Recommending Automated Extract Method Refactorings

Danilo Silva
Dept. of Computer Science
UFMG, Brazil
danilofs@dcc.ufmg.br

Ricardo Terra
Dept. of Computer Science
UFLA, Brazil
terra@dcc.ufla.br

Marco Tulio Valente
Dept. of Computer Science
UFMG, Brazil
mtov@dcc.ufmg.br

ABSTRACT

Extract Method is a key refactoring for improving program comprehension. However, recent empirical research shows that refactoring tools designed to automate Extract Methods are often underused. To tackle this issue, we propose a novel approach to identify and rank Extract Method refactoring opportunities that are directly automated by IDE-based refactoring tools. Our approach aims to recommend new methods that hide structural dependencies that are rarely used by the remaining statements in the original method. We conducted an exploratory study to experiment and define the best strategies to compute the dependencies and the similarity measures used by the proposed approach. We also evaluated our approach in a sample of 81 extract method opportunities generated for JUnit and JHotDraw, achieving a precision of 48% (JUnit) and 38% (JHotDraw).

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Software Design, Refactoring

Keywords

Refactoring, Extract Method, Program comprehension supporting tools, Self-documenting methods

1. INTRODUCTION

Extract Method is a key refactoring for improving program comprehension. Besides promoting reuse and reducing code duplication, Extract Method contributes to readability and comprehensibility, by encouraging the extraction of self-documenting methods [1, 2]. However, recent empirical research shows that automated refactoring tools, especially those supporting Extract Method refactorings, are most of the times underused [3, 4, 5, 6, 7]. For example, in a study including developers working in their natural environment, Negara et al. found that the number of participants who are aware of the automated support for Extract Method, but still apply the refactoring manually is higher than the number of participants who predominantly apply this refactoring automatically [3]. Similarly, Kim et al. report that 58.3% of surveyed developers do all of their Extract Method refactorings manually [4]. Finally, Murphy-Hill et al. observed that Extract Method accounts for 12% of the refactoring operations performed by the developers of Eclipse's refactoring tools, but corresponds to only 4.4% of the refactorings performed by ordinary Eclipse's users [5, 6]. Such findings provide evidence that recommendation systems designed to identify Extract Method refactoring opportunities can play an important role in modern IDEs [8]. Particularly, they can be effective instruments to increase the popularity of refactoring tools, mainly among typical IDE users.

In this paper, we propose a novel approach to identify Extract Method refactoring opportunities that can be directly automated by IDE-based refactoring tools. Moreover, we also propose a ranking function to classify the list of initial candidates generated by the proposed approach, according to their potential to improve program comprehension. This function is inspired by the *minimize coupling/maximize cohesion* design guideline. More specifically, 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).

Code 1 presents an example from MyWebMarket, an illustrative but real web-based system [9]. In this code, lines 3–9 represent a well-defined computation: this fragment opens a transaction with the persistence framework (Hibernate) to update a `Product` object in the underlying database. As we can observe, this code fragment manipulates types and variables (`Session` and `Transaction`) that are only used to start and finish an `update` transaction. In other words, these types and their respective variables are not used by the remaining statements in the method (line 2 and lines 10–16). Finally, lines 3–9 respect the Extract Method refactoring preconditions and they can be extracted without any further modification in the code to a new method using the refactoring tool of an IDE. Therefore, our approach would recommend its extraction to a new method.

ICPC '14, June 2-3, 2014, Hyderabad, India

```
public String update() throws Exception {
  logger.info("Starting update()");
  Session s= HibernateUtil.getSessionFactory()
               .openSession(); //hibernate
  Transaction t =
    s.beginTransaction(); //hibernate
  s.update(this.product); //hibernate
  t.commit(); //hibernate
  s.close(); //hibernate
  this.task = SystemConstants.UD_MODE;
  this.addActionMessage(this.getText(
    "updateSuccessful",
    new String[]{"product"})
  );
  logger.info("Finishing update()");
  return INPUT;
}
```

Code 1: Example method from MyWebMarket

We evaluate our approach using two different studies. In the first study, we relied on the MyWebMarket system to conduct an exploratory study with three central goals. First, to experiment and evaluate different strategies to compute the structural dependencies used to rank the Extraction Method candidates. For example, we evaluate dependency sets including three kinds of elements: variables, types, and packages. Second, we evaluate different set similarity coefficients used to define the distance between the dependencies used by a candidate to extraction and the dependencies in the remaining statements in the method. Third, we relied on this exploratory study to contrast our recommendations with the Extract Method candidates provided by JDeodorant, a well-known refactoring recommendation system [8, 10]. Finally, in the second study, we relied on a sample of 81 candidates we have manually synthesized in two well-known systems (JUnit and JHotDraw) to evaluate the precision and recall of our approach. For JUnit, we report that our approach achieved a precision of 48% and a recall of 48%, assuming that a developer initially manifests interest on receiving recommendations for a particular method he/she is maintaining or trying to comprehend. In the same usage scenario, we achieved a precision of 38% and a recall of 38% for JHotDraw. On the other hand, since we generate a massive number of recommendations, our overall precision for the whole system is less than 20%.

The remainder of this paper is organized as follows. Section 2 presents related work on recommendation systems with emphasis on systems that provide Extract Method recommendations. Section 3 details our recommendation approach, including its two central phases (Extract Method candidates generation and ranking). Sections 4 and 5 present the exploratory study and the evaluation with JUnit and JHotDraw, respectively. Section 6 concludes the paper and suggests further research work.

