Recommending Source Code Locations for System Specific Transformations

Gustavo Santos*, Klérisson V. R. Paixão†, Nicolas Anquetil*,
Anne Etien*, Marcelo de Almeida Maia†, Stéphane Ducasse*
*Université de Lille, CNRS
Centrale Lille, INRIA, UMR 9189 – CRIStAL, France
Email: gugajansen@gmail.com, {nicolas.anquetil, anne.etien, stephane.ducasse}@inria.fr
†Federal University of Uberlândia, Brazil
Email: {klerisson, marcelo.maia}@ufu.br

***Abstract*—From time to time, developers perform sequences of code transformations in a systematic and repetitive way. This may happen, for example, when introducing a design pattern in a legacy system: similar classes have to be introduced, containing similar methods that are called in a similar way. Automation of these sequences of transformations has been proposed in the literature to avoid errors due to their repetitive nature. However, developers still need support to identify all the relevant code locations that are candidate for transformation. Past research showed that these kinds of transformation can lag for years with forgotten instances popping out from time to time as other evolutions bring them into light. In this paper, we evaluate three distinct code search approaches ("structural", based on Information Retrieval, and AST based algorithm) to find code locations that would require similar transformations. We validate the resulting candidate locations from these approaches on real cases identified previously in literature. The results show that looking for code with similar roles, *e.g.*, classes in the same hierarchy, provides interesting results with an average recall of 87% and in some cases the precision up to 70%.**

I. Introduction

Developers sometimes perform sequences of source code transformations in a systematic way [1], [2]. These sequences are composed of small code transformations (*e.g.*, create a class, then extract a method to this class), which are applied to groups of somehow related code entities (*e.g.*, methods in sibling classes). Due to the repetitive nature of these transformations, manually applying them is a tedious and error-prone task [3].

Existing tools automate the application of repetitive transformations [4]–[9]. However, once the developers know the sequence of transformations to perform, finding all the code entities that are candidates for these transformations involves inspecting the entire source code of the system. In fact, past research showed that developers forget code locations that are candidate for transformation, some of these locations popping out from time to time as other evolutions bring them to light [10].

To find candidates for a given transformation, we start from the assumption that similar code entities might be transformed in a similar way. Thus, clone detection and code search tools can be used to identify these candidates [11]–[15]. In general, these tools use as input the source code of the system and one source code example. Figure 1 (upper part) depicts the expected behavior of these tools. As a result, code search tools generate a list of code locations that are similar to the given example. But, they still require that, for each candidate, developers manually: (i) check whether the candidate is a correct recommendation and, if so, (ii) effectively transform the code.

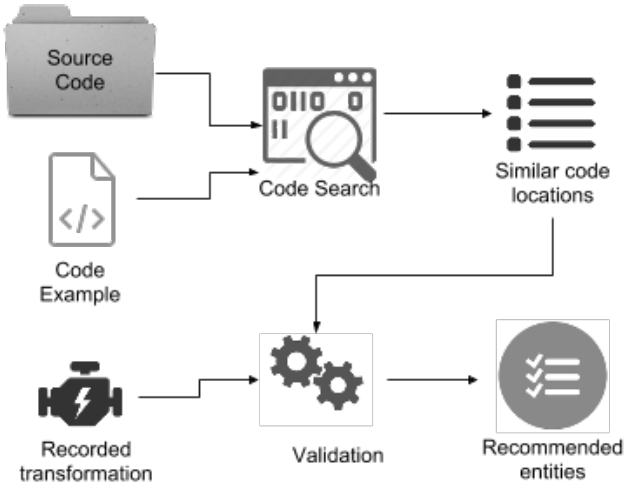

Fig. 1. Searching code with code. Our approach retrieves code entities from an example and refine the results based on a given recorded sequence of transformations.

In this paper, we evaluate three different code search approaches, using basic concepts from related work:

Structural searches for code placed in similar locations, *e.g.*, same package, superclass, etc.;

AST-based searches for code entities with similar Abstract Syntax Trees (AST); and

IR-based searches for code entities with similar vocabulary, extracted from identifiers and comments.

Prior work shows that false positives results, *i.e.*, incorrectly reported candidates, are bad for both usability and adoption of such approaches [16]–[18]. We further propose to improve precision on the code search results by trying to apply the sequences of transformations in each candidate, see Figure 1 (lower part).

The main contributions of this paper are: (i) we present three methods to automatically compute a list of candidates locations for application of a given sequence of code transformations; (ii) we check the correctness of the recommendations proposed by these approaches; and (iii) we evaluate and compare all three methods.

This paper is organized as follows: Section II provides a motivating example of the sequence of transformations we are working with. Section III provides a background of code search approaches in the literature. Section IV presents our approach to find and validate candidates for transformation. Section V presents our case study with real-world systems. Section VI presents threats to validity of our study and Section VII concludes.

II. Motivating Example

In this section, we present an illustrative example of repetitive source code transformations.

We extracted this example from JHotDraw[1], a framework for technical graphics in Java. Specifically in version 7.4.1, developers replaced the color system hierarchy to inherit the Java's AWT API.

Several modifications were done to implement this replacement, and a total of eight classes were systematically modified. Figure 2a shows the partial diff between source code of the class HSLRGBColorSystem and the new class HSLColorSpace, in terms of added (+) and removed (-) lines. Figure 2b shows the same sequence of transformations, now applied to the class CMYKNominalColorSystem.

Considering both examples in Figure 2, we observe the same sequence of transformations:

- Import class ColorSpace from Java AWT (line 3 in Figure 2a);
- Rename the class to "*ColorSpace" (line 5);
- Extend ColorSpace (line 5);
- Implement new interface NamedColorSpace (line 5);
- Implement the Singleton design pattern (lines 7–18).

We introduce the definitions and terminology that we use in the rest of this article. Not all of them are well-known or standard terminology in the context of code search and change recommendation, therefore we define them explicitly:

Definition 1 A *macro* (also known as: transformation pattern [8], systematic edit [9], or edit script [6]) is a sequence of code transformations originally composed by the developer which can be automatically performed in several code locations.

In the beginning of this section, we described the transformations that were systematically performed in JHotDraw. This description would be a starting point to create a macro for this system.

