Detecting Fragile Comments

Inderjot Kaur Ratol and Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
{inderjot.ratol@mail,martin@cs}.mcgill.ca

***Abstract*—Refactoring is a common software development practice and many simple refactorings can be performed automatically by tools. Identifier renaming is a widely performed refactoring activity. With tool support, rename refactorings can rely on the program structure to ensure correctness of the code transformation. Unfortunately, the textual references to the renamed identifier present in the unstructured comment text cannot be formally detected through the syntax of the language, and are thus fragile with respect to identifier renaming. We designed a new rule-based approach to detect fragile comments. Our approach, called Fraco, takes into account the type of identifier, its morphology, the scope of the identifier and the location of comments. We evaluated the approach by comparing its precision and recall against hand-annotated benchmarks created for six target Java systems, and compared the results against the performance of Eclipse's automated in-comment identifier replacement feature. Fraco performed with near-optimal precision and recall on most components of our evaluation data set, and generally outperformed the baseline Eclipse feature. As part of our evaluation, we also noted that more than half of the total number of identifiers in our data set had fragile comments after renaming, which further motivates the need for research on automatic comment refactoring.**

***Index Terms*—Software evolution, refactoring, source code comments, inconsistent code, fragile comments.**

I. Introduction

With the evolution and growth of a software system often comes the need for internal improvements to its structure and organization, known as *refactoring* [1]. Refactoring a system helps maintain the quality of the code and increases comprehensibility. Individual refactorings can take many forms, including renaming code elements, extracting statements into a method, and changing a method's signature. Renaming elements is, in particular, a very common type of refactoring performed to maintain a set of names that reveal the purpose of the code elements to facilitate code comprehension [2].

Many refactorings can be fully or partly automated by tools [3]. Examples include JetBrains Resharper [4] for C# and Eclipse's built-in refactoring tool [5] for Java. Such tools support code transformation by automatically changing a system's source code based on a selection from the catalog of refactorings and, when applicable, the parameterization of the refactoring. Studies show that renaming code elements is one of the most popular refactoring activities performed using automated refactoring tools [6], [7], [8].

Automating laborious refactoring tasks, such as renaming identifiers, relies heavily on encoded knowledge of the rules of the programming language to perform correct transformations. Unfortunately, references to a renamed identifier in unstructured comment text cannot be formally detected through the syntax of the language, and are thus fragile with respect to identifier renaming. We use the term *fragile comments* to refer to comments which, upon modification to the code, become inconsistent. In one study of three different projects, the authors observed that 97% of the code changes made while refactoring also needed to change the comments to maintain consistency [9]. Inconsistencies between code and comments are a problem because programmers rely on comments to understand the code and relationships between the different parts of code, its usage and to communicate amongst each other [10], [11], [12]. To avoid introducing inconsistencies between comments and code during refactoring, automatic refactoring tools need additional support to detect fragile comments and potentially update them.

Existing techniques for comment synchronization fall into two camps. In the first camp, we have simple approaches based on exact lexical search and replacement. For example, the refactoring support in Eclipse's Java development tools component [13] provides an option to search-and-replace occurrences of a renamed identifier in text strings including comments. Pure lexical approaches can be helpful in some cases, but their precision is too low to be useful in the general case. In the case of semi-structured comments (e.g., when combined with the use of in-comment tags such as `@param` in Java), text-replacement based approaches typically work much better, but these constitute a small subset of all possible comments that forms the easiest subset of identifier references to detect. In the second camp, we have specialized but domain-specific approaches that can detect inconsistencies between comments and code for programming concepts like synchronization, locking and memory allocation [14], [15], [16]. Although domain-specific techniques can achieve impressive precision, they are limited to a specialized subset of all possible types of comments.

With this paper, we advance the state-of-the-art of refactoring techniques by introducing Fraco, a *general-purpose* tool-supported approach for detecting fragile comments when renaming identifiers in a Java source code. Fraco relies on a new rule-based algorithm that takes into account the type of an identifier, its morphology, the scope of the identifier, and the location of comments. By taking into account common commenting practices and language rules, the approach avoids the limitation of naive text-replacement approaches that generate large amounts of false positives, while not including any domain-specific rules that would limit the approach to a subset of comment types.

We evaluated Fraco by comparing its precision and recall against a manually-created benchmark of six target Java systems, and compared the results against the performance of Eclipse's automated in-comment identifier replacement feature. The tool performed with near-optimal precision and recall on most components of our benchmark, and generally outperformed the baseline Eclipse feature. As part of our evaluation, we also noted that more than half of the total number of identifiers in our data set had fragile comments after renaming, which further motivates the need for research on automatic comment refactoring.

The contributions of this paper include *a)* a general and language-independent formulation of the problem of *fragile comment detection*; *b)* a tool-supported algorithm for the automatic detection of renaming-induced fragile comments in Java source code; *c)* a publicly-available[1] benchmark of fragile comments that can be used for independent research; *d)* empirical data evaluating both the proposed algorithm and a publicly-accessible tool available as part of the Eclipse IDE.

The remainder of this paper is organized as follows. In Section II we illustrate the challenges of detecting fragile source code comments. In Section III we present a precise formulation of the fragile comment detection problem that can be instantiated for different programming languages. In Section IV we describe our detection algorithm and tool for the Java language. Sections V and VI we present the design of the evaluation study and the results obtained. We describe the related work in Section VII and conclude in Section VIII.

II. Motivation

We illustrate the challenge of detecting fragile comments when renaming identifiers with three cases taken from the source code of the Checkstyle project version 7.2 [17]. We enhance the discussion with descriptions of the behavior of Eclipse's in-comment identifier replacement feature, which we will refer to as Eccore (Eclipse Comment Refactoring).

Checkstyle defines a class `Check` that is the base class for various checking rules. Here the issue is that "check" is also a very commonly used word when documenting methods, such as the one illustrated in Figure 1. If we wish to rename class `Check` to `Rule` for example, a naive text replacement feature, such as Eccore, will erroneously replace all comments that simply indicate that a method "checks" something, thus generating an outrageous number of false positives.

```java
/**
 * Check whether a class may be considered as
 * a checkstyle module. Checkstyle's modules are
 * non-abstract classes [...]
 */
private static boolean isCheckstyleModule(Class<?>
    loadedClass) {
```

Fig. 1. Example of false positive when renaming the class named `Check`.