2. RELATED WORK

We organized this section into four groups: (a) JMove, which is a Move Method refactoring recommendation system our approach is inspired by; (b) JDeodorant, which is a well-known technique for identifying Extract Method refactoring opportunities, which we used to compare our approach to; (c) other Extract Method tools directly (or indirectly) related to our approach; and (d) an overview on the state-of-the-art recommendation systems.

2.1 JMove

In a recent work, we described a technique for identifying Move Method refactoring opportunities based on the similarity between dependency sets [11]. This technique is implemented by a recommendation system called JMove, which detects methods located in incorrect classes and then suggests moving such methods to more suitable ones. More specifically, the proposed technique initially retrieves the set of static dependencies established by a given method m located in a class 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 the step (b) is greater than the similarity measured in (a), it infers 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.

Whereas JMove recommends moving a method m to a more suitable class C_i, our current approach recommends extracting a fragment from a given method m into a new method m' when there is a high dissimilarity between m' and the remainder statements in m.

Finally, it is worth noting that there are other techniques to identify Move Method opportunities based, for example, on search-based algorithms [12, 13], Relational Topic Model (RTM) [14], metrics-based rules [15], etc.

2.2 JDeodorant

Proposed by Tsantalis and Chatzigeorgiou, JDeodorant is a system that can identify and apply some common refactoring operations on Java systems, including Extract Method [8]. Their approach relies on the concept of program slicing to select related statements that can be extracted into a new method. Specifically, two criteria are used to compute such slices: (a) the full computation of a variable, referred to as complete computation slice; (b) all code that affects the state of an object, referred to as object state slice. Nevertheless, our approach differs from JDeodorant in three main points:

- Our algorithm for generating Extract Method candidates imposes as few restrictions as possible and it is not based on specific code patterns (like a complete computation slice).

- We are more conservative when enforcing rules to preserve program behavior, as statement reordering or duplication are never allowed. While this may be considered a drawback because we might miss some refactoring opportunities, it is also an advantage in terms of soundness. Even with the sophisticated static analysis used in JDeodorant, some subtle dependencies between statements are not detected, for example, due to dependencies established via external resources, such as files and databases.

- We heavily rely on a scoring function, which considers the structural dependencies present in the code, to filter and rank the recommendations, whereas JDeodorant relies on specific slicing criteria to generate meaningful recommendations.

2.3 Other Extract Method Tools

The vast majority of the studies found in the literature for extracting methods are also based on the concept of program slicing [16, 17, 18]. For example, Abadi et al. [16] proposed the use of fine slicing to support the Extract Method refactoring. The method computes executable program slices that can be finely tuned, and can be then used to extract non-contiguous fragments of code.

Murphy-Hill and Black [19] reported a set of usability guidelines for automated Extract Method refactoring tools. By implementing three refactoring assisting tools according to the proposed guidelines, the authors show that speed, accuracy, and user satisfaction can be significantly increased. However, the described tools do not aim to identify and rank Extract Method refactoring opportunities.

In view of the automatic refactoring tools underuse problem, Ge and colleagues [20] reported a set of manual refactoring workflow patterns, including those frequently adopted by developers when performing Extract Methods. Using these patterns, they propose BeneFactor, a refactoring tool that detects a developer's manual refactoring and reminds him that an automatic refactoring supported is available. On the other hand, our approach aims to detect refactoring opportunities that developers are not aware about.

2.4 Other Recommendation Systems

Recommendation Systems for Software Engineering (RSSEs) is an emerging research area [21]. An RSSE is a software application that provides potential valuable information for a software engineering task in a given context. In this paper, we report a solution based on recommendation system principles with focus on identifying Extract Method refactoring opportunities.

Furthermore, RSSEs can help developers in a wide range of activities, besides refactoring-related purposes. For example, current RSSEs can recommend relevant source code fragments to help developers to use frameworks and APIs (CodeBroker [22], Strathcona [23], and APIMiner [24]), software artifacts that must be changed together (eRose [25]), parts of the software that should be tested more cautiously (Suade [26]), and tasks for repairing software architectures (ArchFix [27]).

3. PROPOSED APPROACH

The proposed approach has two main steps. First, we generate an exhaustive list of Extract Method candidates, which are block of statements that can be directly marked by a developer and extracted to a method using the refactoring support of modern IDEs (Section 3.1). In a second step, the list of candidates is ranked in order to recommend the best Extract Method opportunities to the developer. This ranking is based on the values returned by a similarity coefficient, using an approach inspired by our previous work with the JMove tool (Section 3.2).

3.1 Candidates Generation

For each method M_i of a system S, this algorithm generates candidates for extraction. To generate the candidates, the algorithm first relies on a hierarchical model representing the block structure of the method's source code. In this model, a block is a sequence of continuous statements, i.e., statements that follow a linear control flow.

Code 2 presents a method from JHotDraw, which will be used in this section to illustrate our approach.

```
1   public Rectangle2D getFigureDrawBounds() {
2     Rectangle2D r = super.getFigDrawBounds();
3     if (getNodeCount() > 1) {
4       if (START.get(this) != null) {
5         Point p1 = getPoint(0, 0);
6         Point p2 = getPoint(1, 0);
7         r.add(START.get(this).getBounds(p1, p2));
8       }
9       if (END.get(this) != null) {
10        Point p1= getPoint(getNodeCount()-1, 0);
11        Point p2= getPoint(getNodeCount()-2, 0);
12        r.add(END.get(this).getBounds(p1, p2));
13      }
14    }
15    return r;
16  }
```

Code 2: Example method from JHotDraw

Figure 1 illustrates the block structure of Code 2. We can observe that the code is composed of four blocks of statements. The first block has three statements (statements 2, 3, and 15). Statement 3 has a child block (block 2), composed by statements 4 and 9. Finally, each of such statements has their own child blocks (block 3, composed by statements 5 to 7; and block 4, composed by statements 10 to 12).