The general goal is to have tools to apply automatically macros in all the code locations where they need to be. This goal supposes that the macro can actually be replayed in different locations (not treated in this paper, see [8], [19]), then we can find all the locations that are candidate for such a replay. For example, from both cases in Figure 2, one could notice that both modified classes inherited from AbstractColorSystem before the transformations took place. One could use this information as a hint to identify other classes that need to be modified.

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

```
1   package org.jhotdraw.color;
2
    - import java.awt.*;
    - import javax.swing.*;
3   + import java.awt.color.ColorSpace;
4
    - public class HSLRGBColorSystem extends AbstractColorSystem {
5   + public class HSLColorSpace extends ColorSpace implements NamedColorSpace {
6
    -     public HSLRGBColorSystem() {
7   +     private static HSLColorSpace instance;
8
    -     }
9   +     public static HSLColorSpace getInstance() {
10  +         if (instance == null) {
11  +             instance = new HSLColorSpace();
12  +         }
13  +         return instance;
14  +     }
15  +
16  +     public HSLColorSpace() {
17  +         super(ColorSpace.TYPE_HSV, 3);
18  +     }
19
20      ...
```

(a) Partial diff between class HSLRGBColorSystem in version 7.4.1, and HSLColorSpace in version 7.5.1.

```
1   package org.jhotdraw.color;
2
3   + import java.awt.color.ColorSpace;
4
    - public class CMYKNominalColorSystem extends AbstractColorSystem {
5   + public class CMYKNominalColorSpace extends ColorSpace implements NamedColorSpace {
6
    -     public CMYKNominalColorSystem() {
7   +     private static CMYKNominalColorSpace instance;
8   +
9   +     public static CMYKNominalColorSpace getInstance() {
10  +         if (instance == null) {
11  +             instance = new CMYKNominalColorSpace();
12  +         }
13  +         return instance;
14  +     }
15  +
16  +     public CMYKNominalColorSpace() {
17  +         super(ColorSpace.TYPE_CMYK, 4);
18      }
19
20      ...
```

(b) Partial diff between CMYKNominalColorSystem in version 7.4.1, and CMYKNominalColorSpace in version 7.5.1.

Fig. 2. Code transformations performed in JHotDraw to improve the color system hierarchy.

Definition 2 An *application condition* for a macro, selects from all entities in a system (classes or methods in this paper), the ones that must be transformed by the macro.

In the concrete example, the application condition would be: all the subclasses of AbstractColorSystem in JHotDraw. However, it might not be clear for the developer whether this simple condition is correct, necessary, and/or sufficient to find all the correct locations in the system. In other systems, the condition might be more complex than just considering the hierarchy of a specific class.

Figure 2 shows that, when applying the Singleton design pattern, developers did not set the constructor accessibility to private (line 16). Also, two other classes (not shown here) did

not implement the Singleton pattern at all. Such errors and omissions could lead to inconsistent code. When it comes to bigger systems, this situation could be even worse. Santos *et al.* [2] also showed cases in the ECLIPSE IDE in which the transformations were not applied to all the opportunities that should have been transformed.

To automatically recommend source code locations that are likely candidates to apply a macro, one needs examples of such locations, given by the developers, from which the application condition can be abstracted.

Definition 3 A *code example* is a location in the source code where the macro was successfully applied. In practice this example consists in a single source code entity (a class or a method) where to start replaying the macro.

As an example, the macro on JHOTDRAW starts by adding an import declaration to the class HSLRGBColorSystem. The first location example is the one where the macro is created.[2] This first example gives initial data on the entities modified and their properties.

Definition 4 A *candidate location* is an entity in the source code that is candidate to be a *code example*. Candidate location can be wrong in two senses: (i) the macro cannot be replayed on it; or (ii) the macro could be replayed, but the developer does not wish to do so because it does not meet the, possibly informal, application conditions.

III. RELATED WORK

In this section, we discuss related work on recommending candidates for source code transformations. We do not include change impact analysis approaches in this paper; these approaches suggest which code entities should change, but they do not recommend which transformations should be performed. In this section, we classify code search approaches according to the goal of the recommendations.

Querying Similar Code Entities: Code search tools propose to locate source code based on a given query [20]–[24]. Some tools retrieve code samples from vast repositories of source code [25]–[27]. Some approaches rely on the declaration of queries using Domain Specific Languages (DSL) [13], [28]. Other approaches look for similarities between Abstract Syntax Trees (AST) [29]. Opposed to the existing code search approaches, we do not intend to provide code samples from other repositories of source code. Our goal is to find samples that require similar transformations within a specific software system. McIntyre and Walker [30] proposed REVERB, a tool that observes code edits from the developer in the ECLIPSE IDE. Then, it searches for code with dependencies in common, such as the methods being called, to find where small-scale changes should be applied. Another tool that leverages the dependency relations among various program elements is AUTOQUERY [31]. It relies on conversion of code snippets into program dependence graphs and searches using dependence-based code search technique. However, for most of the approaches in this category, the tool only provides the list of code entities that should be transformed next. The transformation effort is still required from the developer.

Recommending Refactoring Opportunities: Several approaches propose to identify code in which a refactoring must be applied. For example, Khomh *et al.* propose the detection of God classes to recommend the application of Extract Class refactoring [32]. Bavota *et al.* [33] discuss state of the art approaches that recommends the application of other refactorings described in Fowler's catalog [34]. Refactorings such as Extract Class have well defined purposes, therefore these approaches search for very specific properties in source code for recommendation. Most of the approaches rely on syntactic dependencies, such as method calls and shared variables [35]–[38]. Some other approaches rely on code metrics, *e.g.*, too many methods in a class [39]–[42]; and conceptual information retrieved from the source code vocabulary [43], [44]. In different way, Schuster *et al.* proposed refactoring at compilation time [45]. Their idea is to match a pattern-template macro with code fragments and replaces them with equivalent but simpler pattern. In our work, the transformations we found are specific to the system on which the developer is working. The rationale behind the transformations, *i.e.*, the application condition, is different for each system.

