CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis

Juan Zhai
Purdue University, Rutgers University

Xiangzhe Xu
Nanjing University

Yu Shi
Purdue University

Guanhong Tao
Purdue University

Minxue Pan[†]
Nanjing University

Shiqing Ma
Rutgers University

Lei Xu[†]
Nanjing University

Weifeng Zhang
Nanjing University of Posts and Telecommunications

Lin Tan
Purdue University

Xiangyu Zhang
Purdue University

ABSTRACT

Code comments provide abundant information that have been leveraged to help perform various software engineering tasks, such as bug detection, specification inference, and code synthesis. However, developers are less motivated to write and update comments, making it infeasible and error-prone to leverage comments to facilitate software engineering tasks. In this paper, we propose to leverage program analysis to systematically derive, refine, and propagate comments. For example, by propagation via program analysis, comments can be passed on to code entities that are not commented such that code bugs can be detected leveraging the propagated comments. Developers usually comment on different aspects of code elements like methods, and use comments to describe various contents, such as functionalities and properties. To more effectively utilize comments, a fine-grained and elaborated taxonomy of comments and a reliable classifier to automatically categorize a comment are needed. In this paper, we build a comprehensive taxonomy and propose using program analysis to propagate comments. We develop a prototype CPC, and evaluate it on 5 projects. The evaluation results demonstrate 41573 new comments can be derived by propagation from other code locations with 88% accuracy. Among them, we can derive precise functional comments for 87 native methods that have neither existing comments nor source code. Leveraging the propagated comments, we detect 37 new bugs in open source large projects, 30 of which have been confirmed and fixed by developers, and 304 defects in existing comments (by looking at inconsistencies between existing and propagated comments), including 12 incomplete comments and 292 wrong comments. This demonstrates the effectiveness of our approach. Our user study confirms propagated comments align well with existing comments in terms of quality.

[†] Corresponding authors.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380427

ACM Reference Format:
Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan[†], Shiqing Ma, Lei Xu[†], Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380427

1 INTRODUCTION

Modern software systems usually contain a large volume of code comments [58]. Commenting code has been recognized as a good programming practice [23], which facilitates both code comprehension and software maintenance. For example, the researchers in [80, 81, 87] conducted experiments showing that code comments can help improve code readability while the researchers in [30, 36] demonstrated that code comments played a significant role in maintaining software. Moreover, code comments provide abundant information that can be leveraged to help perform a wide range of software engineering tasks, such as bug detection [69, 77–79], specification inference [17, 59, 90], testing [24, 86] and code synthesis [15, 25, 56, 62, 88]. However, as far as we know, existing work barely utilizes program analysis to systematically derive, refine, and propagate comments that provide rich semantics beyond traditional artifacts that have been used in program analysis such as types, control flow and data flow. For example, by propagation through program analysis, comments can be passed on to code entities that are not commented such that code bugs can be detected by cross-checking code with the propagated comments.

Due to the lack of standard of composing documentation, developers have substantial flexibility and they tend to use arbitrary ways to compose documentation. They usually comment on different aspects of code elements like classes, methods and variables, and use comments to describe various contents, such as summarizing the functionality, explaining the design rationale and specifying the implementation details. In addition, as comments are written in natural language, they are intrinsically ambiguous and accurate linguistic analysis is needed to acquire their exact meanings and scopes. To better understand code and more effectively propagate comments, we must first know which code elements they comment on and what kind of information they convey. That is to say, it is imperative to design a fine-grained and elaborated taxonomy of code comments and develop a reliable classifier to automatically categorize a comment.