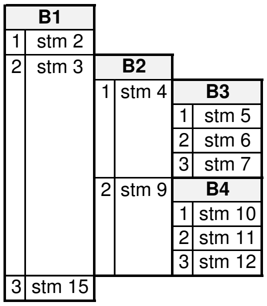


Figure 1: Block structure retrieved from Code 2

After retrieving the block structure, we generate a list with sub-sequences of statements that can be extracted from each block, as described in Algorithm 1. It is important to note that not every possible sub-sequence turns into a candidate because it must follow at least three kinds of preconditions relevant when performing an Extract Method: Syntactical Preconditions, Behavior-Preservation Preconditions, and Quality Preconditions.

Syntactical Preconditions: These preconditions concern the syntactical organization of the statements selected for extraction. The three nested loops in Algorithm 1 (lines 2, 4, and 5) guarantee that the list of selected statements attend the following preconditions:

- The selected statements are part of a single block of statements. For example, considering the blocks in Figure 1, it is not possible to select a candidate with just statement 5 (block 3) and statement 10 (block 4).
- Inside a block, only continuous statements are selected. For example, in Figure 1, it is not possible to select a candidate with only statements 5 and 7.

- If a statement is selected, then all statements in its child blocks of statements are automatically selected. For example, if statement 4 is selected in Figure 1, then statements 5 to 7 are automatically selected.