Another challenge is to determine where to be permissive or strict with case and word morphology. For example, Checkstyle defines a public inner class `Listener`. If we wish to rename `Listener` to `Observer`, a general case-insensitive matching strategy would generate many false positives, while a case-sensitive matching strategy (such as Eccore) would miss important comments such as the use of the keyword `listener` in Figure 2. In this case the fragile comment detection technique needs to be sensitive to the scope of the comments.

```java
/** Represents a custom listener. */
public static class Listener {

  private String className; /* Name of the listener */

  /** @return the class name of listener. */
  public String getClassname() {return className; }
}
```

Fig. 2. Example of false negatives when using case-sensitive matching.

As a final example, we note that some comments can come very close to referring to an identifier without actually mentioning the identifier. Figure 3 shows a typical case of identifier re-statement in plain language. In this situation, flipping the polarity of the boolean field would require a renaming of the identifier to something like `setUseInlineTags`, which would silently render the comment inconsistent with the code. This case is also not detected by Eccore. These examples only illustrate a small subset of the situations where it is non-trivial to accurately detect fragile comments. In general, the richness and variety of commenting practices means that simple text-replacement algorithms cannot adequately cope with the problem of detecting fragile comments.

```java
/**
* Sets whether inline tags must be ignored.
* @param ignoreInlineTags whether inline tags
* must be ignored.
*/
public void setIgnoreInlineTags(boolean ignoreInlineTags){
```

Fig. 3. Example of indirect mentions of identifiers.

III. Problem Formulation

The problem of fragile comment detection is cast in the context of a *software project* which comprises a number of *program elements* and a number of *comment units*.

Technically, a **program element** is any element that can be defined in a program. In this project, we are only concerned with program elements that can be explicitly named. Consequently, in our problem formulation a program element has a corresponding *declaration* and *identifier*. The **declaration** is the program text that defines the program element, whereas the **identifier** is the part of the declaration that names the element. Declarations fall into different categories based on the type of element they define (e.g., class, method, local variable).

A **comment unit** is any bounded unit of text considered to be comments according to the syntax of a programming language. Block and line comments are the most common types of comment units. In the remainder of this paper, we refer to comment units simply as *comments*. Within a comment, a **phrase** is any coherent set of characters. Note

[1] http://www.cs.mcgill.ca/~swevo/Fraco

that, when necessary, we distinguish between a phrase and a phrase's text. This distinction is necessary when comments have multiple occurrences of some text of interest. We say that a phrase in a comment **refers to** a program element if an informed developer can determine that the phrase purposefully and specifically refers to the element. We formalize this concept as the $\mathrm{refersTo}$ relation. For example, in Figure 4, the phrase “buffer” in the comment block refers to the field `buffer` of the same class. In this case, the phrase “buffer” in the comment can be said to *refer to* the field `buffer`.

```
private Buffer buffer;

/** Reloads the existing buffer. */
public void reload() {
   if(buffer != null) { buffer.load(); }}
```

Fig. 4. Examples of the refersTo relation

Conceptually, tuples $\langle \mathrm{phrase}, \mathrm{element} \rangle$ that are members of the *refersTo* relation fall into two categories:

Lexical match: The phrase's *text* is the same as the text of the declaration's identifier, with some tolerance for minor variations (e.g., case sensitivity). It is important to note that lexical matches do not necessarily imply a $\mathrm{refersTo}$ relation because of synonymy: it is possible that a comment mentions an identifier that is shared by multiple program elements, or simply refers to a general concept after which an identifier is named (e.g., “file”).

Semantic match: The phrase semantically matches a program element if the most likely interpretation of the phrase by an expert is that it refers to the element. For pragmatic reasons, we define semantic matches as the class of matches that are semantic *without* being also lexical.

If a phrase refers to an element, we consider that the phrase is at risk of being invalidated if the identifier is renamed, and we deem it *fragile* with respect to this identifier. By extension, a comment is considered *fragile* with respect to an identifier if it contains at least one fragile phrase.

Because the $\mathrm{refersTo}$ relation requires human judgment, it can only be approximated by an automatic approach. To distinguish between phrases that truly refer to an element and phrases estimated to refer to an element, we define the relation $\mathrm{matches}$ which contains a tuple $\langle \mathrm{phrase}, \mathrm{element} \rangle$ if the corresponding algorithm estimates that the phrase refers to the element.

Our proposed approach is therefore an implementation of the $\mathrm{matches}$ relation, and its performance can be measured on a per element basis. Given an element e, let $\mathrm{refersTo(e)}$ be the set of all phrases that refer to this element, and let $\mathrm{matches(e)}$ be the set of all phrases estimated to refer to this element. The true set of fragile phrases for e is thus $\mathrm{refersTo(e)}$ and a solution instance given by an algorithm is $\mathrm{matches(e)}$. With these definitions, we easily derive the standard performance measures of precision $P(e)$ and recall $R(e)$ for an implementation of $\mathrm{matches}$:

$$P(e) \equiv \frac{|\mathrm{matches(e)} \cap \mathrm{refersTo(e)}|}{|\mathrm{matches(e)}|}$$

$$R(e) \equiv \frac{|\mathrm{matches(e)} \cap \mathrm{refersTo(e)}|}{|\mathrm{refersTo(e)}|}$$

Although precision can be measured accurately, a major obstacle to computing recall is that in the general case the extent of $\mathrm{refersTo(e)}$ is not known.

IV. Approach

Our approach is defined in terms of a given program element *declaration*. We divide the processing required to detect fragile comments into four conceptual phases. First, we *detect* all comments and *link* them to the input declaration (§IV-A). We then *filter out* some comments based on the program's scoping rules (§IV-B). The remaining comments are then *preprocessed* for textual analysis, along with the identifier of the input program element (§IV-C). Finally, we apply various *matching rules* to the resulting data (§IV-D–§IV-F).

We implemented our approach for the Java programming language, but we hypothesize that it should be straightforward to adapt it to other programming languages. To facilitate future adaptations, we describe our approach in a language-independent manner to the extent possible, and supply the Java-specific implementation details when applicable. However, we make no claim about the potential ease with which the approach can be adapted to other programming languages. We provide tool support for fragile comment detection in Java through an Eclipse plug-in called Fraco.