Recommending Other Recurring Transformations: LIBSYNC [46] and APIEVOLUTIONMINER [47] focused on updating the API on which a system depends. The tools extract code transformation rules from other systems that updated to the same API usages in the past, then they recommend locations in source code and transformations to replace old API calls to new ones. FIXWIZARD [48] focused on recurring bug fixes. In the code history of five real open-source projects, up to 45% of bug fixing transformations were repetitive. Based on the recurring examples the authors found, the tool also recommends both code locations and required transformations to fix the bug. PRMINER [49] focused on programming rules, *e.g.*, function b() must be called after function a(). The rules are also extracted from the code history of real software systems, in which inconsistencies in these rules led to bugs. The tool also locates code that violates these rules and recommends transformations to fix them. Similar to refactoring approaches, both API usage and bug fix approaches search for very specific properties in code, *e.g.*, API calls and known patterns that would introduce bugs. Moreover, the recommended transformations are mostly extracted from the code history of the system under analysis. In our work, the transformations are considered as occasional but repetitive. Therefore, we require support for the application of these transformations *in situ*.

Using Transformations to Find Recommendations: Automated code transformation has been proposed in the literature to provide support for developers to compose their own transformations. More recently, some tools proposed to analyze the code under transformation to find other locations in which they could be performed. Andersen *et al.* proposed patch

[2]Recording the macro supposes that the developer executes the transformations at least once on the source code.

inference techniques to derive a term-replacement from diff output [50], [51]. Thung *et al.* proposed a recommender system to offer code change candidates that enable backporting of Linux drivers code [52]. LASE [6] and CRITICS [9] rely on code examples from the developer, *e.g.*, the source code before and after the developer fixed a bug in a method. The tools calculate unmodified statements in the modified methods, to further search for methods containing similar statements by matching nodes in the AST. Both tools rely on transformations related to bug patches, which generally comprise few and very localized transformations (*e.g.*, inside a method). The transformations we study in this paper have a higher level of granularity, including from the addition of statements in a method to modifying the hierarchy of classes (*e.g.*, the example in Section II). Therefore, our validation approach, *i.e.*, trying to apply transformations to candidates, require an approach that automates more complex transformations.

IV. RECOMMENDING CODE LOCATIONS

In this section, we present our approaches to find code locations that are candidates for systematic transformation. Our approach has specific requirements:

- the source code of the entire system must be available. As a starting point, all entities in the entire source code are candidate locations;
- a macro (see Section II) has been created (optional);
- one or more code examples have been specified. The code entity on which the macro was recorded already counts as one example.

The goal of our approach is to find other locations in the source code that are similar to the code examples and, therefore, seem to require similar transformations. We tested code search approaches to recommend a list of candidate locations where to re-apply the transformations.

The code search approaches we use are inspired by approaches in the literature. First, we search for code in similar locations, *e.g.*, same package, same superclass, etc. (Section IV-A). Second, we search for code with similar structure, as represented by their ASTs (Section IV-B). Third, we search for code with similar identifiers and comments (Section IV-C). And fourth, we use the macro (when available) to refine the list of candidate locations by checking whether the transformations can be performed on them (Section IV-D).

A. Structural approach

Nguyen *et al.* [48] identified recurring bug fixes in the code history of five real open-source systems. The recurring fixes often occurred in code locations with similar properties, such as methods containing code clones, classes extending the same superclass or implementing the same interface, methods overriding the same parent method, or classes implementing the same design pattern.

Based on these findings, we implemented a location code search approach which depends on *two* or more code examples. We call this approach "Structural" because it considers basic information of where the code is located. We use concrete example from JHOTDRAW (presented in Section II) to show how the approach works. In this case, developers modified two classes with similar basic properties, as shown in Table I. Both classes belong to the same package ("pckg") and inherit from the same superclass ("sup.").

TABLE I
PROPERTIES FROM EXAMPLES IN JHOTDRAW. PROPERTIES ARE EXTRACTED FROM THE NAME OF THE CLASS ITSELF, THE NAME OF THE PACKAGE, AND THE NAME OF THE SUPERCLASS (SEE ALSO FIGURE 2).

class	HSLRGB-ColorSystem	CMYKNominal-ColorSystem
pckg	org.jhotdraw.color	org.jhotdraw.color
sup.	AbstractColorSystem	AbstractColorSystem

For classes, the properties include their package, superclass and class names. For methods, we would compute the properties of their classes (*i.e.*, package, superclass and class names), as well as the signature of the method. The structural approach then searches other entities in the system sharing the same similar properties. In the example presented in Table I, the approach searches classes in package org.jhotdraw.color which inherit from AbstractColorSystem. The name of the class is only considered when one searches for similar methods.

The search is an all-inclusive one, *i.e.*, it assumes that all classes (or methods) in the same location, whether physical (*e.g.*, package) or logical (*e.g.*, superclass), require similar transformations. This means that in the worst case, *i.e.*, no similar properties are found, the result will be all the classes in the system.

B. AST-based approach

In Section III, we mentioned tools that analyze code examples to find candidates for transformations. These tools, namely LASE [6] and CRITICS [9], look for methods that have similar statements in comparison with two or more code examples. In some sense, both tools look for instances of clones, relying on the AST of methods under analysis. Based on this idea, we implemented a code search approach which depends on a single code example (as opposed to the prior work that required two).[3]

In the concrete example with JHOTDRAW, we focus on comparing the constructor of the class HSLColorSpace for illustrating example. In a practical setting, the entire class will be analyzed because it was the first entity affected by the transformations in our example (see Section II). Assuming we want to know whether CMYKNominalColorSpace is a good candidate to replay the macro, we would try to match the source code between the constructors. Figure 3 presents a diff between these two constructors.

First, we use a greedy text-based algorithm to compute the longest common subsequence (LCS) [53] of code. When two methods have different source code, the LCS algorithm aligns what is the most common code between them. In this case, both constructors have the same call to super and the same

[3]This algorithm was inspired by a similar one in LASE.