There have been efforts in software documentation classification. Padioleau et al. [58] built a taxonomy based on meanings of comments. The work in [46] proposed a taxonomy of knowledge types in API reference documentation and used the taxonomy to assess the knowledge they contain. Based on this taxonomy, researchers in [43] developed a set of textual features to automatically categorize the knowledge. In [74], researchers studied comment categorization to provide better quantitative insights for comment quality assessment. Features are manually given for machine learning techniques to automatically classify comments. Researchers in [60, 61] first manually classified more than 2,000 code comments and then used supervised learning to achieve about 85% classification accuracy. Their taxonomies are not designed to be coupled with program analysis. It is unclear how to propagate and infer comments based on their classification.

Hence, our goal is to first build a comprehensive taxonomy from different perspectives (e.g., what and why) and different code entities (e.g., class and method), and then design a uniform analysis to enable bi-directional analysis: (1) program analysis propagates and updates comments, and (2) comments provide additional semantic hints to enrich program analysis. Multiple software tasks can benefit from the bi-directional analysis. For example, leveraging program analysis to propagate comments can provide automation support in maintaining documentation which is difficult [21] and leveraging comment analysis can help detect software bugs by checking the comment semantics against source code.

In this paper, we propose CPC, a principled and sophisticated software reasoning method that couples comment analysis and program analysis. It automatically classifies comments based on different perspectives and code entities (namely builds a comment taxonomy), and thus each comment is attributed to a code element and becomes a first-class object just like other classic objects in program analysis (e.g., variables and statements). Based on the taxonomy, CPC leverages program analysis techniques to propagate comments from one code entity to another to update, infer, and associate comments with code entities. Then CPC extracts semantics from the propagated comments to facilitate various software engineering tasks such as code bug detection. Our contributions are as follows:

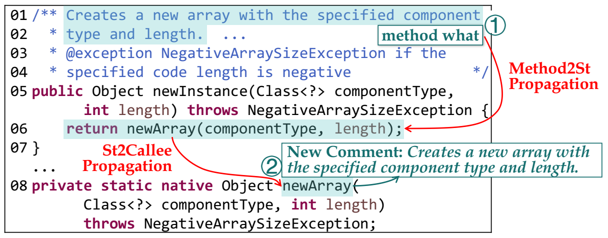
- We construct a comprehensive comment taxonomy from different perspectives with various granularity levels, and train six classifiers using three algorithms to automatically categorize comments into appropriate perspectives and granularity levels.
- We propose a novel bidirectional method of leveraging program analysis to propagate comments and leveraging comment analysis to facilitate bug detection, which achieves a seamless synergy of comment analysis and program analysis.
- We develop a prototype CPC based on the proposed idea, and evaluate it on 5 large real-world projects. The evaluation results demonstrate that 41573 new comments can be derived by propagation from other code locations with 88% accuracy. Among them, we can derive precise functional comments for 87 native methods that have neither existing comments nor source code. Our user study shows propagated comments are as useful as existing comments in helping developers.

2 MOTIVATION

Modern software provide abundant natural language (NL) comments and there is substantial existing work on analyzing NL comments and leveraging them in a wide range of software engineering applications. However, as far as we know, existing work hardly leverages program analysis techniques to derive, refine, and propagate comments systematically. Such propagated comments contain wealthy semantics beyond traditional artifacts that have been widely used in program analysis like data types. For example, by using program analysis techniques, we can pass comments on to code entities that are not commented and leverage information contained in the propagated comments to detect code bugs. Our overall goal is to achieve code-comment co-analysis: (1) program analysis propagates and updates code comments, and (2) code comments provide additional semantic hints to enrich program analysis.

Software developers tend to comment on different aspects of different code elements [58]. Comments of different perspectives entail different propagation rules. Consider the two comments “Throws IndexOutOfBoundsException if the index is out of range (index < 0 || index >= size()).” and “Shifts any subsequent elements to the left (subtracts one from their indices).” of method *remove(int index)*. The former can be propagated through data flow while the latter cannot. Suppose we have an assignment *o = list.remove(i)*, by propagating the first comment from the method definition (to the statement), we can know that if the condition *i < 0 || i >= size()* holds, the assignment statement will throw an *IndexOutOfBoundsException*, which can be used to check the code. However, the second comment describes the implementation details involving the data structure used in *remove(int index)*, which would be misleading and make no sense if propagated to the assignment. Hence, the first step towards comments propagation and inference is to build a complete taxonomy for comments.