A. Detecting and Linking Comments

Intuitively, comments that are located meaningfully “close” to a declaration should be treated differently from general comments in the program. We capture this intuition with the concept of comment *locality*. For a given declaration, the comments in a system's source code can thus be divided into two categories: *local* and *global*.

Local Comments: To qualify as a local comment for a declaration, a comment must either be a *header* for the declaration, or be lexically located within the declaration (which we refer to as an *inner* comment). A declaration can have zero or one header comment. A comment is considered a declaration's header if it is located immediately above the declaration (without any consideration for white space in between). As for inner comments, only declarations that have a lexical *body* can have inner comments. This includes classes and methods, but excludes fields and local variables. The relation between an inner comment and its corresponding declaration is transitive, so that, for example, inner comments for a method or inner class are also considered inner comments for the declaring class.

Global Comments: For a given declaration, all comments that do not qualify as local comments automatically fall into the *global* category. For example, given a class declaration, all the comments that appear within or above other classes are considered global to the input declaration.

B. Scope-Based Filtering

In practice, the problem of searching for fragile comments can be mitigated if we observe that the scoping rules of the programming language greatly affect the likelihood that a given comment may contain a phrase that refers to a given declaration. For example, in a realistic Java code base it would be surprising to see an in-line comment in a method refer to a local variable defined in a different method. We capture this intuition by defining the function applies which takes as input a declaration and all the comments for a program, and returns the subset of the comments where the declaration can be expected to be visible.

The precise definition of the applies function is language-dependent and must take into account both the scoping rules of the language and practical knowledge of common commenting practices for this language. We implemented the function for Java based on the *Java Language Specifications, Java SE 8 Edition* [18]. The implementation of the applies function can be reduced to a lookup in a scope table, which we provide as an appendix.

C. Preprocessing

Both identifiers and comments must be preprocessed before we can apply matching rules. The preprocessing steps for identifiers and comments are similar, but for clarity we detail them separately.

Identifiers: Conceptually an identifier consists either of a single term, or of multiple terms that can be distinguished through typographical conventions. The first step in preprocessing identifiers is to split them into terms. We split the identifiers using camel casing rules, with an additional rule to preserve acronyms. After a compound identifier is split into two or more terms, we tag each term with its part-of-speech (POS) tag and identify its lemma (word root). POS tagging and lemmatization are two common natural language processing techniques used for text analysis.

A POS tag is a label that we associate with a word to indicate its function (generally in a sentence, but also in a sentence fragment, such as an identifier). For example, in the identifier `addListener`, upon tagging a single word at a time, the term `add` would be tagged as a verb and the term `listener` as a noun. We use the POS tagger of Stanford Core NLP library [19] to perform POS-tagging. Comparing lemmas instead of original words can help pave over non-essential differences such as use of the singular or plural form of a word, or different conjugations of a verb. For lemmatization, we use Stanford Core NLP library's Lemmatizer. The preprocessing step generates two dictionaries as output containing ⟨term,POS-tag⟩ tuples and ⟨term,lemma⟩ tuples respectively for each term found in the identifier.

Comments: We preprocess the comments by first splitting them into sentences using rules based on the regular-expressions we designed. Standard punctuation-based algorithms do not work well for comments because of the common presence of source-code elements that include punctuation. For the purpose of sentence-splitting, block and in-line comments are treated differently. The Javadoc block comments are split into sentences based on the list of delimiters we devised. It includes period, comma, semi-colon, angle brackets and "#". We do not split sentences using characters that are legal to use for Java identifiers (e.g., underscore) to preserve any identifiers present in the comments. We split in-line comments based on periods and commas. Sentences are then further split into *tokens* using white space as a separator. Finally, tokens that are detected (through a regular expression) to be compound code terms (such as `addFigure`) are further split as identifiers. However, we keep both the split and un-split version of the token because some matching rules work with the original term (see below). After tokenization, we remove stop words (e.g., "the", "an") from the list of tokens obtained. Finally, we apply lemmatization to the tokens in the list. For POS-tagging, the comments are tagged with the parts-of-speech on a sentence basis, i.e. a whole sentence is assigned the parts-of-speech tags at once, whereas identifier's terms are tagged individually.

D. Overview of Matching Rules

We implement the matches(e) function through a number of *matching rules*. The matching rules can be roughly organized into two categories: (mostly) *lexical* rules that target the text of phrases and identifiers, and (more) *semantic* rules that seek to match comments and identifiers that refer to the same thing despite having different spelling. We organize the rules in these two categories to facilitate the presentation, but it should be noted that most matching rules are neither purely lexical nor semantic, but involve a combination of features. Given an element e, the approach returns the union of the results of both lexical and semantic rules.

In the case of lexical matching, spelling and typographical errors are technically a potential cause of false negatives. We originally considered to mitigate this issue by including a rule that implemented fuzzy lexical matching. We experimented with a pair-wise comparison of letters in both the identifier and matching phrase. However, none of the systems in our development set contained a single instance of a reference between a phrase and an identifier that was found through fuzzy matching alone. For this reason we do not consider fuzzy matching to be necessary and left it out of the final version of the approach.

E. Lexical Matching Rules

The assumption behind lexical matching is that if a phrase has the same text as a element's identifier, it may refer to it. In practice, however, returning all instances of phrases whose text matches an identifier produces a deluge of false positives due to synonymy. Additionally, limiting the search to exact lexical matches misses cases where the name of some identifiers is transformed morphologically (e.g., used in the plural, such as "receives the `Events`" which refers to class `Event`). We therefore devised a new algorithm for lexical matching of program identifiers that takes into account the type of the identifier, its morphology, and the location of the comment containing a phrase.

Table I provides a case-based specification of the algorithm. Each cell in the table details the matching variant for one of 14 possible cases determined by the type of identifier, whether the identifier is a single or compound term, and whether the phrase to match is in a local or global comment. The matching rules are expressed as predicates using the binary operators and functions detailed in Table II.

