Tomcat 7.0.2	100	54 (54.0%)	51	25.7
Xerces 2.10.0	99	37 (37.4%)	48	39.2
Overall	1,182	566 (47.9%)	670	23.5

Table 4.5 also presents: the number of Extract Method instances we synthesized for each systems (second column); the proportion of the instances that were created by inlining method invocations where the class of the invoker was different from the class of the invoked method (third column); the number of source files changed in the process (fourth column); and the average size—in number of statements—of the methods in the oracle (last column). Moreover, Figure 4.1 shows a histogram of the size of the methods in the oracle. As usual, there are few large methods and many medium/small methods.

4.2.1.2 Oracle Generation

We constructed the oracle using the same process described in Section 4.1.1.3. However, in this case, we did not assume that there are no relevant Extract Method instances in the selected systems. We relaxed this constraint to allow the generation of a large

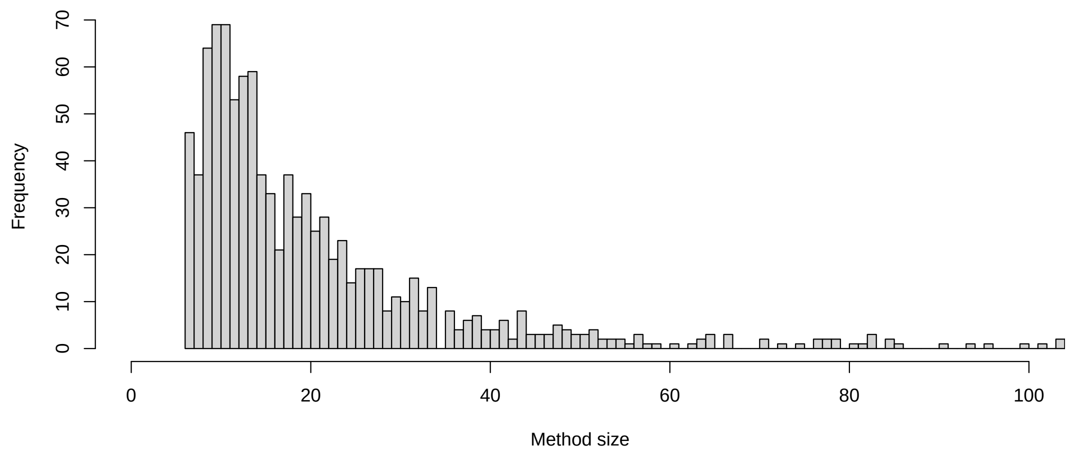


Figure 4.1. Size of the methods in the oracle

dataset, accepting that, in this case, our oracle is possibly incomplete. Nevertheless, we still consider that the instances in the oracle are relevant with a high confidence. Moreover, we introduced the following changes in the process:

- We ignore the visibility of class members (i.e., we changed private, protected, and package members to public) to increase the opportunities to inline invocations of methods that are not from the same class of the invoker. In the previous study, we observed that, due to members' visibility, a small number (8.6%) of inlined methods were from different classes.

- Rather than giving precedence to methods implemented in other classes or to larger methods, we randomly chose the method invocations to be inlined. In such way, we eliminate any bias in the size distribution (but still consider only those greater than the *minimum size threshold*).

- The maximum number of Extract Method instances we generate for a single system is limited to 100, avoiding that the largest systems dominate our sample.

- We do not inline a method more than once in different invocation points. The intention of this rule is introduce more variety in the Extract Method instances of the oracle.

- We apply a maximum of five Inline Method refactoring operations in the same file, also with the intention to introduce more variety.

In total, we synthesized 1,182 instances, which 47.9% has originated from a different class, in 670 source files.

4.2.2 Results and Discussion

Table 4.6 reports recall and precision values achieved by JExtract on the 13 systems. Similarly to the previous study, we present the number of relevant recommendations found (#), recall (Rc.), and precision (Pr.) for each configuration: Top 1, Top 2, and Top 3. We must state that precision, in contrast to the previous study, should be viewed as a lower bound to actual precision because the original systems in this scenario have more chance to include relevant Extract Method instances (besides the ones synthesized by inlining methods). Thus, such instances are considered false positives when reported by our approach.