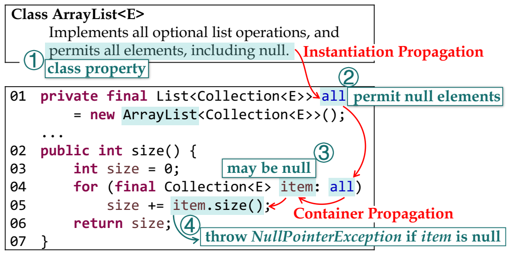
There have been efforts in software documentation classification [46, 58, 60, 61, 74]. The researchers in [58] propose a taxonomy based on the following four dimensions: comment contents, comment authors/users, comment locations (e.g., before a loop or in a macro), and comment composition time. In [46], researchers manually classify API documentation based on the knowledge types (e.g., functionalities, concepts, directives and code examples) to help humans understand and gauge the quality of API documentation. They also study the distribution of different kinds of comments. The taxonomies in [61, 74] are similar and both include categories like purpose (the functionality of the code), under development (topic related to ongoing/future development) and metadata (authors, license, etc.). They are produced to facilitate quality analysis of comments. The taxonomy in [29] is proposed to investigate developers' documentation patterns while the work [74] studies comment categorization to provide better quantitative insights about the documentation for comment quality assessment. Their taxonomies,



```
01 /** Creates a new array with the specified component
02   * type and length.  ...
03   * @exception NegativeArraySizeException if the
04   * specified code length is negative                */
05 public Object newInstance(Class<?> componentType,
        int length) throws NegativeArraySizeException {
06    return newArray(componentType, length);
07 }
  ...
08 private static native Object newArray(
        Class<?> componentType, int length)
        throws NegativeArraySizeException;
```

①
method what
Method2St Propagation
St2Callee Propagation
②
New Comment: *Creates a new array with the specified component type and length.*

Figure 2: Comment Defect Detection



Class ArrayList<E>
Implements all optional list operations, and permits all elements, including null.

```
01  private final List<Collection<E>> all
        = new ArrayList<Collection<E>>();
    ...
02  public int size() {
03      int size = 0;
04      for (final Collection<E> item: all)
05          size += item.size();
06      return size;
07  }
```

①
class property
Instantiation Propagation
②
permit null elements
③
may be null
Container Propagation
④
throw *NullPointerException* if *item* is null

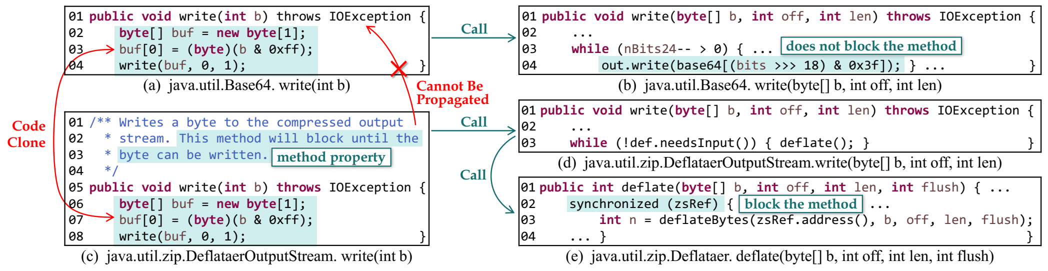
Figure 1: Code Bug Detection

however, do not distinguish comments of different code entities and are not designed to be coupled with program analysis. It is unclear where and how to propagate and infer comments based on their classification. For our purpose, we propose a comment taxonomy according to the commented subjects (e.g., classes, methods, and statements) and perspectives (e.g., what, why, and how). For each kind of comments, we develop specific rules to propagate them through program analysis. We will use 3 cases to demonstrate the benefits of propagating comments according to their categories. In a nutshell, our technique can reveal bugs in both code and comments.