TABLE II

OPERATORS USED IN THE RULES OF TABLE I. WHEN A PHRASE p IS USED AS INPUT, WE ASSUME THAT ITS COMMENT-CONTEXT (THE REST OF THE TEXT IN THE COMMENT) IS ALSO AVAILABLE TO THE OPERATOR (REPRESENTED AS c IN THE EXAMPLES). WHEN AN IDENTIFIER i IS USED AS INPUT, WE ASSUME THAT ITS DECLARING ELEMENT IS ALSO AVAILABLE TO THE OPERATOR (REPRESENTED AS d IN THE EXAMPLES).

Operator	Description	Positive Example
$=$	Case-sensitive match	`Tag` $=$ `Tag`
$\approx$	Case-insensitive match	`Tag` $\approx$ `tag`
$\tilde{=}$	Case sensitive match that tolerates the plural form	`Tag` $\tilde{=}$ `Tags`
$\tilde{\approx}$	Case-insensitive match that tolerates the plural form	`tag` $\tilde{\approx}$ `Tags`
noun(p)	True if the POS tag of p is a noun or proper noun (sensitive to the language model used in the POS tagger)	noun(`tag`)
paren(p)	True if p is immediately followed by the opening and closing round parenthesis having number of tokens in between the parenthesis equal to the number of declaration's arguments, if any.	paren(`read`) where $c =$ `...read(File)...` and $d =$`copy(String file)`
decl(i, p)	True 1) If p is present in the i's declaring class. OR 2) If first condition is false, check if the simple name of i's declaring class can be found in the same comment as p, without considering case.	decl(`copy`,`copy`) where $c =$ `...copy this interval...` and $d =$ `Interval`
upper(i)	True if i is all in upper case characters	upper(`SORTED`)

As it can be observed, more complex rules are necessary to determine the correct matches for simple terms (e.g., `add`, `copy`) due to their common use in program text which creates a massive amount of ambiguity. We illustrate how to use the matching rules through an example, assuming that we want to determine the comments that are fragile with respect to the declaration of a method `copy` declared in class `Interval` of some given project and that the comment in Figure 5 is under consideration. We consider two cases: first, if the comment is a local comment for the method (i.e., its header block), and then if the comment is a global comment that is not associated with the method.

```
/**
 * Make a copy of this interval.
 */
```

Fig. 5. Sample comment unit.

Considering the comment shown in Figure 5 as a local comment, we look at the $Cell_{2,3}$ of Table I, which specifies that a phrase matches the identifier if $p \approx i$. The only phrase in the comment that validates this predicate is `copy` because `copy` $\approx$ `copy`, so the rule returns the only instance of the string "copy" in the comment.

However, in the case where the comment is global, the rule of $Cell_{1,3}$ of Table I applies. The first part of the rule states that any matched phrase must be the same as the identifier (insensitive of case). However, in this case there is an additional requirement. To limit false positives, it is necessary that either the phrase be followed by parentheses, indicating the reference to a method, or the method's declaring class must be found in the comment. Thus, the rule matches the instance of the string "copy" because of its verbatim similarity with the declaration in question. But, if the comment did *not* include the suffix "of this interval", the rule would return an empty set for the comment.

F. Semantic Matching Rule

The lexical rules match identical or near-identical terms pairwise. In many situations, a set of keywords in the text of the comments can refer, as a whole, to an identifier. We designed a matching rule to capture this situation. Because this rule is intended to match "units of meaning" in comments that are likely to refer to an identifier, we refer to it as "semantic" to distinguish them from the lower-level text matching rules described in the previous section. The semantic matching rule is applicable only to the local comments.

Given all the (non-stopword) lemmas of an identifier obtained as described in IV-C, the approach looks for identical lemmas in the text of the comment unit. If all lemmas are found, the corresponding lemmas in the comment are returned as the fragile phrase.

For instance, as shown in Figure 6, after preprocessing the comment and identifier which includes the removal of stop words like "and", the sentence contains 4 token-lemmas matching the 4 term-lemmas of the identifier. The implementation of this semantic matching heuristic must take into account some of the idiosyncracies of common commenting practices that depend on the type of identifier being matched.

```
/**
 * Parses block comment as javadoc and prints its tree.
 * @param node block comment begin
 * @return string javadoc tree
 */
private String parseAndPrintJavadocTree(DetailAST node)
```

Fig. 6. Example from Checkstyle's source code.

Methods: There are two special cases for method identifiers:

(1) If a method's identifier has the word `get` as its first term, the term can be matched with both the word `get` and the tag `@return` or lemma `return` in a comment. The `@return` tag signifies the return of a value which aligns with the main functionality of the getter methods and therefore justifies the use of `@return` in place of the word `get`. For example, in the code below the phrase ⟨`last`,`node`⟩ would be returned:

```
/**
 * @return the last node that was selected,
 * or null if there are no Nodes selected.
 */
 public Node getLastNode()
```

(2) If a method's identifier starts with `is` and has a local comment of type Javadoc, for instance `isSelected`, the word `is` is not matched and instead we look for the presence of the

TABLE I

LEXICAL MATCHING RULES. p REFERS TO THE INPUT AND i TO THE IDENTIFIER OF THE ELEMENT UNDER PROCESSING. EACH NON-HEADER CELL IN THE BOTTOM TWO ROWS IS REFERRED AS $Cell_{r,c}$ WHERE r IS THE ROW NUMBER WITH VALUE EITHER 1 OR 2 AND c IS THE COLUMN NUMBER RANGING FROM 1 TO 7. THE OPERATORS ARE DEFINED IN TABLE II.

Comment Type	**Types**		**Methods**		**Fields**		**Locals**
	One term	Multiple terms	One term	Multiple terms	One term	Multiple terms	
Global Comment	$(p \overset{\sim}{=} i) \wedge (\text{noun}(p) \vee \text{paren}(p))$	$p \overset{\sim}{=} i$	$(p \approx i) \wedge (\text{decl}(i,p) \vee \text{paren}(p))$	$p = i$	$(p = i) \wedge (\text{upper}(i) \vee \text{decl}(i,p))$	$p = i$	$p = i$
Local Comment	$(p \overset{\sim}{\approx} i) \wedge (\text{noun}(p))$	$p \overset{\sim}{\approx} i$	$p \approx i$	$p = i$	$p \approx i$	$p = i$	$p = i$

word "true" or "false" immediately following the `@return` tag. This strategy is a performance-motivated proxy for verifying that the method returns a boolean. For example, in the code below the phrases ⟨member,Enum⟩ and ⟨enum,member⟩ on line 2 and 4 respectively would be returned as fragile:

```
/**
 * Checks if current AST node is member of Enum.
 * @param ast AST node
 * @return true if it is an enum member
 */
private static boolean isEnumMember(DetailAST ast)
```

Local Variables: In one special case for matching formal parameters, if the comment is a *Javadoc* comment, we match the parameter's name only against the tokens present in the text related to its `@param` tag, leaving out the rest of the text.