Table 4.6. Study 2: Recall and precision

		Top 1			Top 2			Top 3		
System	#	#	Rc.	Pr.	#	Rc.	Pr.	#	Rc.	Pr.
Ant 1.8.2	99	26	0.263	0.263	43	0.434	0.221	51	0.515	0.177
ArgoUML 0.34	98	24	0.245	0.245	39	0.398	0.200	53	0.541	0.182
Checkstyle 5.6	100	41	0.410	0.410	64	0.640	0.323	77	0.770	0.266
Findbugs 1.3.9	99	27	0.273	0.273	49	0.495	0.249	67	0.677	0.231
Freemind 0.9.0	100	34	0.340	0.340	46	0.460	0.232	61	0.610	0.207
Jasper Reports 3.7.4	100	28	0.280	0.280	48	0.480	0.241	60	0.600	0.202
JEdit 4.3.2	99	26	0.263	0.263	52	0.525	0.264	60	0.606	0.205
JFreechart 1.0.13	100	15	0.150	0.150	35	0.350	0.176	50	0.500	0.168
Log4j 2.0-beta	88	30	0.341	0.341	50	0.568	0.287	62	0.705	0.239
Quartz 1.8.3	70	19	0.271	0.271	31	0.443	0.221	41	0.586	0.197
Squirrel SQL 3.1.2	30	9	0.300	0.300	12	0.400	0.203	15	0.500	0.170
Tomcat 7.0.2	100	30	0.300	0.300	37	0.370	0.188	47	0.470	0.160
Xerces 2.10.0	99	34	0.343	0.343	44	0.444	0.222	55	0.556	0.186
Overall	1,182	343	0.290	0.290	550	0.465	0.234	699	0.591	0.201

We can draw the following observations from these results:

- The overall results are inferior to the previous study, specially when we inspect Top 1 recall (29.0% vs. 46.3%). However, this difference is less evident in Top 2 recall (46.5% vs. 58.9%) and Top 3 recall (59.1% vs. 67.4%). Nevertheless, we still consider these results acceptable. For example, 59.1% of the oracle instances are covered when using Top 3 configuration.

- The results do not vary widely from system to system (standard deviation of 0.062 in Top 1 recall). The stronger differences are observed in: (i) JFreechart, which presents the worst Top 1 recall (15%); and (ii) Checkstyle, which presents the best Top 1 recall (41%). These observations suggest that our tool can be used in systems from different domains.

Similar to the previous study, Table 4.7 presents the results when we accept a tolerance of 1 statement from the ideal answer. In this scenario, there is a significant improvement in the results, specially on Top 1 recall (42.5% vs. 0.29%).

Table 4.7. Study 2: Recall and precision (tolerance of one statement)

		Top 1			Top 2			Top 3		
System	#	#	Rc.	Pr.	#	Rc.	Pr.	#	Rc.	Pr.
Ant 1.8.2	99	37	0.374	0.374	52	0.525	0.369	59	0.596	0.340
ArgoUML 0.34	98	42	0.429	0.429	54	0.551	0.415	59	0.602	0.357
Checkstyle 5.6	100	55	0.550	0.550	69	0.690	0.520	79	0.790	0.488
Findbugs 1.3.9	99	40	0.404	0.404	63	0.636	0.467	76	0.768	0.448
Freemind 0.9.0	100	43	0.430	0.430	60	0.600	0.414	73	0.730	0.407
Jasper Reports 3.7.4	100	41	0.410	0.410	61	0.610	0.407	67	0.670	0.357
JEdit 4.3.2	99	39	0.394	0.394	56	0.566	0.411	61	0.616	0.346
JFreechart 1.0.13	100	39	0.390	0.390	53	0.530	0.357	64	0.640	0.333
Log4j 2.0-beta	88	38	0.432	0.432	58	0.659	0.471	62	0.705	0.429
Quartz 1.8.3	70	33	0.471	0.471	48	0.686	0.493	53	0.757	0.423
Squirrel SQL 3.1.2	30	10	0.333	0.333	15	0.500	0.322	18	0.600	0.295
Tomcat 7.0.2	100	40	0.400	0.400	47	0.470	0.335	53	0.530	0.297
Xerces 2.10.0	99	45	0.455	0.455	55	0.556	0.374	63	0.636	0.361
Overall	1,182	502	0.425	0.425	691	0.585	0.415	787	0.666	0.378

4.2.2.1 Impact of Method Size

Figure 4.2 shows a barchart comparing the recall values achieved considering only methods of a certain size range. We can observe that we achieve better results for small methods. In fact, since the number of candidates is related to the number of statements of the method, we argue that our observation is somewhat expected, i.e., it is harder to rank the recommendation of the oracle on the top position when there are numerous candidates in the method.