Code Bug Detection. Properties are critical information embedded in comments that define intended behaviors of code elements. The top box of Fig. 1 shows the comment of class *ArrayList<E>* from *JDK*. This class permits all elements including null (denoted with green background) as items in the list. Here, the description of permitting null elements is recognized as a property comment by our technique (step ①). As a property comment, it can be propagated to the code where class *ArrayList* is actually used. The bottom box of Fig. 1 is the code snippet from *Apache Commons Collections*, where the class field *all* is instantiated as an *ArrayList* instance at line 1. The class property (permitting null elements) is hence propagated from class *ArrayList* to its instance (step ②) applying the *Instantiation Propagation* rule (detailed in Section 6.1). When variable *all* is accessed later in the program (line 4), the same property should also hold. Since variable *all* has the property of allowing null elements, each of its elements *item* is permitted to be *null* (step ③). As the *size()* method of element *item* is invoked to measure the size, with *item* being *null*, it will cause null pointer access and hence trigger a *NullPointerException*. This is a new bug detected by our technique (step ④). In total, we detect 29 such bugs which cannot be detected by existing techniques since they only use information contained in existing comments which rarely comment on local variables especially control variables only used during iteration. All the 29 bugs have been confirmed and fixed by developers. In addition, we detect another 8 bugs based on our propagated comments.

Comment Defect Detection. Comments are critical for code understanding. They also serve as instructions/manuals for (third party) developers to utilize classes and methods. Defective comments can mislead developers and even incur critical bugs. Fig. 2 demonstrates a real-world case where comments are missing for native methods. Method *newArray()* is implemented using native code (line 08), and it has two arguments *componentType* and *length*. Although comments are highly desirable here due to the black-box nature of native implementation, there is no comment for the native method, which can potentially lead to bugs (e.g., pass −1 to parameter *length*). Such native methods are implemented in other languages (e.g., C++ and assembly) where the source code, in general, is unavailable. Comments of these methods cannot be generated using existing techniques since they either summarize source code to infer comments, or analyze existing software repositories and use the comments from similar code. We showcase how our technique can address this problem using the example in Fig. 2. Firstly, there is only one statement in method *newInstance()* (line 05). Hence, we can apply rule PROPERTY-METHOD2ST (detailed in Section 6.1) to propagate the *what*-comment (lines 01-02, meaning the functionality) associated with method *newInstance()* to the statement at line 06 (step ①). Secondly, the statement at line 06 only invokes the native method *newArray()*, which satisfies the preconditions of rule PROPERTY-ST2CALLEE (detailed in Section 6.2). Thus, the comment can be further propagated from the statement at line 06 to its callee method *newArray()* (step ②). Through the propagation, a new comment can be generated for the native method, specifying the functionality of *newArray()* is to "Create a new array with the specified component type and length.". Using our technique, we are able to infer comments for 87 native methods that have neither source code nor comments in *JDK*. Note that these native methods may be invoked by many other Java methods such that their generated comments can be used to help these invocations.

Wrong Propagation Without Classification. A comment taxonomy is vital for comment propagation as different kinds of comments convey different semantic perspectives. As such, some of them cannot be directly propagated. For instance, even if two code snippets are exactly the same, propagating comment from one to the other may be problematic. Consider the two code snippets in Fig. 3(a) and Fig. 3(c). The method in Fig. 3(c) has a *property*-comment "This method will block until the byte can be written.". Although the two code snippets are syntactically identical, we cannot propagate the property comment from the method in (c) to the method in (a). This is because the method invoked at line 4 of (a) and that invoked at line 8 of (c) have different implementations which have different characteristics. Specifically, line 4 in (a) calls the write method in Fig. 3(b) that is non-blocking. In contrast, line 8 in (c) calls the write in Fig. 3(d) that is blocking, indicated by the "*synchronized*" keyword in Fig. 3(e). Therefore, it is incorrect to propagate the aforementioned property comment. However, existing techniques [84, 85] use the comment in (c) as comment for the method in (a), as they work by identifying code clones and sharing comments across all clones. In contrast, our technique does not allow propagating property comments in such cases.



```
public void write(int b) throws IOException {
    byte[] buf = new byte[1];
    buf[0] = (byte)(b & 0xff);
    write(buf, 0, 1);                              }
```