G. Tool Support

We developed our complete approach as an Eclipse plug-in called Fraco. The plug-in seamlessly integrates the normal Eclipse-based workflow. The detection of the matches output is triggered whenever the user renames an identifier using Eclipse's usual *Rename refactoring* feature. The results, i.e. the fragile phrases with respect to the renamed identifier, are reported as Eclipse *warning markers*, which by default appear in the *Problem View* and as annotations in the gutter (sidebar) of Eclipse's Java editor. From the Problem View, a developer can click on a *fragile comment warning* to immediately access the location of the fragile comment detected by our approach.

H. Development of the Approach

We developed the approach iteratively using three code bases as a development set. The development set included Checkstyle, the Google Guava library, and JetUML. We manually studied the relationship between the identifiers and comments to form a collection of heuristics needed to detect the fragile comments. Initially, we tried lexical matching without the additional constraints shown in Table I. As one would expect, it created more false positives than correct matches. We then integrated the concept of proximity between the identifier and a comment and introduced the distinction as *local* and *global* comments. This strategy curbed the number of false positives to a great extent but not enough to achieve practical usefulness. We then developed the identifier type-sensitive variants using POS tags, case-sensitivity and inclusion of the *parent* identifier in certain cases, which helped us achieve performance levels that aligned with practical usefulness. In the case of the semantic matching rule, the use of lemmas yielded the desired results (no false positives) on the development set.

V. EVALUATION STUDY DESIGN

The performance of Fraco can be evaluated in terms of the metrics of *precision* and *recall* (see Section III) for a sample of input *identifiers*. The design of our study comprises four main components: *a)* The *sampling* of identifiers; *b)* The creation of a *benchmark* for these identifiers; *c)* The computation of *baseline results*; *d)* The computation of *metrics*.

A. Sampling

We wanted to evaluate the approach so as to cover a variety of potential identifier renaming situations. Because of the underlying structure of programs and commenting practices, a naive random sampling approach is not appropriate in our case. First, in most software projects only a fraction of identifiers is ever mentioned in comments. By sampling randomly, any aggregated result would be heavily biased by the underlying prior distribution of identifiers in comments. Second, the proportion of different identifiers types (e.g., local variables vs. classes) is not uniform and so drawing from the general population of identifiers is likely to lead to a glut of local variables, thus degrading our ability to evaluate the performance of the approach for other identifier types. Finally, a constraint on the sampling is scalability and understandability of the underlying software, given that the resulting benchmark must be created through manual code inspection (see §V-B).

First, we selected six moderately-sized and well-commented **target systems** (see Table III). While these systems offer a diversity of application domain and open-source communities, their medium size and general-purpose application domain makes it reasonable for external investigators to inspect.

Second, we defined the **target population of program elements** as only the elements that have at least one fragile phrase. In principle this means all elements $\{e \mid \text{refersTo}(e) \neq \emptyset\}$. However, as described in §III, refersTo can only be estimated with matches, which means that our target population is partially defined in terms of the approach itself. In practice, this imperfect and unavoidable situation is mitigated by the high overlap between the output of refersTo and matches.

Third, we used a **stratified random sampling** strategy to achieve a diversity of program element types while keeping the size of the data set to a manageable level. Stratified random sampling protects against selection bias while ensuring that all classes of interest are covered in a sampled population. As part of this procedure, we randomly selected 100 elements from each of the six systems in proportion to the number of elements of each type in the target population (of elements with at least one fragile comment).

Table III shows the number of identifiers in the sample for each program element type. For each target system (row), the table indicates the total number of program elements of a given type, followed by the number (in parentheses) of program elements of this type for which at least one fragile phrase was detected. In the right column for each element type, we indicate the number of program elements of that type in the sample. For example, for Log4j we detected 244 classes, of which 116 had a non-empty result for $\mathrm{matches}$, which is 33% of all identifiers across all types $(116 + 139 + 73 + 20)$.

When piloting our evaluation on our development set, we discovered that the performance of the semantic rule contribution to the $\mathrm{matches}$ relation had very low coverage. In other words, there were relatively few cases where the semantic rule yielded results. For this reason, we evaluated the semantic rule separately using a sample that consists of all elements in each system (e.g. 244 classes of Log4j).

B. Benchmark

We created a benchmark for fragile comments that constitutes a general contribution of this work. The benchmark includes, for each element e as reported in Table III, the full set $\mathrm{refersTo}(e)$. The benchmark was created through manual inspection by the first author of this paper. For each phrase returned as the results of the $\mathrm{matches}(e)$ relation for an element e, the researcher made a binary decision as to whether the phrase referred to the element's identifier or not. The validation or invalidation of phrases as fragile is a low subjectivity task given that comments are intended for human consumption and therefore generally not ambiguous *to a human reader*. In addition, we included the false negatives discovered through the computation of the approximations of recall as described in §V-D.

C. Baseline

Although the performance of our approach can be interpreted in absolute terms, we were interested in comparing it with an existing baseline. Basic lexical matching is technically an option, but as described in §IV-H it performs so poorly that it cannot reasonably qualify as a baseline for this type of work. A more appropriate baseline is offered by Eclipse's refactoring tool. Along with refactoring the code, Eclipse's refactoring tool allows the user to select an option to replace the textual matches present in comments when renaming an identifier. Hereinafter, for convenience, we refer to this feature as Eccore ("Eclipse comment refactoring"). Eccore is only available for *type* and *field* code elements (so not available for *methods* and *local variables*). To use Eccore as a baseline when conducting the evaluation, we programmatically rename all the applicable declarations (types and fields) and consider all textual replacements to be *fragile phrases*.