4.2.3 Exploring Ranking Strategies

Taking advantage of this larger dataset, we decided to further investigate ***RQ #2*** from the exploratory study (refer to Section 3.4):

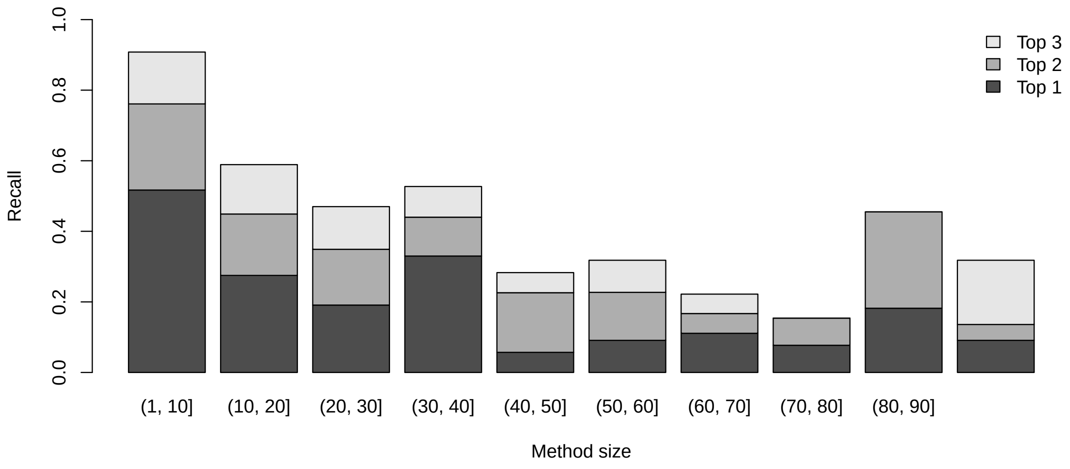


Figure 4.2. Recall for different method size ranges

RQ #2 – What is the best set similarity coefficient and dependency set strategy to rank Extract Method candidates?

First, we investigated if *Kulczynski* is the best coefficient in face of these 13 systems. Table 4.8 shows the recall (using only Top 1 config, for brevity) achieved for each project using each coefficient: Jaccard (JAC), Sorenson (SOR), Sokal and Sneath 2 (SS2), PSC, Kulczynski (KUL), and Ochiai (OCH). We can observe that *Kulczynski* is superior in most systems, except two of them: JFreechart and Log4j. Therefore, this observation corroborates with our decision to use this coefficient.

Table 4.8. Study 2: Comparison of similarity coefficients (Top 1 recall)

System	JAC	SOR	SS2	PSC	KUL	OCH
Ant 1.8.2	0.232	0.242	0.232	0.242	**0.263**	0.232
ArgoUML 0.34	0.163	0.153	0.153	0.214	**0.245**	0.224
Checkstyle 5.6	0.310	0.320	0.310	0.400	**0.410**	0.390
Findbugs 1.3.9	0.253	0.253	0.253	**0.273**	**0.273**	0.253
Freemind 0.9.0	0.250	0.250	0.240	0.260	**0.340**	0.300
Jasper Reports 3.7.4	0.250	0.260	0.260	0.250	**0.280**	0.270
JEdit 4.3.2	0.182	0.182	0.182	0.242	**0.263**	0.242
JFreechart 1.0.13	**0.170**	**0.170**	0.160	0.150	0.150	0.160
Log4j 2.0-beta	0.318	0.318	0.307	0.341	0.341	**0.364**
Quartz 1.8.3	0.200	0.186	0.186	0.243	**0.271**	0.200
Squirrel SQL 3.1.2	0.233	0.233	0.233	0.267	**0.300**	0.267
Tomcat 7.0.2	0.210	0.200	0.210	0.280	**0.300**	0.290
Xerces 2.10.0	0.182	0.192	0.182	0.273	**0.343**	0.303
Overall	0.227	0.228	0.223	0.264	**0.290**	0.270

Second, we investigated if we could improve our results by assigning different weights for each kind of dependency (due to *variables*, *types*, and *packages*). Table 4.9 shows in each column a different weighting scheme, denote by w_v-w_t-w_p, which are the weight factors of the scoring function (refer to Section 3.2.2.2).