Algorithm 1 Candidates generation algorithm

```
Input: A method $M$
Output: List with extraction 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 first two preconditions are naturally guaranteed by the way that source code fragments are selected in an IDE, where it is just possible to select continuous code fragments. Therefore, they guarantee that the recommendations provided by our approach can be directly extracted to methods, without statement reordering. Finally, the third precondition is checked by the refactoring tool.

Behavior-Preservation Preconditions: Algorithm 1 relies on the $isValid$ function (line 7) to check whether the candidates preserve the program's behavior. For example, if two or more variables are assigned by a code fragment selected for extraction and they are used by other statements we cannot apply an Extract Method. The reason is that it is not possible to define a method with multiple return values.[1] For example, a code fragment with statements 5 and 6 in Code 2 is not a valid candidate since the variables `p1` and `p2` are assigned by these statements and further used at statement 7. Currently, our prototype implementation relies on the preconditions defined by the Extract Method refactoring tool provided by the Eclipse IDE.

Quality Preconditions: Algorithm 1 also relies on the $isValid$ function (line 7) to check whether the candidates to extraction follow minimal design quality preconditions. For this purpose, we currently propose just 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 a minimum number of statements K. More specifically, assuming that C is a set of candidate statements and M' are the remaining statements in the method, we check the following condition:

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

In other words, 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 will never produce candidates.

3.2 Ranking

After the candidates generation step, a scoring function is used to rank the candidates and to show the most relevant ones as Extract Method recommendations. The proposed function considers the static dependencies established by a candidate to extraction. 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.

3.2.1 Dependency Sets

The proposed scoring function considers that a block of statements B can establish dependencies with variables, types, and packages, which are denoted by $Dep_{var}(B)$, $Dep_{type}(B)$, and $Dep_{pack}(B)$, respectively. These sets are constructed as described next.

- *Variables:* If a statement s from a block of statements B declares, assigns, or reads a variable v, then v is added to $Dep_{var}(B)$. The reason to consider this form of dependency is that methods should encapsulate the use of local variables (and therefore the partial computation they are used for).

- *Types:* If a statement s from a block of statements B uses a type (class or interface) T, then T is added to $Dep_{type}(B)$. The reason to consider this form of dependency is that methods should also hide the services provided by well-defined sets of types. For example, in Code 1 a method can be extracted to hide the services provided by the `Session` and `Transaction` types, used to connect with Hibernate. Specifically, we consider that the following scenarios characterize the use of a type:
  - If s calls a method m, the type T that declares m is included in $Dep_{type}(B)$.
  - If s reads or writes to a field f, the type T that declares f is included in $Dep_{type}(B)$.
  - If s creates an object of a type T, then T is included in $Dep_{type}(B)$.
  - If s declares a variable v, the type T of v is included in $Dep_{type}(B)$.
  - If s handles an exception of type T, then T is included in $Dep_{type}(B)$.

- *Packages:* For each type T included in $Dep_{type}(B)$, as described in the previous item, the package where T is implemented and all its parent packages are also included in $Dep_{pack}(B)$. For example, if a type `foo.bar.Baz` is used, then `foo` and `foo.bar` are included in $Dep_{type}(B)$. The reason to consider this form of dependency is analogous to types, but taken at a higher level of abstraction. It is worth noting that common root packages, such as `com` or `java`, are ignored as they do not denote meaningful modules.

3.2.2 Scoring Function

The distance between dependency sets Dep and Dep' is defined as:

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

[1] Currently, we implemented our approach for Java. However, it is straightforward to adapt it to other OO languages.