D. Metrics

We evaluate our approach using the standard metrics of precision and recall. For a given program element e, we compute precision exactly a defined in Section III. In general, precision measures the degree of absence of false positives, which in our case are phrases falsely reported as fragile. In contrast, recall measures the degree of absence of false negatives. In our case, false negatives correspond to fragile phrases that remain undetected. Recall is not generally possible to compute, since to compute $\mathrm{refersTo}(e)$ one would need to manually inspect the entire source code of a system. For this reason we must resort to approximations. The Eccore feature is applicable to class and field identifiers and therefore, we use the union of the sets of true positives for both approaches as the approximation of $\mathrm{refersTo}(e)$ in the denominator of the recall equation. We denote this version of recall as Recall$^{\cup}$. As Eccore is not applicable to methods and local variables, we require an alternative strategy to estimate the recall, which, is done with a liberal equivalent defined as follows. For a given program element e, we perform a textual search for all instances of the element's identifier in comments in the *same file* as e, and identify any fragile phrase in the set. We then take the union of the set of these fragile phrases and $\mathrm{matches}(e)$ as the equivalent of $\mathrm{refersTo}(e)$. We refer to this version of recall as Recall*. In general, we can expect both of our approximations of recall to be an *upper bound* approximation of the true recall of the approach.

E. Threats and Limitations

The threats to validity and limitations of our experimental design are as follows. First, we define the population from which we draw our sample of program identifiers as a function of our approach, as described in §V-A. However, this will only introduce bias as a function of the number of identifiers for which our approach generates *a)* only false positives, or *b)* no positives in the presence of false negatives. Case a) can be precisely controlled and we verified that across all our target projects the sampling error is between 0 and 3 elements for all projects except for Spring Data Redis for which 8 elements are erroneously part of the sample. Case b) is impossible to determine reliably, but can be estimated to be very low given the high recall reported in the next section. There exists the threat of investigator bias when deciding whether a match is a true positive or not as the investigator is familiar with the working of Fraco. However, this threat is mitigated by the fact that the task is of low subjectivity, and that we released our benchmark publicly. Third, for the experiment to be useful we selected Java systems that were not only popular, but well-commented. Finally, as mentioned above, the computation of recall we designed is an approximation of a theoretical value that is not feasible to compute precisely. In consequence of these experimental conditions, the way to interpret the results in the next section is as an illustration of the potential of the approach in six distinct contexts, as opposed to a general prediction of the operational performance of the tool.

TABLE III
COMPOSITION OF THE EVALUATION SAMPLE

Project Name	**Version**	**Class Type**		**Method Type**		**Field Type**		**Local Variable Type**	
		Total (Pop.)	*Sample*	*Total (Pop.)*	*Sample*	*Total (Pop.)*	*Sample*	*Total (Pop.)*	*Sample*
Log4j	1.2.17	244 (116)	33	853 (139)	40	640 (73)	21	776 (20)	6
JUnit	4.12	218 (96)	54	587 (72)	40	144(11)	6	377 (0)	0
Joda time	2.9.6	227 (118)	32	933 (223)	54	500 (52)	12	706 (18)	2
JFlex	1.6.1	71 (36)	28	297 (66)	40	250 (34)	20	307 (26)	12
Chronicle Map	3.11.0	265 (68)	42	784 (67)	41	342 (23)	14	473 (10)	3
Spring Data Redis	1.7.8	426 (194)	54	1199 (119)	33	624 (50)	13	689 (3)	0

VI. RESULTS AND DISCUSSION

We report the results of our evaluation in three parts organized to facilitate the interpretation of the data collected. First, we present the results of the evaluation of the lexical matching rules for methods and local variables (§VI-A). These results must be interpreted in absolute terms because Eccore does not support replacement for identifiers of these types. We then report the results of the lexical matching rules for types and fields (§VI-B), which we compare against Eccore's. The results of the first two sections are based on the same sample. Since the comparison with baseline, i.e. Eccore, is done using the same sample, we annotated the precision results for both the techniques separately and later compared the results of both techniques to calculate recall. Finally, we report on the results of the semantic matching rule (§VI-C) which is based on all of the identifiers in the target systems.

A. Lexical Matching of Methods and Local Variables

Table IV shows the results of the evaluation of the lexical matching rules for the methods and local variables. We note that JUnit and Spring Data Redis do not have mentions of local variable identifiers in the comments in the sample, so performance results in these cases are not available. For methods the precision is above 95% for all evaluation systems except Spring Data Redis. However, all of the false positives in Spring Data Redis are caused by the artificial case where a comment refers to a method that is being overridden, which is not a likely scenario in practice given that renaming an overriding method changes the behavior of the code. The recall* is generally very good, with only Chronicle Map registering eight false negatives. Seven of these cases are caused by overloading or other types of ambiguity related to arguments. For example, a method named `of(first,second)` is not matched to an in-comment reference `of(...)`. The eighth case was one of confusion between a field and method name. Local variables are seldom referred to in comments. In our sample, we observe perfect recall* but equivocal precision for both JFlex and Chronicle Map. For these systems, the precision is lower due to the use of common English words like *move* as a local variable identifier, which generates natural ambiguity in the refersTo relation.

B. Lexical Matching of Types and Fields

Table V shows the results of the evaluation of lexical matching rules for types and fields. The results can be interpreted in the same way as those in Table IV except that in this case we use recall$^{\cup}$ as defined in Section III. The main observation for types is that it is relatively a simple problem to solve. Eccore shows perfect precision across all projects and Fraco only generates one false positive.

On the other hand, the recall of Fraco is superior for five out of the six projects because of the additional context-sensitive tolerance for plurals and case-insensitive matching. For Spring Data Redis, the recall$^{\cup}$ is lower due to some unexpected uses of fully-qualified names. In the case of fields, Fraco clearly dominates with perfect precision and very high recall for all but one system. In contrast, Eccore flounders in many situations. For example, if both a field and a method's parameter have the same identifier, Eccore replaces the references of the formal parameter as well. In a more egregious case, when we rename a field named `it` in Chronicle Map, Eccore generates 244 false positives in various comments including the copyright block of files.