Table 4.9. Study 2: Comparison of weighting schemes (Top 1 recall)

System	1-1-1	1-0-0	0-1-0	0-0-1	1-1-0	2-1-1	1-2-1	1-1-2	100-10-1	1-10-100
Ant 1.8.2	0.263	0.242	0.212	0.182	0.273	**0.283**	0.273	0.273	0.263	0.242
ArgoUML 0.34	0.245	0.255	0.163	0.112	**0.296**	0.286	0.224	0.214	0.276	0.204
Checkstyle 5.6	0.410	0.500	0.240	0.180	0.410	0.450	0.390	0.400	**0.520**	0.360
Findbugs 1.3.9	0.273	0.242	0.212	0.182	0.303	0.303	0.263	0.273	**0.313**	0.263
Freemind 0.9.0	0.340	**0.370**	0.140	0.130	0.350	0.360	0.320	0.290	**0.370**	0.230
Jasper Reports 3.7.4	0.280	0.290	0.220	0.190	0.310	**0.320**	0.300	0.250	0.310	0.270
JEdit 4.3.2	0.263	**0.354**	0.192	0.131	0.313	0.313	0.263	0.273	0.343	0.273
JFreechart 1.0.13	0.150	0.140	0.150	0.150	**0.180**	0.160	**0.180**	0.150	0.170	0.170
Log4j 2.0-beta	0.341	0.261	0.148	0.216	0.307	**0.364**	0.330	0.341	0.307	0.341
Quartz 1.8.3	**0.271**	0.171	0.186	0.171	**0.271**	0.243	**0.271**	**0.271**	0.243	0.229
Squirrel SQL 3.1.2	**0.300**	**0.300**	0.167	0.100	0.267	**0.300**	**0.300**	**0.300**	**0.300**	0.267
Tomcat 7.0.2	0.300	0.330	0.140	0.130	**0.340**	0.300	0.270	0.300	0.310	0.260
Xerces 2.10.0	0.343	0.293	0.141	0.121	0.313	**0.354**	0.323	**0.354**	0.313	0.273
Overall	0.290	0.291	0.179	0.156	0.305	0.312	0.284	0.283	**0.313**	0.260

From these results, we outline the following observations:

- No weighting scheme drastically improves the results achieved by the default configuration (1-1-1), since the maximum recall achieved is 31.3% (for the 100-10-1 config).
- There is not a clear overall winner, i.e., the configuration with the best performance (in bold) varies from system to system.
- Nevertheless, there is a tendency towards weighting schemes including *variables*, as opposed to the observations on MyWebMarket, where *types* had a stronger influence. In fact, the worst recall were achieved by the schemes that ignores *variables* (0-1-0 and 0-0-1).

4.2.4 Threats to Validity

In this study, we must also state the threats we previously discussed. First, we cannot extrapolate our results to other systems (external validity). However, we included 13 open-source systems from different domains, which provides at least an indication that our approach is general enough to be performed in other systems. Second, similar to the previous study, we cannot claim that the Extract Method instances we based the evaluation on represent the whole spectrum of real refactoring instances normally performed by maintainers. However, the selected systems are well-known projects, most of them largely used in industrial environments. Therefore, we can at least assume that they have a professional internal design quality and their methods encapsulate relevant design decisions.

4.3 Final Remarks

This chapter reported two studies we conducted to evaluate our approach. In Section 4.1, we compared the effectiveness of JExtract and JDeodorant using three systems: MyWebMarket, JUnit, and JHotDraw. In this particular study, JExtract presented an advantage over JDeodorant. For example, using a typical configuration (Top 3), it achieved overall recall and precision of 67.4% and 23.2%, against 4.2% and 6.2% for JDeodorant.

In Section 4.2, we evaluated our tool with 13 open-source systems. JExtract achieved acceptable results in every system, indicating its applicability in different domains. In this study, the overall recall, using Top 3 config, was 59.1%. Moreover, when investigating alternative ranking strategies we found that: (i) *Kulczynski* was again the best set similarity coefficient; and (ii) different weighting schemes (i.e., values for the weight factors w_v, w_t, and w_p) may yield slightly better results, but not consistently for all systems.[1]

[1]Datasets and a supporting tool that automates their generation are publicly available at JExtract's web site: `http://aserg.labsoft.dcc.ufmg.br/jextract`

Chapter 5

Conclusion

Extract Method is a key refactoring for improving program comprehension and maintainability. Moreover, its one of the most popular refactoring due to its versatility [Fowler, 1999; Murphy et al., 2006; Murphy-Hill et al., 2012; Tsantalis et al., 2013]. However, existing Extract Method supporting tools are most of the times underused and do not support developers in the task of identifying potential code fragments candidates for extraction [Negara et al., 2013; Kim et al., 2012; Murphy-Hill et al., 2012; Murphy-Hill and Black, 2008b].