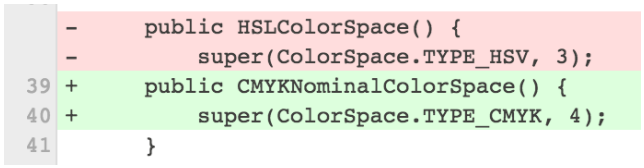

Fig. 3. Diff between constructors of classes HSLColorSpace and CMYKNominalColorSpace in JHOTDRAW.

reference to class ColorSpace. Therefore, the longest common subsequence in this case is "super(ColorSpace.".

The approach then retrieves the sequence of nodes, in the AST of both methods, that contains this subsequence. It is worth noting that the LCS algorithm is used only to retrieve the most similar code and, consequently the sequence of nodes that contains this code. From there on, we compare each node of the sequence separately. In this example, the computed sequence of nodes comprises:

- the invocation of the constructor in the superclass (super(ColorSpace, int)), which also contains
- the reference to the class ColorSpace,
- the access of an attribute in class ColorSpace (TYPE_HSV, for class HSLColorSpace), and
- the declaration of an integer value (3).

The constructor in HSLColorSpace is then *four* nodes similar to the one in CMYKNominalColorSpace. This result is used to rank the candidate set, *i.e.*, to determine which locations are more similar to the example. The top ranking locations are then considered *candidate locations*.

C. IR-based Approach

Information retrieval (IR) techniques use lexical analysis to search documents relevant to a query (the best known example would be the Google search engine). One of the most widely used searching model is called bag-of-words. Under this model, text (in our case, source code text) is represented as unordered sets of terms. Then, given a query, which is also a set of terms, the IR engine retrieves documents that contain similar terms. To account for the relative importance of a term in all documents of the corpus and in each individual document, a reasonable similarity function is the *cosine similarity* of term frequency and inverse document frequency, known as TF-IDF [54].

We implemented a search engine which indexes source code. This approach views classes (or methods) as documents and terms are retrieved from identifiers and comments. We process each term to (i) split identifiers with the camel case and underscore naming convention; (ii) remove affixes and suffixes, (ii) discard common words that do no add meaning (stop-words); and (iii) discard words that are keywords from the programming language (additional stop-words). Table II shows set of terms extracted from HSLRGBColorSystem, where the term "satur" is the result of processing the original term "saturation".

This approach works with a single code example as the previous one. This code example is processed and provided to the search engine as a query. Our IR-based approach computes a numeric score on how much each source code entity is similar to the query (the code example). Then, we rank the candidate set, *e.g.*, all the classes, according to their cosine similarity. The top ranking entities are then considered as *candidate locations*.

TABLE II
SET OF TERMS EXTRACTED FROM CLASS HSLRGBCOLORSYSTEM.

satur color count system compon green light blue base primari hslrgb

Again, consider the case on Section II in which a developer changes the class HSLRGBColorSystem. Table III shows the top ranked entities for this "query", according to the cosine similarity with HSLRGBColorSystem.

TABLE III
MOST SIMILAR CLASSES TO HSLRGBCOLORSYSTEM.

Class name	Similarity
RGBColorSystem	0.3551
HSLRYBColorSystem	0.1869
HSVRYBColorSystem	0.1399
HSVRGBColorSystem	0.0706
AbstractColorSystem	0.0417
CMYKICCColorSystem	0.0342
CMYKNominalColorSystem	0.0302
CompositeColor	0.0255
AbstractHarmonicRule	0.0228
DefaultHarmonicColorModel	0.0219

D. Replayable Approach

Given a list of candidates for transformation, it is not clear for a code search approach whether the transformations can be actually replayed in each candidate location. To validate their recommendations, we propose to use the macro (when it is available) and try to replay it. If the replaying operation fails, we assume the candidate location is a wrong one and we remove it from the list of recommendations.

Concretely, we extended MACRORECORDER [8], the tool we use in this paper to record and replay macros. The replaying operation will fail if: (i) an exception is thrown during the transformation, *i.e.*, the code entity to be transformed could not be retrieved in the candidate location; or (ii) the transformations in the macro produced code that was not compilable, consequently the tool rolls back the all the changes done by the macro.

It is worth noting that MACRORECORDER does not perform the transformations immediately on code. The tool first performs them on a model to check preconditions and display the modified code to the developer, who will ultimately accept or reject the modifications. Moreover, it is expected that the macro will not fail replaying the macro in a correct candidate.

V. EVALUATION

In this section, we evaluate the code search approaches proposed in this paper. Section V-A presents the real-world

systems under analysis and the macros that were recorded for them. We describe the metrics we used in this evaluation in Section V-B. Then, Section V-C presents how we compute candidate locations for the transformations. We evaluate structural, AST-based, and IR-based approaches in Section V-D. We evaluate our fourth approach, *i.e.*, using the macro to validate the recommendations, in Section V-E. We discuss two approaches that compute ranked recommendations, namely AST-based and IR-based approaches, in Section V-F.

A. Target Systems

Our dataset is based on sequences of transformations found in previous work [2], [8]. In total, we selected two Java systems and five Pharo[4] systems, described as follows.

Eclipse went through a considerable restructuring to integrate the OSGi technology. We focused in the user interface plugin, which was separated into five new plugins in the version 3.0.

JHotDraw is a framework for technical graphics. Its restructuring aimed at specializing the interface of color spaces (as discussed in Section II).

PetitDelphi is a parser for Delphi that has been enhanced to generate an AST from a tokenized tree. The restructuring aimed at pruning the generated AST nodes.

PetitSQL is another parser, for SQL. Its rearchitecting focused on correcting API usage of the grammar.

PackageManager is a package management system, similar to Maven, for Pharo. Its rearchitecting focused on changing the interface to access package metadata.

MooseQuery is a framework to query dependencies between entities in the FAMIX model [55]. It was restructured to be language independent (the original implementation focused on object-oriented languages).

Pillar is a language and family of tools to write and generate documentation in text, PDF, HTML pages, etc. The tests were restructured in order to provide a simpler and reusable interface.

Table IV summarizes descriptive data about the systems. Several of these systems are small, however they are written in Pharo which is a concise language. For an automation tool, the repetitiveness of the transformations is more important than the size of the system (see Threats to Validity in Section VI).