C. Semantic Matching

Table VI show the precision of the semantic matching rule for all identifiers that produced at least one match. We do not attempt to compute the recall for semantic matching because it is not possible to reliably determine the extension of the set of true positives. The main observation we can draw from these results is that although coverage is relatively low, precision is again very high. The few false positives for methods were generated by the first special case for the return values of method identifiers (see §IV-F). However, the handful of false positives generated by this rule are largely offset by the true positives it properly captures. Finally, when we project the evaluation of the semantic matching rules back onto the sample used for lexical rules, we conclude that the number of fragile phrases detected increases by ratios between 4.4% (JFlex) and 51.8% (Chronicle Map).

VII. RELATED WORK

The work to mitigate the problem of inconsistencies between comments and code can be split into three categories: preliminary attempts at *comment-aware refactoring*, research on *detection of inconsistencies* between code and comments, and approaches to obviate the need for consistency maintenance by *generating comments automatically*.

A number of early proposals for **comment-aware refactoring** have focused on the problem of retaining the comments

TABLE IV
Results of the evaluation for identifiers of local variables and methods. The columns indicate the number of identifiers searched (Ids), the number of true positives (TP), the number of false positives (FN) the number of file-relative false negatives (FN), the precision (P), and file-relative recall (R*, described in §V-D)

Project Name	**Method Type**						**Local Variable Type**					
	Ids	*TP*	*FP*	*FN*	*P*	*R**	*Ids*	*TP*	*FP*	*FN*	*P*	*R**
Log4j	40	110	2	3	98	97	6	6	0	0	100	100
JUnit	40	102	0	1	100	99	0	0	0	0	NA	NA
Joda time	54	214	0	1	100	99	2	2	0	0	100	100
JFlex	40	60	0	0	100	100	12	10	2	0	83	100
Chronicle Map	41	139	7	8	95	94	3	2	1	0	67	100
Spring Data Redis	33	57	16	0	78	100	0	0	0	0	NA	NA

TABLE V
Results of the evaluation for identifiers of types and Fields. See Table IV for the legend.

Project	**Types**											**Fields**										
	IDs	**Eccore**					**Fraco**					**IDs**	**Eccore**					**Fraco**				
		TP	*FP*	*FN*	*P*	*R*	*TP*	*FP*	*FN*	*P*	*R*		*TP*	*FP*	*FN*	*P*	*R*	*TP*	*FP*	*FN*	*P*	*R*
Log4j	33	161	0	29	100	84	174	0	17	100	91	21	10	15	11	40	47	87	0	3	100	96
JUnit	54	372	0	65	100	85	407	1	46	99	90	6	4	0	3	100	57	7	0	0	100	100
Joda time	32	272	0	49	100	84	323	0	7	100	97	12	4	0	23	100	15	28	0	0	100	100
JFlex	28	138	0	103	100	57	223	0	29	100	88	20	28	107	0	20	100	26	0	2	100	93
Chronicle...	42	297	0	51	100	85	350	0	27	100	93	14	9	3	12	75	42	21	0	0	100	100
Spring...	54	225	0	17	100	93	223	0	52	100	81	13	11	49	6	18	65	17	4	0	80	100

TABLE VI
Results of the semantic matching based on all applicable identifiers.

Project Name	**Class Type**				**Method Type**				**Field Type**				**Local Variable Type**			
	Ids	*TP*	*FP*	*P*	*Ids*	*TP*	*FP*	*P*	*Ids*	*TP*	*FP*	*P*	*Ids*	*TP*	*FP*	*P*
Log4j	12	15	0	100	80	94	4	96	5	6	0	100	2	2	0	100
JUnit	26	39	0	100	73	85	1	98	0	0	0	NA	0	0	0	NA
Joda time	10	11	0	100	90	157	0	100	0	0	0	NA	0	0	0	NA
JFlex	9	9	0	100	41	54	1	98	2	2	0	100	2	2	0	100
Chronicle Map	17	27	1	96	9	11	0	100	1	1	0	100	2	2	0	100
Spring Data Redis	37	40	0	100	61	71	1	98	0	0	0	NA	0	0	0	NA

at their proper location in a declaration element's abstract syntax tree (AST), and to preserve their indentation [20], [21]. Existing refactoring tools, like Eclipse's, implement a similar idea to keep the comments linked to the appropriate code elements. However, these approaches neglect the possible inconsistencies introduced between the modified source code and existing comments. Eclipse [22] also comes with an in-comment text replacement feature we called *Eccore*, which supports *comment refactoring* to a certain extent. As discussed earlier, Eccore only detects and replaces identical lexical matches for two types of code-elements (types and fields), and so does not support name replacement for methods and local variables. Fraco detects fragile comments for all types of code elements. In addition to the lexical matches detected by Eccore, Fraco also detects phrases that involve multiple non-contiguous tokens in the comments.

As for **inconsistency detection**, Tan et al. proposed a technique to automatically extract program rules to detect inconsistencies related to locking mechanisms in source code [15]. They also devised an approach to extract information from comments to detect inconsistencies in source code related to specific programming concepts like memory allocation and synchronization [14]. Apart from detecting inconsistencies related to specifically targeted programming concepts, various approaches have been devised to keep the source code of methods consistent with comments. *@TComment* is a technique that detects inconsistencies between a method's parameters tolerance of null values and its related Javadoc comments [23]. This approach is however constrained to Javadoc method parameter comments. Zhou et al. devised a similar approach that detects inconsistencies between API documentation and source code by extracting documentation from Javadoc comments and performing a static analysis of the code of methods [24]. Corazza et al. devised an approach to detect the coherence between comments and a method's implementation using information retrieval concepts. There are also proposals that focus on a specific type of comments for detecting inconsistencies. For example, Sridhara has developed a tool to detect the fragility of "TODO" comments [16]. All these techniques focus on a subset of either comment types or programming concepts. In contrast, Fraco supports the detection of the possible inconsistencies produced for all possible types of comments upon renaming an identifier for any type of program element.

Automatic comment generation tools offer a different solution to the problem of code-comment consistency maintenance by relieving some of the manual work involved in the creation (and thus maintenance) of comments. *JSummarizer* generates comments for Java classes by utilizing the stereotypes of classes and methods present in the class [25], [26]. Sridhara et al. developed a tool for automatically generating comments for methods based on the code of the method [27]. *Autocomment* automatically generates comments for methods by utilizing the information mined from QA websites for code fragments similar to those in the method [28]. Guo et al. propose an approach to automatically generate comments for design patterns [29]. Li et al. designed an approach to automatically document test cases by identifying important aspects of a test case and use templates to generate the documentation automatically [30]. Although they share our goal of providing high-quality comments to developers, these approaches involve a completely different strategy in that they do not take into consideration the pre-existing relation between code and comments.