(a) java.util.Base64. write(int b)

```
public void write(byte[] b, int off, int len) throws IOException {
    ...
    while (nBits24-- > 0) { ...
        out.write(base64[(bits >>> 18) & 0x3f]); } ...            }
```

(b) java.util.Base64. write(byte[] b, int off, int len)

```
/** Writes a byte to the compressed output
  * stream. This method will block until the
  * byte can be written.
  */
public void write(int b) throws IOException {
    byte[] buf = new byte[1];
    buf[0] = (byte)(b & 0xff);
    write(buf, 0, 1);                              }
```

(c) java.util.zip.DeflataerOutputStream. write(int b)

```
public void write(byte[] b, int off, int len) throws IOException {
    ...
    while (!def.needsInput()) { deflate(); }                    }
```

(d) java.util.zip.DeflataerOutputStream.write(byte[] b, int off, int len)

```
public int deflate(byte[] b, int off, int len, int flush) { ...
    synchronized (zsRef) {  ...
        int n = deflateBytes(zsRef.address(), b, off, len, flush);
    ... }                                                        }
```

(e) java.util.zip.Deflataer. deflate(byte[] b, int off, int len, int flush)

Figure 3: Wrong Propagation Without Classification

Table 1: Examples of Software Comments Taxonomy

Entity	Perspective	Comment Example	Project/Class/Method or Field
Class	What	This class is a member of the Java Collections Framework.	JDK/ArrayList
	Why	This enables efficient processing when most tasks spawn other subtasks.	JDK/ForkJoinPool
	How-it-is-done	Resizable-array implementation of the List interface.	JDK/ArrayList
	Property	Implements all optional list operations, and permits all elements, including null.	JDK/LinkedList
	How-to-use	But using this class, one must implement only the computeNext method, and invoke the endOfData method when appropriate.	Guava/AbstractIterator
Method	What	Pushes an item onto the top of this stack.	JDK/Stack/push(E item)
	Why	It eliminates the need for explicit range operations.	JDK/ArrayList/subList(int from, int to)
	How-it-is-done	Shifts any subsequent elements to the left.	JDK/LinkedList/remove(int index)
	Property	This method is not a constant-time operation.	JDK/ConcurrentLinkedDeque/size()
	How-to-use	This method can be called only once per call to next().	JDK/Iterator/remove()
Statement	What	Make a new array of a's runtime type, but my contents.	JDK/ArrayList/toArray(T[] a)
	Why	To get better and consistent diagnostics, we call typeCheck explicitly on each element.	JDK/Collections/checkedCopyOf(Collectioncoll)
	How-it-is-done	Place indices in the center of array (that is not yet allocated).zou	JDK/WorkQueue/WorkQueue(ForkJoinPool, ForkJoinWorkerThread)
	Property	This shouldn't happen, since we are Cloneable.	JDK/ArrayList/clone()
	How-to-use	Use as random seed.	JDK/WorkQueue/registerWorker(ForkJoinWorkerThread wt)
Variable	What	The number of characters to skip.	Guava/CharStreams/SkipFully(long n)
	Why	Helps prevent entries that end up in the same segment from also ending up in the same bucket.	Guava/LocalCache/int segmentShift
	How-it-is-done	Modified on advance/split.	Guava/CharBufferSpliterator/int index
	Property	The index must be a value greater than or equal to 0.	JDK/Vector/setElementAt(E obj, int index)
	How-to-use	The collection to be iterated.	JDK/Collections/Collection iterate