In total, we select 13 real sequences of transformations that were found in the history of their respective systems. For some systems (ECLIPSE, PETITSQL, PACKAGEMANAGER, and MOOSEQUERY) we selected more than one sequence of transformations. Table V presents descriptive data about our dataset. It describes, for each case: the number of occurrences of the repetitive task, and the number of transformations involved for each task, *i.e.*, the number of transformations in the sequence. We use this set of occurrences as our oracle.

[4]http://www.pharo.org/

TABLE IV
DESCRIPTIVE METRICS OF OUR TARGET SYSTEMS. THE FIRST TWO SYSTEMS ARE IN JAVA, THE OTHER FIVE ARE WRITTEN IN PHARO.

System (version)	Packages	Classes	KLOC
Eclipse-UI (2.1)	68	2253	185
JHotDraw (7.4.1)	39	614	59
PetitDelphi (0.210)	7	313	8
PetitSQL (0.34)	1	2	0.3
PackageManager (0.58)	2	117	2.5
MooseQuery (0.245)	2	3	0.2
Pillar (0.178)	24	278	14

TABLE V
DESCRIPTIVE METRICS OF OUR DATASET.

Sequences of Transformations	Occurrences	Number of Transformations
Eclipse I	26	4
Eclipse II	72	1
JHotDraw	9	5
PetitDelphi	21	2
PetitSQL I	6	3
PetitSQL II	98	3
PackageManager I	66	5
PackageManager II	19	3
PackageManager III	64	2
PackageManager IV	7	4
MooseQuery I	16	1
MooseQuery II	8	4
Pillar	99	4
Average	37	3

B. Evaluation Metrics

In this section, we present the metrics we use in the evaluation. Our approaches return a list of candidate locations as a result (the $Candidates$ set). For each instance of macro, the oracle set represents the code locations that were in fact modified by the developers (the $Correct$ set).

Precision is the percentage of identified candidates that are correct. Recall measures the percentage of correct locations identified by a given approach. F-measure (F_1) is the harmonic mean of precision and recall. These metrics are also described more formally as follows:

$$precision = \frac{|Correct \cap Candidates|}{|Candidates|} \quad (1)$$

$$recall = \frac{|Correct \cap Candidates|}{|Correct|} \quad (2)$$

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \quad (3)$$

Typically, a better recall comes with lower precision, and vice-versa. On one hand, recall is important because we want to avoid omissions, *i.e.*, the approach should be able to find all the correct transformation opportunities. On the other hand, as a recommendation tool for the developer, it is also important

that the approach returns as little incorrect candidates as possible (*i.e.*, a high precision). Prior work shows that incorrect candidates (false positives), are bad for both usability and adoption of such approaches [16]–[18]. Therefore, we hope for higher precision rather than higher recall.

On top of these three metrics, we added two more for AST-based and IR-based approaches. Both rank their list of candidates in decreasing order of similarity. In this case, special ranking metrics, such as the Discounted Cumulative Gain (DCG) [56] and the Precision at n (P@n) [57], were proposed by practitioners to weight correct recommendations based on their ranking position. Concretely, those metrics weight correct results near the top of the ranking higher than in lower positions. The assumption is that a developer is less likely to consider elements near the end of the list.

With DCG, see Equation 4, rel_i indicates the relevance of an entity at rank i and decreases as i augments. In Equation 5, r stands for relevant candidates retrieved at rank size of n.

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{4}$$

$$P@n = \frac{r}{n} \tag{5}$$

We compare the AST-based and IR-based approaches using these metrics. It is worth noting that DCG is not normalized. Therefore, we only compare both approaches under the same setting, *i.e.*, for the same system under analysis. Moreover, DCG is cumulative; it increases as more candidates are provided. Therefore, we can only compare both approaches under the same candidate list as well.

C. Finding Candidates for Transformation

We compute candidates locations for transformation using the approaches described in Section IV. Our approaches require some input which is retrieved from target systems as follows.

- The versions of the source code for each system are indicated in Table IV. All classes and methods of the system are used as input as specified in Section IV.
- For Pharo systems, we used the *macros* as recorded by MACRORECORDER tool [8]. This tool is also used to experiment with our "fourth" approach that proposes to filter the *candidate list* by dropping those candidates where the macro cannot be replayed (Section IV-D).
  For systems written in Java, we have no macro replaying tool for now and we could not test this combined approach.
- Our approaches require one (for AST-based and IR-based) or more (for structural) *code examples*. These code examples are selected randomly from all the actual occurrences of the macro (see again Table V).

Each approach will generate a list of candidates for transformation from which we can compute precision and recall or DCG metric according to our oracle.

D. Overall Results

Table VI presents precision and recall values for structural, AST-based and IR-based approaches. We observe that the Structural approach is performing reasonably well (especially considering its simplicity) with an average precision of 60%. Although this approach only considers package, class, and method names, all recommendations are correct (100% precision) in five (out of 13) cases. Concerning recall, the structural approach also gives good results on average (87%), and eight out of 13 cases with perfect recall. The F-measure results confirm better results for Structural approach.

Given the simplicity of the Structural filter, one could suspect that good results might be linked to a lower number of classes in the systems under analysis, however Spearman correlation shows weak correlations ($\rho = 0.36$ for precision and $\rho = -0.31$ for recall). Therefore that does not seem to be the case.

We observed overestimation with Structural approach in some cases as well. In four cases, less than 25% of candidates are correct. For example, in PETITDELPHI, developers systematically removed methods of one class which represented a specific grammar rule. The structural approach recommended all the methods of this class as *candidate locations*, whether they did represent this grammar rule or not. The result is due to the Structural approach which does not look at the AST. Similar situation occurred in PETITSQL II.

In ECLIPSE I and PACKAGEMANAGER IV, a few candidates were not found because they were contained in other package than the one from the examples. Similarly, ECLIPSE II and JHOTDRAW, some few candidates inherited from a different superclass than the one from the code examples. These cases were considered exceptions, as can be seen by the very good recall, *i.e.*, these few cases did not represent the majority of the gold standard.

The AST-based and IR-based approaches achieved an average precision around 40%. Recall average values for these two methods are 79% and 67% respectively. Regardless of the lower precision and recall, these two approaches raised important scalability issues. For example, performing code search around one thousand methods in PETITDELPHI (a medium system in our dataset), took more than 15 minutes. It turns out that comparing source code ASTs or processing identifiers takes too long to deploy such approaches into the development environment.