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

This distance is defined using the Kulczynski set similarity coefficient [28, 11, 29]. 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 4 presents an exploratory study we conducted to decide to use Kulczynski to compute this distance.

Suppose a candidate to extraction C (a list of statements) defined in a method M. Suppose also that M' are the remaining statements in M. The ranking score of C is defined as the sum of the distance of the dependency sets of M and M', using variables, types, and packages:

$$\begin{aligned} score(C) = \; & 1/3 \times dist(Dep_{var}(C), Dep_{var}(M')) \; + \\ & 1/3 \times dist(Dep_{type}(C), Dep_{type}(M')) \; + \\ & 1/3 \times dist(Dep_{pack}(C), Dep_{pack}(M')) \end{aligned}$$

The assumption is that a good candidate to method extraction should encapsulate the use of variables, types, and packages, as expressed by the proposed dependency sets. In other words, the variables, types, and packages manipulated by a candidate C should be very different from the ones manipulated by the statements that will remain in the original method after extracting C to a new method. Note that the three kinds of dependencies have the same weight in this formulation. In future work, we plan to investigate whether some types of dependencies are more important than others to rank the refactoring recommendations.

4. EXPLORATORY STUDY

4.1 Research Questions

We designed an exploratory study to address the following overarching research questions:

RQ #1 – How many Extract Method instances respect the preconditions required 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?

RQ #5 – How does the provided recommendations compare with JDeodorant's ones?

4.2 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 the second author of this paper to evaluate refactoring recommendations to repair software architectural violations [9]. 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 [30].[2]

MyWebMarket was first implemented as a monolithic system (version used in this study) and has evolved to more modularized versions. In this first monolithic version, there are 25 instances of Extract Method, which were deliberately introduced by the system's developer. Therefore, we consider such instances as a trustworthy oracle.

4.3 Study Setup

To provide answers to our research questions, we performed the following tasks:

- We initially set the minimal threshold value to 3 (i.e., $K = 3$) to answer RQ #1 and RQ #2.
- When investigating RQ #2, we considered the similarity coefficients presented in Table 1. To measure similarity between two blocks of statements B and B' (i.e., $S_{BB'}$), the considered coefficients rely on variables a (the number of dependencies on both blocks), b (the number of dependencies on block B only), and c (the number of dependencies on block B' only).

Table 1: General Purpose Similarity Coefficients

Coefficient	Definition $S_{BB'}$	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.

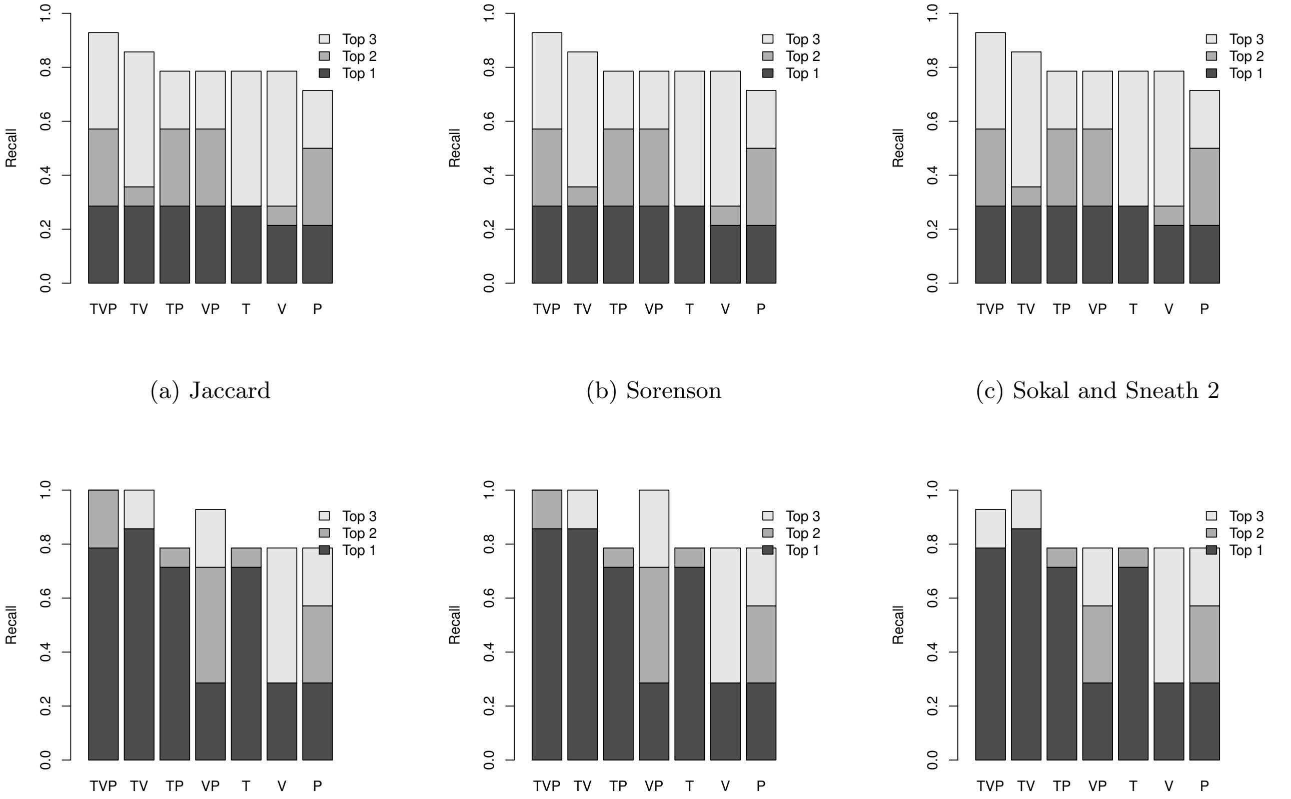
4.4 Results and Discussion

4.4.1 *RQ #1: How many Extract Method instances respect the preconditions required by the proposed approach?*

In MyWebMarket, 14 Extract Method instances (56%) respect the preconditions proposed in Section 3.1. The remaining 11 instances do not attend the syntactical preconditions because they refer to non-contiguous code.

As an example, Code 3 presents an Extract Method instance where we failed to provide a correct recommendation. The presented method should implement only functional concerns. Therefore, the persistence-related code (lines 3–5 and 13–15) should be extracted into a new method. However, the automated refactoring support provided by an IDE cannot handle this code because it is non-continuous. As a matter of fact, if we move lines 3–5 to after line 11 (which preserves behavior in this case), our approach would suggest the correct Extract Method recommendation. Likewise, the remaining 10 MyWebMarket's failing instances also fits in this pattern, i.e., we would be able to suggest the correct

[2] MyWebMarket source code can be downloaded from: `http://aserg.labsoft.dcc.ufmg.br/jextract`



(a) Jaccard

(b) Sorenson

(c) Sokal and Sneath 2

(d) PSC

(e) Kulczynski

(f) Ochiai

Figure 2: Top n recall using different similarity coefficients and entity set combinations

recommendations after making subtle behavior-preservation changes in the code, typically statement reordering.