3 THE TAXONOMY OF COMMENTS

As aforementioned, taxonomy is critical for comment propagation. However, existing taxonomies cannot be leveraged to facilitate comment propagation due to two main reasons. The first one is that *comments are not associated with the corresponding code entities, making it impossible to leverage program analysis to propagate comments.* For example, we need to make sure a comment is commenting on a variable before we can propagate it through a definition-use relation of the variable. The second reason is that *the taxonomies are not designed to be coupled with program analysis and comments in a category (by existing work) tend to describe multiple perspectives of a code entity such that it is unclear how to propagate such comments.* Hence we propose to construct a comment taxonomy by classifying comment texts based on two dimensions: code entity and content perspective, where code entity means elements like classes and methods and content perspective means functionalities, rationales, implementation details, etc. Such a taxonomy is vital since different comments describe different code entities from different perspectives (e.g., what, why, and how) which entail different propagation rules. To develop a comprehensive and rigorous taxonomy, we performed a *content analysis* which is a methodology for studying the contents of documents and communication artifacts [55] (Section 4). Our final taxonomy is illustrated in Table 1. The first column lists the code entities, namely, *class*, *method*, *statement*, and *variable* which are the subjects that are commonly commented by developers. For each code entity, we are interested in the following five perspectives: *what*, *why*, *how-it-is-done*, *property* and *how-to-use.*

What. The *what* perspective provides a definition or a summary of functionality of the subject and/or its interface. Critical semantics can be extracted from *what* information, such as security sensitivity, which is important for vulnerability identification. By reading such type of comments, developers can easily understand the main functionality of the corresponding code entity, without diving into (implementation) details. For example, the comment "Pushes an item onto the top of this stack" in the seventh row of Table 1 describes the main functionality of method *push(E item).*

Why. The *why* perspective explains the reason why the subject is provided or the design rationale of the subject. There are two scenarios in which *why* perspective is important. First, it helps developers understand methods whose objective is masked by complex implementation. For example, the comment "Helps prevent entries that end up in the same segment from also ending up in the same bucket" of the method *segmentShift()* conveys why we need this method, while from the implementation we can only tell it moves some objects. Second, there exist multiple methods that look similar but serve different purposes. In this case, developers often provide *why* comments to point out why these similar methods are needed and explain why they are not plain redundancy.

How-it-is-done. The *how-it-is-done* perspective describes the implementation details like the design or the work-flow of the subject. Such information is critical for developers to understand the subject, especially when the complexity is high. Detecting inconsistencies between *how-it-is-done* comments and implementation is a way to find bugs. Moreover, many program analyses avoid analyzing complex library implementation due to the entailed space and time overhead. Instead, program analysis developers often rely on *how-it-is-done* comments to synthesize (much simpler) code snippet to model library functionalities. For example, the comment "Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices)." of method *add(int, E)* implies that we can implement the functionality by manipulating an array within a loop statement (e.g., "*for (int i=size-1; i>index; i=i-1) elements[i]=elements[i-1]*"). While the original library code is usually highly optimized (and hence complex and difficult to analyse), the model code is simple and much easier to analyze.

Property. The *property* perspective asserts properties of the subject, e.g., pre-conditions/post-conditions of a function and even some statements. Pre-conditions specify the conditions that should hold in order to use the subject while post-conditions indicate the result of using the subject. Such comments are of importance as they can be used in many software engineering tasks, such as program verification, defect detection and program testing. For example, the comment "The index must be a value greater than or equal to 0." of the variable *index* (a parameter of *setElementAt(E, int)*) specifies a pre-condition $index \geq 0$ that must be satisfied for method *setElementAt(E, int)* to work properly.