To address this shortcoming, we proposed an approach to identify Extract Method refactoring opportunities that can be directly automated by IDE-based refactoring tools. Our approach ranks the identified opportunities based on the design principle of separation of concerns. Specifically, we assume that the following sets should be as dissimilar as possible: (i) the set of dependencies established by the code fragment to be extracted; and (ii) the set of dependencies established by the remaining statements in the original method. To compute sets similarity, we rely on *Kulczynski* coefficient, which we chose based on an exploratory study with five other coefficients. A supporting tool, called JExtract, was also implemented.

Our evaluation using a set of synthesized Extract Method opportunities, which were introduced by inlining method invocations, suggests that JExtract is more effective (w.r.t. recall and precision) than JDeodorant, a state-of-the-art tool. Moreover, in a second study with a sample of 1,182 synthesized Extract Method instances from 13 open-source systems, our approach achieved an overall recall of 59.1%.

We organized the remainder of this chapter as follows. First, Section 5.1 reviews the contributions of our research. Next, Section 5.2 points the limitations of our approach. Finally, Section 5.3 presents future work.

5.1 Contributions

This research makes the following contributions:

- An approach that identifies and ranks Extract Method refactoring recommendations, which aim to decompose long or complex methods to improve the system design (Chapter 3).
- An exploratory study conducted to support the decisions regarding the heuristic we used to rank the suggested refactoring opportunities (Section 3.4).
- A publicly available prototype tool, called JExtract, that implements our approach and hence suggests Extract Method refactoring opportunities for a requested method or for the entire system (Section 3.5).
- An evaluation comparing our approach with JDeodorant, a state-of-the-art tool, using two open-source systems and a small scale Web application (Section 4.1).
- A quantitative evaluation using a sample of 1,182 synthesized Extract Method instances from 13 open-source systems (Section 4.2).
- Two publicly available datasets of well-known Extract Method opportunities: the first with 95 instances distributed among three systems; and the second with 1,182 instances distributed among 13 systems (Chapter 4).
- The proposed approach to create the synthesized datasets, including a publicly available supporting tool to automate the process (Section 4.1.1.3 and 4.2.1.2).

5.2 Limitations

Our approach has the following limitations:

- Our approach is limited to suggest the extraction of code fragments that can be automatically applied by current IDE refactoring tools.
- Our approach cannot untangle code fragments before their extraction, i.e., it does not suggest refactoring that requires reordering or duplicating statements previous to the extraction.
- We do not recommend names for the new methods created when applying a suggested Extract Method refactoring.

- We did not evaluate with experts whether the recommendations provided by our approach are helpful in real refactoring tasks.

5.3 Future Work

We intend to complement this research with the following future work:

- *Proposed approach:* (i) the capability to reorder statements before the code extraction to achieve a better separation of concerns; (ii) the recommendation of names for new methods, possibly using a technique based on the work of Sridhara et al. [2011]; and (iii) the combination of Extract Method with Move Method to move misplaced code within a method to a more appropriate class.

- *Evaluation:* (i) a study with experts to assess the quality of the recommendations provided by the tool in real scenarios; (ii) the construction of a new dataset containing real Extract Method instances identified in historical data from open-source systems and (iii) a study to investigate the impact of ignoring core Java types or packages when computing the dependency sets.

- *JExtract Tool:* (i) a study on the possibility to recommend refactoring opportunities for a method that is being edited, without an explicit request by the developer; and (ii) improvements in usability and user interface.

Bibliography

Abadi, A., Ettinger, R., and Feldman, Y. A. (2009). Fine slicing for advanced method extraction. In *3rd Workshop on Refactoring Tools (WRT)*, pages 1–4.

Allen, F. E. (1970). Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19.

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*, pages 1–26.

Beck, K. (2003). *Test-driven Development: By Example.* Addison-Wesley Professional.

Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52(1-3):53–100.