```
 1  public String save() throws Exception {
 2    logger.info( "Starting save()" );
 3    Session s= HibernateUtil.
 4              getSessionFactory().openSession();
 5    Transaction t = sess.beginTransaction();
 6
 7    purchaseOrder.setOrderDate(new Date());
 8    for (PurchaseOrderItem item :
 9      purchaseOrder.getPurchaseOrderItems()){
10      item.setPurchaseOrder(this.purchaseOrder);
11    }
12
13    s.save(this.purchaseOrder);
14    t.commit();
15    s.close();
16
17    this.task = SystemConstants.UD_MODE;
18    ....
19  }
```

Code 3: A non-contiguous Extract Method instance (lines 3–5 and 13–15)

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

This research question aims to investigate the impact in our results of two central decisions made when designing our approach: (a) the set similarity coefficient used to compute the distance between dependency sets; (b) the strategy used to compute dependency sets, i.e., which is the best combination of dependencies (due to the use of variables, types, or packages). For each set similarity coefficient reported in Table 1, we tested the recommendations produced by each possible combination of dependency sets, as presented in Figure 2. In this figure, `T` stands for *Types*, `V` for *Variables*, and `P` for *Packages*. As mentioned, we also investigate combinations of dependency sets, e.g., `TV` stands for a compound score of *Types* and *Variables* dependency sets.

Figure 2 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 28.6%

of the Extract Method instances only looking at the first recommendation (Top-1) for each method.

Two central findings come from the results in Figure 2:

- 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 a maximum Top-1 recall of 28.7%; and (ii) effective coefficients—namely Ochiai, PSC, and Kulczynski—that achieved recall values above 80%.

- 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. However, combining dependency sets can slightly increase the recall achieved by just `T`. For example, both PSC and Kulczynski achieved their maximum recall value using the dependency sets for types, variables, and packages (i.e., `TVP`).

In summary, the combination of the Kulczynski coefficient and the `TVP` dependency sets achieved the maximal recall value. 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.

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

Considering that after exploring the RQ #2 our approach was set up to use the Kulczynski coefficient and the `TVP` dependency sets, this third research question investigates the impact of the minimal threshold parameter (K) on the recall results achieved by this particular setup. Figure 3 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 (the value used when investigating the RQ #2). For example, the Top-2 recall decreases from 100% (K=3) to 22% (K=6).

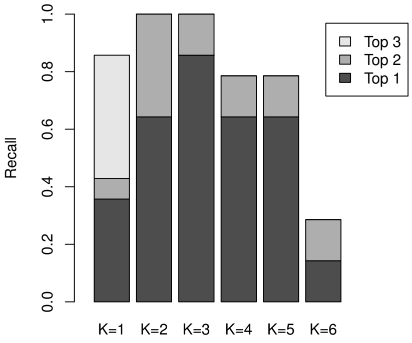


Figure 3: Top n recall using different values of K

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

In the previous research questions, we used recall to guide central design decisions in our approach (RQ #2) and to evaluate the relevance of the minimal size threshold parameter (RQ #3). However, as usual, any recall measure should be complemented by measures of precision.

When reasoning about precision, it is important to report that our approach generates a large number of recommendations, so it heavily depends on the ranking function to filter out non-relevant ones. Specifically for MyWebMarket, we generate 951 valid recommendations, which correspond to 35.2 recommendations/method on average.

Figure 4 shows the overall precision and recall of our approach, for different threshold values of the scoring function. In this case, we first removed from the rank all recommendations with a ranking score less than a given threshold (x-axis). Therefore, the removed recommendations were not considered when measuring the presented precision results (y-axis). We can observe that when evaluating all recommendations, our overall precision is very low, usually less than 20%, unless we set up a minimum score above 0.8. But, in this case, recall faces a significant decrease, falling below 30%. In fact, Figure 4 shows that there is no score threshold that gives high precision without sacrificing recall. 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.

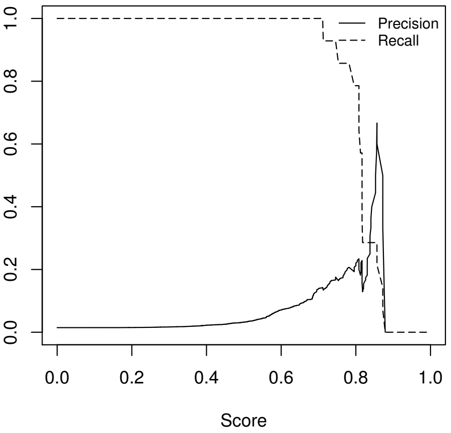
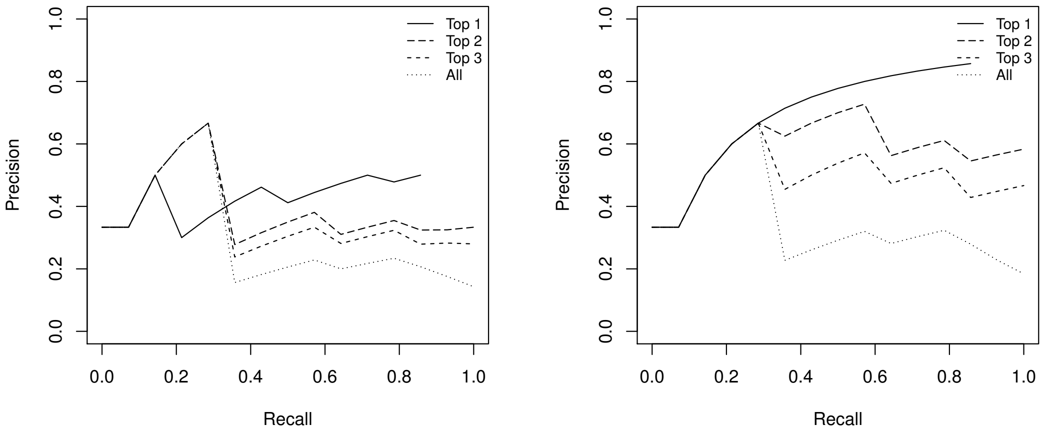


Figure 4: Precision and recall by score threshold (MyWebMarket)

This first observation led us to adopt the strategy of limiting the number of recommendations for a single method on the rank. For example, we may consider only the Top-1 recommendation of each method. Figure 5 shows the precision-recall curves in this case. These curves show the precision and recall achieved by presenting the Top-k elements in the rank. In other words, the curve shows how precision and recall vary as the maintainer proceeds with the examination of the recommendations in our ranking, starting from the top-ranked recommendation. We can observe that by using the Top-1 strategy it is possible to achieve an overall precision of 50.0%, while preserving a high recall (85.7%).

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 to a use case where a developer



(a) Overall

(b) Method-level

Figure 5: Precision vs Recall on MyWebMarket

initially manifests interest on receiving automated Extract Method refactoring recommendations for a particular method he/she is maintaining or trying to comprehend. In this case, as presented in Figure 5b, we could achieve a precision of 85.7% and a recall of 85.7%.[3]

Lastly, the Top-2 and Top-3 strategies trade precision for a higher recall. More specifically, as showed by Figure 5b, we have a decrease in precision even when considering two recommendations per method (from 85.7% for Top-1 to 58% with Top-2). On the other hand, it is possible to achieve a recall of 100% with two recommendations per method.

Therefore, we can summarize our answer to this research question as follows:

- It is not feasible to present all generated and valid recommendations, since we generate a massive number of recommendations and therefore we would achieve very low precision rates.
- However, presenting just one recommendation per method yields better results. In other words, the proposed score function is most effective when used to rank recommendations confined to the scope of methods. In practical terms, a tool implementing our approach should suggest only a few recommendations per method, preferably just the best one.