How-to-use. The *how-to-use* perspective describes the expected set-up of using the subject, such as platforms and compatible versions. For example, the comment "But using this class, one must implement only the computeNext method, and invoke the endOfData method when appropriate." of the abstract class *AbstractIterator* clearly points out the required implementation in its concrete classes. These comments are important for code-comment inconsistency detection [77].

4 TAXONOMY CONSTRUCTION

In this section, we discuss how we perform a large scale study of program comments to derive the aforementioned taxonomy.

4.1 Comment Sampling

We collected a sample set of natural language comments from four frequently-used libraries, namely JDK 8 [8], Guava [7], Apache Commons Collections [1], and Joda [9]. All the four projects are open sourced. The size of the projects varies from 450 to 2500 classes and from 43 to 310 KLOC, and 30% of the lines of code are comments, which clearly indicates that documentation is not anecdotal in those projects.

Due to the lack of standard of composing documentation, developers have substantial flexibility. They tend to have arbitrary ways of composing comments and comment on different aspects of code elements [58] like methods and parameters. To ensure our study has good coverage, we performed stratified random sampling [55] to collect comments for distinct code entities: classes, methods, statements and variables. For each source file, we randomly sampled comments from each kind of code element in proportion to the number of such elements in the file. This ensured that comments of different kinds of code entities were covered. Developers usually write both single-line and multi-line comments (comment blocks), and the sentences in a multi-line comment tend to provide different types of information like what the function does or how the function is implemented. As such, we choose to use *sentence* as *the comment unit* to construct the sample set. In total, we collected 5000 comment units.

4.2 Coding Procedure

In this section, we illustrate the coding procedure [51] that we followed to construct the comment taxonomy. Coding procedure is a standard analytical process that can be utilized to define and classify a subject data set. To minimize subjectivity, we followed the default setting of the procedure [57] and made use of four coders (participants in a coding procedure). All the coders had at least four years of programming experience and were acquainted with program comments. With the intention of specifying a starting coding framework, one coder carried out a pilot study on 200 comments of the sampling set by identifying different content perspectives with the corresponding characteristics. This study brought forth the initial comment taxonomy which covered the majority of the final taxonomy. Some categories shown in Table 1 did not occur in the 200 comments and we refined the taxonomy in the later phrase. Based on the initial taxonomy, this coder held a 60-minute session to train the remaining coders either on-site or through video conferences. During the session we discussed the meaning and the examples of each category and clarified misunderstandings that arose.

The 5000 comment units were randomly and evenly assigned to all the coders, which ensured that each coder categorized comments of all the four projects. For each comment unit, the coders identify its subject (the type of code entity) and analyze its content (e.g., to identify information like the functionality). Each comment unit may target at different code entities and fall into multiple content categories. For example, the comment "Returns the head of this deque, or null if this deque is empty" of method *pollFirst()* not only describes the functionality (*what*) of this method, but also implies the implementation (*how-it-is-done*) of this method. In such cases, the coders would mark the comment with two labels. As mentioned earlier, it is possible that the coders would identify some categories that were not in the initial taxonomy. Thus the to-be-completed taxonomy was shared among coders via an online spreadsheet, which allowed each coder to add new categories to the taxonomy. Once a new category was identified and included in the taxonomy, all the other coders would be notified and they would discuss and verify. If all the coders agree on the new category, then the taxonomy would be updated to include the new one.

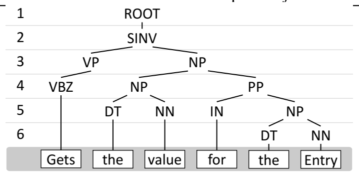
As we manually processed comments, it is inevitable to introduce subjectivity. To minimize such subjectivity, we utilized cross-verification by assigning each comment unit to two different coders. When disagreement occurs, all the coders would involve to have an open discussion to resolve it. Since to what extend two coders agreed on the categories of each comment unit is a direct measurement of both the reliability of the comment taxonomy and the quality of the labeled comments, we calculated the Kappa metric [20] to measure the agreement between two coders. The result percentage is 82.6%, representing substantial agreement [16].