VIII. Conclusion

This paper formalized the problem of detecting fragile comments with respect to rename refactorings and proposed a novel rule-based approach for detecting fragile phrases in source code comments by taking into account the type of identifier being renamed, its morphology, the scope of the identifier and the location of the comments. By limiting the input of the analysis to general programming language features and common naming conventions, our approach can remain general-purpose and detect fragile phrases in comments of any type. Although our approach relies on language-independent principles, we developed a prototype for the Java programming language as an Eclipse plug-in called Fraco. We evaluated Fraco on a sample of 600 identifiers taken from six medium-sized systems and, when possible, compared the results against Eccore, Eclipse's identifier reference replacement feature. While detecting fragile comments for type declarations, both Fraco and Eccore performed at par. However, when renaming fields, the performance of Eccore showed dramatic unreliability, with precision varying between 20% and 100% between systems, and recall varying between 15% and 100%. In contrast, the more sophisticated rule set of Fraco showed a precision of 100% for five of the six systems, and a recall above 90% for all test systems. While Eccore currently does not even support the detection of fragile comments when renaming methods and local variables, Fraco was able to detect fragile phrases in these cases with precision and recall of 95% or above for a majority of systems.

The two clear avenues for future work in this area are *comment repair* and a broadening of the definition of *semantic matching*. Currently, our approach detects fragile phrases without repairing them. Although it is unlikely that the problem of repair is fully automatable for semantic matches, it will be interesting to explore how to differentiate matches that can be reliably repaired from those that require developer assistance. As for semantic matching, our current approach purposefully remains very close to the lexical layer because it targets the invalidation of references to specific identifiers. In the greater context of software evolution research, the problem of detecting general inconsistencies between source code and unstructured text remains a major challenge, which we could modestly approach with an expansion of the semantic rules to include additional components such as synonym analysis and entity recognition.

Appendix

Scoping Rules for the applies function

For an element declaration, the applies function can be computed by inspecting the access modifier of the declared element and its parent type (when applicable), looking up the corresponding scope, and then returning all comments linked to a declaration within the same scope. For example, a private type with no parent type maps to the *class* scope, so applies would return all the comments in the same class. The scope for all local variables (including formal parameters) is the *method* scope, which includes only comments linked to the variable's declaring method.

Element	**Parent Type**	**Scope**
Types		
public	public	global
public	default or protected	package
public	private	parent class
private	any type	parent class
protected	private	parent class
protected	public or default or protected	package
public	None	global
private	None	class
protected	None	package
default	None	package
Methods		
public	public	global
public	private	parent class
public	default or protected	package
protected	public or default or protected	package
protected	private	parent class
default	public or default or protected	package
default	private	class
private	any type	class
Fields		
public	private	parent class
public	default or protected	package
public	public	global
private	any type	class
protected	private	parent class
protected	default or protected or public	package

Acknowledgments

The authors are grateful to B. Goswami, C. Berger, the members of the McGill Swevo lab, and the reviewers for insightful comments. This work is funded by NSERC.

REFERENCES

[1] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[2] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.

[3] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[4] "Jetbrains," https://www.jetbrains.com/dotnet, verified on August 29, 2016.

[5] "Eclipse's refactoring API's documentation," http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm, verified on August 29, 2017.

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

[7] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[8] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 233–243.

[9] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 70–79.

[10] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 83–92.

[11] Y. Padioleau, L. Tan, and Y. Zhou, "Listening to programmers Taxonomies and characteristics of comments in operating system code," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 331–341.

[12] A. T. Ying, J. L. Wright, and S. Abrams, "Source code that talks: an exploration of Eclipse task comments and their implication to repository mining," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, 2005, pp. 1–5.

[13] "Eclipse documentation - current release Eclipse Neon," http://help.eclipse.org/neon/index.jsp?-topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm, verified on August 29, 2017.

[14] L. Tan, D. Yuan, and Y. Zhou, "Hotcomments: how to make program comments more useful?" in *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2005, pp. 7–9.

[15] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *Proceedings of the 21st Symposium on Operating Systems Principles*, 2007, pp. 145–158.

[16] G. Sridhara, "Automatically detecting the Up-To-Date status of ToDo comments in Java programs," in *Proceedings of the 9th India Software Engineering Conference*, 2016, pp. 16–25.

[17] "Checkstyle home page," http://checkstyle.sourceforge.net/, verified on August 29, 2017.

[18] J. Gosling, B. Joy, G. Steele, B. Bracha, and A. Buckley, *The Java® Language Specification, Java SE 8 Edition*, Feb 2015, https://docs.oracle.com/javase/specs/jls/se8/html/index.html. Verified on August 29, 2017.

[19] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit." in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2014, pp. 55–60.

[20] P. Fritzson, A. Pop, K. Norling, and M. Blom, "Comment and indentation preserving refactoring and unparsing for Modelica," in *Proceedings of the 6th International Modelica Conference*, 2008, pp. 3–4.

[21] P. Sommerlad, G. Zgraggen, T. Corbat, and L. Felber, "Retaining comments when refactoring code," in *Companion to the 23rd Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2008, pp. 653–662.

[22] "Eclipse home page," https://eclipse.org/, verified on August 29, 2017.

[23] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, 2012, pp. 260–269.

[24] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs documentation and code to detect directive defects," in *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 27–37.

[25] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 23–32.

[26] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker, "JSummarizer: An automatic generator of natural language summaries for Java classes," in *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 230–232.

[27] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the International Conference on Automated Software Engineering*, 2010, pp. 43–52.

[28] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Proceedings of the 28th International Conference on Automated Software Engineering*, 2013, pp. 562–567.

[29] J.-J. Guo, N.-L. Hsueh, W.-T. Lee, and S.-C. Hwang, "Improving software maintenance for pattern-based software development: A comment refactoring approach," in *Proceedings of the International Conference on Trustworthy Systems and their Applications*, 2014, pp. 75–79.

[30] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, "Automatically documenting unit test cases," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2016, pp. 341–352.