4.4.5 RQ #5: How does the provided recommendations compare with JDeodorant's ones?

In this last research question, we compare our approach with JDeodorant, just to make a preliminary assessment on the quality and relevance of our recommendations, considering the findings of the previous research questions. We set up JDeodorant with its default values, except for the parameter *minimal number of slice statements*, which was set to 3 to be consistent with the *minimal size threshold K* of our approach.

JDeodorant suggested 33 distinct Extract Method recommendations for the whole system. Only two of them matched the oracle recommendations. From the remaining 31, 29 would introduce changes in the behavior of the code because they suggested reordering or duplicating statements involved in a database transaction. For example, in Code 3, the commit statement (line 14) cannot be moved before the save operation (line 13).

In summary, JDeodorant achieved an overall precision of 6.0% and a recall of 14.3%. These values are inferior to the overall results of our approach using the Top-1 strategy (precision of 50.0% and recall of 85.7%).

4.5 Threats to Validity

We must state at least one major threat to the external validity of the reported evaluation. 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 [30]. We should also emphasize that MyWebMarket was developed by the second author of this paper. However, since the system was created two years ago, in the context of another research, we claim that a possible bias favoring the approach presented in this paper is not a relevant threat.

5. EVALUATION

After the findings of the exploratory study, we conducted a second experiment to evaluate our approach with two well-known open-source systems, JUnit and JHotDraw. For this evaluation, we judiciously applied *Inline Method* refactoring operations to these systems in order to create system's versions with well-known *Extract Method* instances. Using these

[3] By construction, considering one recommendation per valid method, precision will be equal to recall, when evaluating the whole set of recommendations. In this particular case, a false positive implies that the only existing refactoring instance in the method was missed.

versions, we report our results in terms of precision (both overall precision and method-level precision) and recall.

5.1 Target Systems

A basic premise of our oracle generation approach is that the original system should have a good design quality. Specifically, it is important to select systems with no Extract Method opportunities (if possible) or with a minimal number of such opportunities. In this way, the synthesized instances will be by far the most significant ones in the code. We tried to address this premise by choosing two well-known systems: JUnit (version 3.8) and JHotDraw (version 5.2). Both systems, especially in their earlier versions we have selected, were designed and implemented by expert software developers.

5.2 Oracle Construction Process

To construct the oracles, we followed these steps:

1. We first retrieved all methods of the target systems with more than three statements (minimal size threshold, as recommended by the exploratory study described in Section 4).

2. For each method retrieved in Step 1, we retrieved all other methods it invokes. These methods should also pass on the following checks:

    - They must have at least $N/4$ statements and at maximum $2 \times N$ statements, where N is the number of statements of the invoker. Besides, its absolute size must be at least three statements to conform to the chosen value of $K = 3$. Avoiding very small or very large inlines is important to increase the chances that the introduced Extract Method opportunity is more significant than the ones that might exist in the original methods.

    - They must pass on the Inline Method preconditions of the IDE.

3. From the list of inline candidates computed at Step 2, we selected a single candidate to apply an *Inline Method* refactoring. We gave precedence to methods implemented in other classes and, as a second criterion, to the method with more statements.

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

Table 2 presents the number of Extract Method instances in our oracle. As can be observed, we generated valid instances for 26.0% and 25.2% of the methods able to produce candidates in JUnit and JHotDraw, respectively (i.e., methods with at least six statements, assuming the minimal size threshold $K = 3$).

Table 2: 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.

5.3 Results

In total, we generated 4,862 Extract Method candidates for JUnit (50.7 recommendations/method) and 4,473 candidates for JHotDraw (20.1 recommendations/method), assuming $K = 3$ (minimal size threshold). Again, such values show that our approach heavily depends on the ranking score function to avoid the presentation of such massive number of recommendations to a maintainer, which would clearly undermine its applicability in real maintenance settings.

Figure 6a and Figure 6c present the overall precision vs recall curves for the target systems, considering all recommendations, just the best recommendation for each method (Top-1), and also Top-2 and Top-3 best recommendations per method. As in the case of MyWebMarket, the overall precision results are usually not satisfactory, both for JUnit and JHotDraw. For example, a precision near 40% is only achievable by accepting a very low recall rate of 12%, for JUnit. It is also worth noting that the Top-1, Top-2, and Top-3 results cannot reach a recall of 100% since they filter out many recommendations. When considering all recommendations it is possible to achieve a recall of 100%, but only when accepting a minimal precision.

On the other hand, Figure 6b and Figure 6d present the method-level precision, i.e., the precision calculated after filtering out the recommendations that are not related to the methods in the oracle. In this scenario, in the end of the presented curves, for JUnit we can achieve a precision of 48% and a recall of 48% using the Top-1 recommendation strategy. Moreover, it is possible to achieve very high precision rates (greater than 80%) at low recall rates (less than 20%). For JHotDraw, the results are slightly worse. For example, the Top-1 recommendation strategy can achieve a precision of 38% and a recall of 38%.

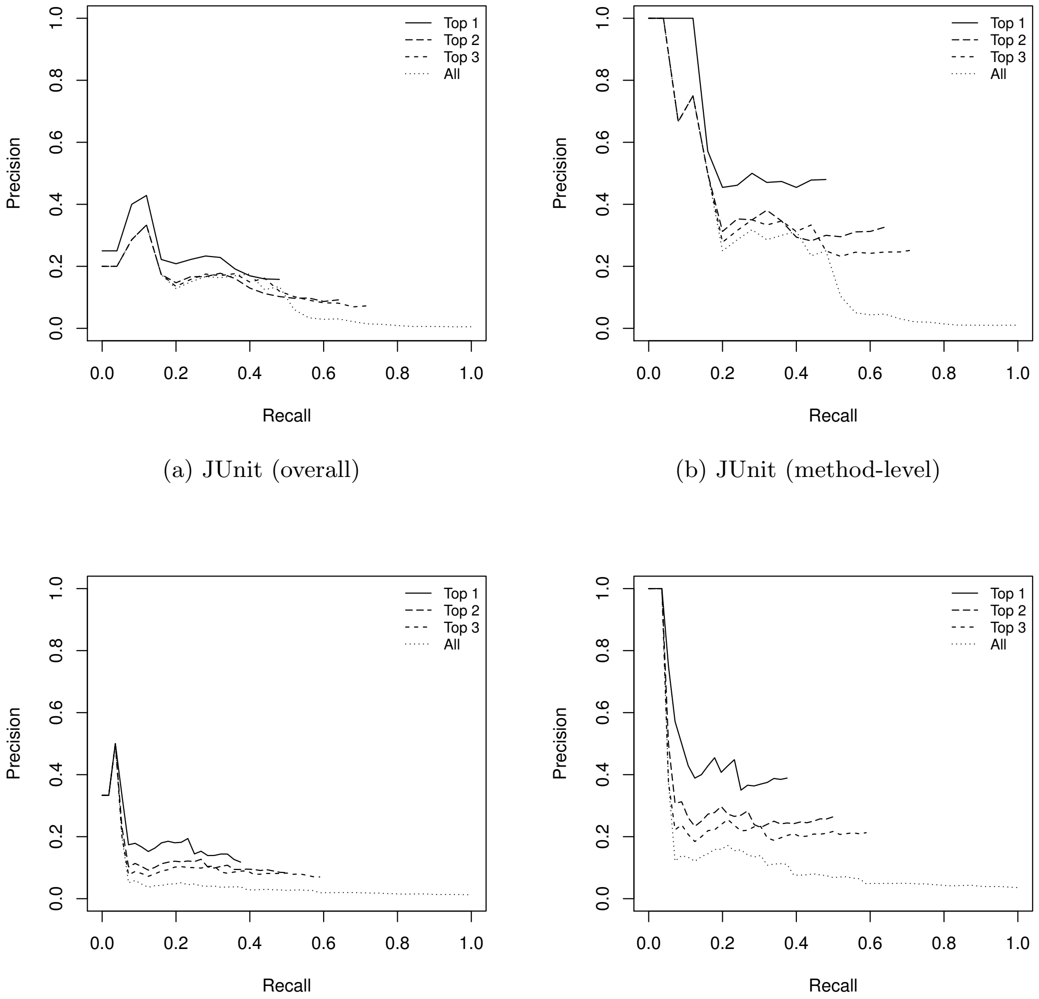