Cimitile, A., Lucia, A. d., and Munro, M. (1996). A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice*, 8(3):145–178.

Ettinger, R. (2012). Program sliding. In *26th European Conference on Object-Oriented Programming (ECOOP)*, pages 713–737.

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.

Gallagher, K. B. and Lyle, J. R. (1991). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761.

Ge, X., DuBose, Q. L., and Murphy-Hill, E. R. (2012). Reconciling manual and automatic refactoring. In *34th International Conference on Software Engineering (ICSE)*, pages 211–221.

Harman, M., Binkley, D., Singh, R., and Hierons, R. M. (2004). Amorphous procedure extraction. In *4th International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 85–94.

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

Kanemitsu, T., Higo, Y., and Kusumoto, S. (2011). A visualization method of program dependency graph for identifying Extract Method opportunity. In *4th Workshop on Refactoring Tools (WRT)*, pages 8–14.

Kaya, M. and Fawcett, J. W. (2013). Identifying Extract Method opportunities based on variable references. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 153–158.

Kim, M., Zimmermann, T., and Nagappan, N. (2012). A field study of refactoring challenges and benefits. In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 50:1–50:11.

Komondoor, R. and Horwitz, S. (2003). Effective, automatic procedure extraction. In *11th International Workshop on Program Comprehension (ICPC)*, pages 33–42.

Lakhotia, A. and Deprez, J.-C. (1998). Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(11):677–689.

Lanubile, F. and Visaggio, G. (1997). Extracting reusable functions by flow graph based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259.

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

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

Maruyama, K. (2001). Automated method-extraction refactoring by using block-based slicing. *Software Engineering Notes*, 26(3):31–40.

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.

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), Practice Track*, pages 401–408.

Murphy, G. C., Kersten, M., and Findlater, L. (2006). How are java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83.

Murphy-Hill, E. R. and Black, A. P. (2008a). Breaking the barriers to successful refactoring: Observations and tools for Extract Method. In *30th International Conference on Software Engineering (ICSE)*, pages 421–430.

Murphy-Hill, E. R. and Black, A. P. (2008b). Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44.

Murphy-Hill, E. R., Parnin, C., and Black, A. P. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18.

Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In *28th International Conference on Software Engineering (ICSE)*, pages 452–461.

Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D. (2013). A comparative study of manual and automated refactorings. In *27th European Conference on Object-Oriented Programming (ECOOP)*, pages 552–576.

O'Keeffe, M. K. and Cinnéide, M. Ó. (2006). Search-based software maintenance. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 249–260.

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

Ott, L. M. and Thuss, J. J. (1993). Slice based metrics for estimating cohesion. In *1st International Software Metrics Symposium (METRICS)*, pages 71–81.

Parnas, D. L. (1994). Software aging. In *16th International Conference on Software Engineering (ICSE)*, pages 279–287.

Robillard, M., Walker, R., and Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86.

Sales, V., Terra, R., Miranda, L. F., and Valente, M. T. (2013). Recommending Move Method refactorings using dependency sets. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241.

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.

Sharma, T. (2012). Identifying Extract Method refactoring candidates automatically. In *5th Workshop on Refactoring Tools (WRT)*, pages 50–53.

Sridhara, G., Pollock, L., and Vijay-Shanker, K. (2011). Automatically detecting and describing high level actions within methods. In *33rd International Conference on Software Engineering (ICSE)*, pages 101–110.

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., Brunet, J., Miranda, L. F., Valente, M. T., Serey, D., Castilho, D., and Bigonha, R. S. (2013a). Measuring the structural similarity between source code entities. In *25th 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. (2012). Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track*, pages 335–340.

Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. (2014). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, pages 1–28.

Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of Move Method refactoring opportunities. *IEEE Transactions on Software Engineering*, 99:347–367.

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.

Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. (2013). A multidimensional empirical study on refactoring activity. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 132–146.

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.

Wang, X., Pollock, L., and Vijay-Shanker, K. (2014). Automatic segmentation of method code into meaningful blocks: Design and evaluation. *Journal of Software: Evolution and Process*, 26(1):27–49.

Weiser, M. (1981). Program slicing. In *5th International Conference on Software Engineering (ICSE)*, pages 439–449.

Wilking, D., Kahn, U. F., and Kowalewski, S. (2007). An empirical evaluation of refactoring. *e-Informatica Software Engineering Journal*, 1(1):27–42.

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

Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.