Summary: The Structural approach gives better results concerning both precision and recall. These results indicate that repetitive transformations usually affect similar code locations, e.g., classes in the same package, with the same superclass, or methods in the same class.

E. Replayable approach results

In this section, we add to the three approaches evaluated above the idea of validating the candidate locations by trying to replay the macro on them. We report here the results for Pharo systems because this is the context in which we have the MACRORECORDER tool to replay macros. We report only precision results because it is expected that macros

TABLE VI
STRUCTURAL, AST-BASED, AND IR-BASED RESULTS. *Occ.*: NUMBER OF OCCURRENCES OF THE ORACLE (AS SHOWN IN TABLE V); *Prec.*: PRECISION; *Rec.*: RECALL; F_1: HARMONIC MEAN OF PRECISION AND RECALL.

	Occ.	Structural			AST-based			IR-based		
Macro		Prec.	Rec.	F_1	Prec.	Rec.	F_1	Prec.	Rec.	F_1
Eclipse I	26	0.68	0.81	0.73	0.05	0.12	0.07	0.36	0.61	0.45
Eclipse II	72	1.00	0.92	0.95	0.21	0.42	0.28	0.08	0.16	0.10
JHotDraw	9	1.00	0.89	0.94	0.06	0.42	0.10	0.29	1.00	0.44
PetitDelphi	21	0.12	1.00	0.21	0.04	1.00	0.07	0.02	0.04	0.02
PetitSQL I	6	0.24	1.00	0.38	0.27	1.00	0.42	0.16	0.66	0.25
PetitSQL II	98	0.40	1.00	0.57	0.32	1.00	0.48	0.94	0.32	0.47
PackageManager I	66	1.00	1.00	1.00	0.74	1.00	0.85	0.92	0.89	0.90
PackageManager II	19	1.00	1.00	1.00	1.00	1.00	1.00	0.54	1.00	0.70
PackageManager III	64	1.00	1.00	1.00	1.00	1.00	1.00	0.87	1.00	0.93
PackageManager IV	7	0.66	0.57	0.61	0.66	0.57	0.61	0.05	1.00	0.09
MooseQuery I	16	0.41	1.00	0.58	0.34	1.00	0.50	0.19	1.00	0.31
MooseQuery II	8	0.12	0.20	0.15	0.02	0.80	0.03	0.16	0.10	0.12
Pillar	99	0.19	1.00	0.31	0.77	1.00	0.87	0.75	1.00	0.85
	Average	0.60	0.87	0.64	0.42	0.79	0.48	0.41	0.67	0.43

are configured to be replayed on correct candidate locations. Therefore, the replayable approach does not affect recall.

TABLE VII
REPLAYABLE APPROACH RESULTS. PRECISION RESULTS WITHOUT REPLAYABLE APPROACH WERE PRESENTED IN TABLE VI.

Macro	Structural +Replayable Prec.	AST-based +Replayable Prec.	IR-based +Replayable Prec.
PetitDelphi	0.12	0.04	0.02
PetitSQL I	0.85	0.85	0.57
PetitSQL II	0.40	0.32	0.94
PackageManager I	1.00	0.74	0.92
PackageManager II	1.00	1.00	0.54
PackageManager III	1.00	1.00	0.87
PackageManager IV	0.66	0.66	0.05
MooseQuery I	0.66	0.48	0.31
MooseQuery II	0.40	0.10	0.33
Pillar	0.93	0.93	0.93
Average	0.70	0.61	0.54

Table VII shows that the precision increased in four out of ten cases, with two very significant increases observed in PETITSQL I (from 24% to 85%), and PILLAR (from 19% to 93%). Similarly to PETITDELPHI and PETITSQL, the structural approach recommended all the methods in the hierarchy of document classes in PILLAR. We manually inspected each case and, although MACRORECORDER could replay the macro, the resulting code would have been incorrect. However, such behavior does not present a serious threat because the transformations are effectively performed by MACRORECORDER after inspection from the developer. Moreover, with the high precision (93%) we conclude that these cases were exceptions.

Summary: Although simple, the Structural-Replayable approach gives very good results with an average precision of 70%. *The replayable filter is also easy to implement (when there is a record-and-replay tool available) and it improves precision for all the other approaches.*

F. Combining Structural with AST-based and IR-based approaches

In particular cases, *e.g.*, PILLAR, we observed that AST-based and IR-based approaches performed better than the structural one, despite some performance issues. However, the structural approach performed better and required less resources in most of the systems. In this section, we use the list of candidates generated by the structural approach as the candidate set for AST-based and IR-based approaches (instead of the entire system). For this analysis, we focus on the results of structural approach before validation with the macro. Thus, we include the Java systems in the evaluation.

Table VIII presents DCG results. This metric measures the entire ranking. The AST-based approach performed better than the IR-based one in all but one case, *e.g.*, ECLIPSE I. Since DCG is a cumulative metric, the results in Table VIII indicate that the AST-based approach places correct recommendations in a higher position in comparison with the IR-based approach.

TABLE VIII
DCG RESULTS FOR AST-BASED AND IR-BASED APPROACHES.

Macro	AST-based	IR-based
Eclipse I	4.25	9.36
Eclipse II	7.03	2.27
JHotDraw	3.64	3.59
PetitDelphi	4.12	1.76
PetitSQL I	3.31	1.52
PetitSQL II	20.40	19.46
PackageManager I	15.61	14.78
PackageManager II	6.81	6.34
PackageManager III	15.28	13.59
PackageManager IV	2.74	1.89
MooseQuery I	5.61	0.33
MooseQuery II	2.16	0.50
Pillar	20.28	19.99
Average	11.12	9.54

Table IX presents $P@n$ results. In most of the systems, a higher precision at early positions at the ranking also implied a higher DCG. In ECLIPSE I, precision results are the same

for both approaches, however the AST-based one have lower DCG. This result is also caused by DCG's cumulative property, *i.e.*, the IR-based approach places correct recommendations (after the 20^{th} position) in a higher position in comparison with the AST-based one.