These results confirm our findings in the MyWebMarket study, regarding our potential to produce acceptable results (close to 50% in JUnit) when assuming that a developer initially manifests interest on receiving automated Extract Method refactoring recommendations for a particular method he/she is maintaining. Specifically, he/she should ask for a refactoring suggestion when facing difficulty to comprehend the code of a method, in order to fix, for example, a reported bug. Usually, such refactorings are more common than exclusive refactoring procedures [6, 7].

5.4 Threats to Validity

There are two main threats regarding the validity of this second 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 of their methods encapsulate relevant design decision. Therefore, we claim that our inlined Extract Methods denote code that should be refactored with a high confidence.

6. CONCLUSION

The central contribution of this paper is a novel approach for recommending automated Extract Method refactorings. This approach can contribute to increase the popularity of IDE-based refactoring tools, which are normally considered



(a) JUnit (overall)

(b) JUnit (method-level)

(c) JHotDraw (overall)

(d) JHotDraw (method-level)

Figure 6: Precision vs recall for the target systems (overall and method-level)

underused by most recent empirical studies on refactoring. Specifically, it can contribute to promote the use of Extract Method, which is often considered as one of the key refactorings for improving program comprehension due to its positive impact on a program's readability.

The proposed approach outperformed JDeodorant—a well-known refactoring recommendation system—in detecting Extract Method opportunities in an illustrative but real web-based e-commerce system. Moreover, the described approach achieved precision and recall rates close to 50% when detecting refactoring instances in JUnit and slightly smaller rates in JHotDraw. As its current main limitation, our recommendation approach is less effective when used to generate a system-wide ranking of Extract Method opportunities. In such cases, we observed a relevant decrease in precision and recall, which achieved at best rates close to 16% (precision) at 48% (recall), for JUnit. For this reason, we envision a usage scenario where the developer has an initial insight on the methods that need to be refactored (e.g., long, complex, or buggy methods).

As future research, we intend to extend the evaluation described in Section 5 by also considering the results provided by JDeodorant, when executed in our synthesized sample of Extract Method instances. Moreover, we intend to conduct a new evaluation, including real Extract Method instances, possibly using the refactoring dataset provided by Tsantalis and colleagues [31], and also an evaluation with developers. Finally, we intend to integrate our current prototype implementation with JMove, our previous Move Method recommendation system [11].

Acknowledgment

Our research is supported by CAPES, FAPEMIG, and CNPq.

7. REFERENCES

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

[2] W. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

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

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

[5] E. R. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *31st International Conference on Software Engineering (ICSE)*, 2009, pp. 287–297.

[6] ——, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[7] E. R. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, 2008.

[8] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[9] R. Terra, M. T. Valente, K. Czarnecki, and R. Bigonha, "Recommending refactorings to reverse software architecture erosion," in *16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track*, 2012, pp. 335–340.

[10] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 99, pp. 347–367, 2009.

[11] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending move method refactorings using dependency sets," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 232–241.

[12] M. K. O'Keeffe and M. Ó. Cinnéide, "Search-based software maintenance," in *10th European Conference on Software Maintenance and Reengineering (CSMR)*, 2006, pp. 249–260.

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

[14] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, pp. 1–26, 2014.

[15] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[16] A. Abadi, R. Ettinger, and Y. A. Feldman, "Fine slicing for advanced method extraction," in *3rd Workshop on Refactoring Tools (WRT)*, 2009, pp. 1–4.

[17] D. Liang and M. Harrold, "Slicing objects using system dependence graphs," in *14th International Conference on Software Maintenance (ICSM)*, 1998, pp. 358–367.

[18] R. Ettinger, "Program sliding," in *26th European Conference on Object-Oriented Programming (ECOOP)*, 2012, pp. 713–737.

[19] E. R. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: Observations and tools for extract method," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 421–430.

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

[21] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, 2010.

[22] Y. Ye and G. Fischer, "Reuse-conducive development environments," *Automated Software Engineering*, vol. 12, pp. 199–235, 2005.

[23] R. Holmes, R. Walker, and G. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 952–970, 2006.

[24] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente, "Documenting APIs with examples: Lessons learned with the APIMiner platform," in *20th Working Conference on Reverse Engineering (WCRE), Practice Track*, 2013, pp. 401–408.

[25] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[26] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *28th International Conference on Software Engineering (ICSE)*, 2006, pp. 452–461.

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

[28] B. S. Everitt, S. Landau, M. Leese, and D. Stahl, *Cluster Analysis*, 5th ed. Wiley, 2011.

[29] R. Terra, J. Brunet, L. F. Miranda, M. T. Valente, D. Serey, D. Castilho, and R. S. Bigonha, "Measuring the structural similarity between source code entities," in *25th Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2013, pp. 753–758.

[30] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software: Practice and Experience*, vol. 32, no. 12, pp. 1073–1094, 2009.

[31] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *23rd Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 2013, pp. 132–146.