TABLE IX
AST-BASED AND IR-BASED RESULTS. *P@n*: PRECISION OF THE CANDIDATES AT THE TOP-5, 10, AND 20 POSITION IN THE RANKING.

	P@5		**P@10**		**P@20**	
Macro	AST	IR	AST	IR	AST	IR
Eclipse I	1.00	1.00	1.00	1.00	1.00	1.00
Eclipse II	1.00	0.20	0.80	0.10	0.85	0.05
JHotDraw	1.00	1.00	1.00	1.00	1.00	1.00
PetitDelphi	0.20	0.20	0.10	0.20	0.10	0.25
PetitSQL I	1.00	0.00	0.60	0.40	0.30	0.40
PetitSQL II	1.00	0.60	1.00	0.80	0.95	0.90
PackageManager I	1.00	1.00	1.00	1.00	1.00	1.00
PackageManager II	1.00	1.00	1.00	1.00	1.00	1.00
PackageManager III	1.00	1.00	1.00	1.00	1.00	1.00
PackageManager IV	0.60	0.40	0.40	0.20	0.30	0.33
MooseQuery I	0.80	0.00	0.80	0.08	0.70	0.08
MooseQuery II	0.20	0.00	0.10	0.00	0.15	0.10
Pillar	1.00	1.00	1.00	0.90	0.85	0.90
Average	0.83	0.56	0.75	0.59	0.70	0.61

However, both approaches did not improve PETITDELPHI's precision. In Section V-D, we discussed that developers removed all the methods representing a particular grammar rule. This particularity is represented by using the operator "," (a comma)[5]. The IR-based approach does not consider this operator as a term; instead, the similarity considered only the name of the method. Moreover, the AST-based approach only produce high similarity for methods with the same number of comma operators.

Other limitations appeared when the candidates have few properties in common. For example, in MOOSEQUERY II, the methods transformed by the macro have short names (*e.g.*, from and to), and they only share one return statement in common. Thus, both AST and IR similarities will be low even between correct candidates; the recommendations will then be sorted with incorrect ones. Similar cases occurred in PETITSQL I and PACKAGEMANAGER IV.

Summary: The AST-based approach produced more correct ranking in comparison with the IR-based approach. All but one of the 13 *cases were better ranked by AST similarity for top-20 recommendations.*

VI. THREATS TO VALIDITY

We now discuss threats to the validity of our study:

Internal Validity: The authors were among the developers in four of the systems under analysis. This could mean that there is a bias toward the expected searching results. Thus, one could assume that results are less significant, because we designed a searching process looking at our own source code. While the identification bias is relevant, it does not affect the essence of the study. The repetitive transformations we found occurred before our study, and therefore they were not influenced by our approach. Our participation in the development only helped us to re-discover them.

External Validity: In this study we considered 13 cases where we found repetitive transformations. It is not clear whether conclusions generalize beyond this setting. Most of the systems under analysis are small. One may argue that it is easier to find candidate locations in a smaller system. However, Santos *et al.* [2] showed that developers missed some (out of 21) candidate locations for a small system such as PETITDELPHI. In this paper, we also studied a more complex system, ECLIPSE, in which systematic transformations occurred in the past. However, the cases in PETITSQL and PACKAGEMANAGER, considered as small, seem to indicate that the size is not an issue. The macros we found in these systems repeated 98 and 66 times, respectively.

Construct Validity: In our study, we select at random code examples based on the occurrence of repetitive transformations in the past. One might argue that the code search results are highly dependent on the selection of these examples. To alleviate this threat, we executed the structural approach several times. This approach is the one that produces the preliminary candidates for the remaining approaches. We report in this paper the results in which the selection of code examples produced most candidate locations.

In our study, we use the occurrences of the macro as our oracle. However, previous work on this dataset suggested that developers might have missed some transformation opportunities. We also acknowledge this threat. In a practical setting, the list of candidates that our approach produces, either selected by the macro or ranked by AST or lexical similarity, is shown to the developer as a recommendation. Our approach still requires the developer to accept (or reject) the recommendation, either it will be a surprising recommendation or not, because some of the code locations may be found easily by manual inspection, and others may not.

VII. CONCLUSION

From time to time developers need to systematically apply sequences of source code transformations in a software. Due to the repetitive nature of this task, solutions where proposed to help create the repetitive sequence and reproduce it on different locations in a system.

In this paper, we present different solutions to identify automatically all the locations where such a repetitive sequence of transformations should be applied. We evaluated these three solutions on real cases of sequences that had been identified in existing publications [2], [8]. These examples cover seven different systems (small to large) in two OO languages (Java and Pharo), and a total of 13 examples of sequences specific to the systems where they were found.

Our approaches receive as input one or two code examples and try to generalize these examples to all possible locations on which the sequence should be applied. The results showed that a simple filtering based on "structural analysis" (*e.g.*, classes in the same package, or methods in the same class hierarchy)

[5]Pharo allows one to override operators such as "," or "=".

already produce good results in terms of recall (87% in average) and precision (70% in average). These good results can be further improved by adding an analysis on the AST.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments on early draft of the paper. This work was funded in part by the contract CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020. We also acknowledge the support of the Brazilian research agencies CAPES, CNPQ, and FAPEMIG.

REFERENCES

[1] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *28th International Conference on Automated Software Engineering*, 2013, pp. 180–190.

[2] G. Santos, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "System specific, source code transformations," in *31st International Conference on Software Maintenance and Evolution*, 2015, pp. 221–230.

[3] M. Kim and N. Meng, *Recommendation Systems in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ch. Recommending Program Transformations, pp. 421–453.

[4] J. Kim, D. Batory, and D. Dig, "Scripting parametric refactorings in java to retrofit design patterns," in *31st International Conference on Software Maintenance and Evolution*, 2015, pp. 211–220.

[5] K. Lano and S. K. Rahimi, "Optimising model-transformations using design patterns," in *1st International Conference on Model-Driven Engineering and Software Development*, 2013, pp. 77–82.

[6] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and applying systematic edits by learning from examples," in *35th International Conference on Software Engineering*, 2013, pp. 502–511.

[7] C. D. Roover and K. Inoue, "The ekeko/x program transformation tool," in *14th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 53–58.

[8] G. Santos, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "Recording and replaying system specific, source code transformations," in *15th International Working Conference on Source Code Analysis and Manipulation*, 2015, pp. 221–230.

[9] T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive code review for systematic changes," in *37th International Conference on Software Engineering*, 2015, pp. 111–122.

[10] Z. P. Fry and W. Weimer, "A human study of fault localization accuracy," in *26th International Conference on Software Maintenance*, 2010, pp. 1–10.

[11] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[12] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *11th Working Conference on Reverse Engineering*, 2004, pp. 214–223.

[13] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *25th International Conference on Automated Software Engineering*, 2010, pp. 457–466.

[14] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search," in *30th International Conference on Automated Software Engineering*, 2015, pp. 295–306.

[15] A. Armaly, J. Klaczynski, and C. McMillan, "A case study of automated feature location techniques for industrial cost estimation," in *32nd International Conference on Software Maintenance and Evolution*, 2016, pp. 1–10.

[16] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: An experimental approach," in *30th International Conference on Software Engineering*, 2008, pp. 341–350.

[17] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *35th International Conference on Software Engineering*, 2013, pp. 672–681.

[18] C. Sadowski, J. v. Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *37th International Conference on Software Engineering*, 2015, pp. 598–608.

[19] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *36th International Conference on Software Engineering*, 2014, pp. 803–813.

[20] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: An extensible local code search framework," in *20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 15:1–15:2.

[21] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based automatic query expansion for interface-driven code search," in *11th Working Conference on Mining Software Repositories*, 2014, pp. 212–221.

[22] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes, "Can the use of types and query expansion help improve large-scale code search?" in *15th International Working Conference on Source Code Analysis and Manipulation*, 2015, pp. 41–50.

[23] X. Ge, D. C. Shepherd, K. Damevski, and E. Murphy-Hill, "Design and evaluation of a multi-recommendation system for local code search," *Journal of Visual Languages & Computing*, pp. –, 2016.

[24] K. Damevski, D. Shepherd, and L. Pollock, "A field study of how developers locate features in source code," *Empirical Software Engineering*, vol. 21, no. 2, pp. 724–747, 2016.

[25] S. P. Reiss, "Semantics-based code search," in *31st International Conference on Software Engineering*, 2009, pp. 243–253.

[26] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[27] R. Sindhgatta, "Using an information retrieval system to retrieve source code samples," in *28th International Conference on Software Engineering*, 2006, pp. 905–908.

[28] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: A program query language," in *20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2005, pp. 365–383.

[29] V. Balachandran, "Query by example in large-scale code repositories," in *31st International Conference on Software Maintenance and Evolution*, 2015, pp. 467–476.

[30] M. M. McIntyre and R. J. Walker, "Assisting potentially-repetitive small-scale changes via semi-automated heuristic search," in *22nd International Conference on Automated Software Engineering*, 2007, pp. 497–500.

[31] S. Wang, D. Lo, and L. Jiang, "Autoquery: automatic construction of dependency queries for code search," *Automated Software Engineering*, vol. 23, no. 3, pp. 393–425, 2016.

[32] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *9th International Conference on Quality Software*, 2009, pp. 305–314.

[33] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, *Recommendation Systems in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ch. Recommending Refactoring Operations in Large Software Systems, pp. 387–419.

[34] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[35] G. Bavota, A. Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: An improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, 2014.

[36] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia, "Identifying method friendships to remove the feature envy bad smell (nier track)," in *33rd International Conference on Software Engineering*, 2011, pp. 820–823.

[37] L. L. Silva, K. R. Paixão, S. de Amo, and M. de Almeida Maia, "On the use of execution trace alignment for driving perfective changes," in *15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 221–230.

[38] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique," in *25th International Conference on Software Maintenance*, 2009, pp. 93–101.

[39] P. Joshi and R. K. Joshi, "Concept analysis for class cohesion," in *13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 237–240.

[40] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a java

software system," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, pp. 435–461, 2008.

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

[42] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics based refactoring," in *5th European Conference on Software Maintenance and Reengineering*, 2001, pp. 30–38.

[43] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, pp. 4:1–4:33, 2014.

[44] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *20th European Conference on Object-Oriented Programming*, 2006, pp. 404–428.

[45] C. Schuster, T. Disney, and C. Flanagan, "Macrofication: Refactoring by reverse macro expansion," in *25th European Symposium on Programming Languages and Systems*, 2016, pp. 644–671.

[46] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *25th International Conference on Object Oriented Programming Systems Languages and Applications*, 2010, pp. 302–321.

[47] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "APIEvolutionMiner: Keeping API evolution under control," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), Tool Demonstration Track*, 2014, pp. 420–424.

[48] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *32nd International Conference on Software Engineering*, 2010, pp. 315–324.

[49] Z. Li and Y. Zhou, "PR-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *10th European Software Engineering Conference Held Jointly with 13th International Symposium on Foundations of Software Engineering*, 2005, pp. 306–315.

[50] J. Andersen and J. L. Lawall, "Generic patch inference," *Automated Software Engineering*, vol. 17, no. 2, pp. 119–148, 2010.

[51] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S. C. Khoo, "Semantic patch inference," in *27th International Conference on Automated Software Engineering*, 2012, pp. 382–385.

[52] F. Thung, D. X. B. Le, D. Lo, and J. Lawall, "Recommending Code Changes for Automatic Backporting of Linux Device Drivers," in *32nd IEEE International Conference on Software Maintenance and Evolution*, 2016, pp. 1–11.

[53] E. W. Myers, "An o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[54] R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011.

[55] S. Ducasse, N. Anquetil, M. U. Bhatti, A. Cavalcante Hora, J. Laval, and T. Girba, "MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family," Research Report, 2011. [Online]. Available: https://hal.inria.fr/hal-00646884

[56] K. Järvelin and J. Kekäläinen, "IR evaluation methods for retrieving highly relevant documents," in *23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2000, pp. 41–48.

[57] N. Craswell, *Precision at n*. Boston, MA: Springer US, 2009, pp. 2127–2128.