5 COMMENT CLASSIFICATION

In this section, we introduce how we train a classifier, according to the taxonomy proposed in Section 3, to automatically categorize comments. We collect 5000 comments from 4 projects as mentioned in Section 4. Since our classifier works at the sentence level, the

Table 2: Features for Comments Classification

Feature	Type	Description
tokenNum	numeric	number of tokens in a cleaned comment
classNum	numeric	number of classes mentioned in a cleaned comment
tree	string	a depth first traversal of Stanford parse tree
NPNum	numeric	number of noun phrases in parse tree
VPNum	numeric	number of verb phrases in parse tree
PPNum	numeric	number of prepositional phrases in parse tree
caseNum	numeric	number of case marking relations
preconjNum	numeric	number of preconjunction relations



(a) Stanford Parse Tree

1: ROOT; 2: SINV; 3: VP NP; 4: VBZ NP PP; 5: DT NN IN NP; 6: DT NN

(b) The BFS Sequence of Stanford Parse Tree

Figure 4: Parse Tree and BFS

collected comments are split into sentences. Each comment is manually annotated with the subject being commented and with the perspective (see Table 1).

5.1 Word Embedding and Comment Cleaning

The first step towards comment classification is to train a word embedding [49, 50] based on the collected comments. Text words are represented as fixed-length vectors in word embedding and thus words close to each other in the vector space share more similarities. Existing word embeddings are trained from news articles and hardly represent the domain specific features in software. For example, the word new may be a verb (e.g., "new an object") in software comments, but not in general English. Hence we propose to train a word embedding based on our collected comments using word2vec [12]. The trained word embedding will be used to train comment classifiers introduced in Section 5.3.

Before training the embedding, we clean the collected comments to remove unnecessary information and normalize texts to acquire a more accurate and higher-quality word embedding. Mainly we perform the following four tasks: 1) Substituting class/method/variable names with three corresponding placeholders to make the embedding more general; 2) Removing stop words (common words appearing frequently [70]) which include *will, the, a, an, it, its* and *also* in our case; 3) Reducing derived words to their word stem, namely root form, by applying the porter stemming algorithm [63]. For example, the word "inserts" is transformed into "insert"; and 4) Lowercasing all the words.

5.2 Feature Extraction

To train models to classify comments, we extract eight features shown in Table 2. The first column lists the features, and the second column gives the type of each feature, namely numeric and string. The last column describes each feature. Note that all the features are automatically extracted, meaning that no human effort is required to use our trained classifiers to categorize comments.

Feature *tokenNum* is the number of tokens contained in a cleaned comment, and *classNum* is the number of classes mentioned in a comment. A comment which mentions more classes tends to have a higher probability to be an explanation of implementation details, indicating itself to be a *how-it-is-done*-comment. The remaining six features are extracted from parse trees and Stanford dependencies generated by Stanford Parser [40, 65]. The Stanford parser parses a sentence and determines Part-Of-Speech (POS) tags [41] associated with different words and phrases. Parse trees represent grammatical structure of sentences and Stanford dependencies represent grammatical relations between words in a sentence. For example, Fig. 4(a) shows the Stanford parse tree of the comment "Gets the value for the Entry" where *NP*, *VP*, *PP*, etc., are POS tags. Feature *tree* is a string representation of a parse tree which is composed of nodes that are traversed using breadth first search (BFS). For example, Fig. 4(b) is the BFS of the tree in Fig. 4(a). The features *NPNum*, *VPNum* and *PPNum* count the number of noun phrase (NP) nodes, verb phrase (VP) nodes and prepositional phrase (PP) nodes, respectively. Stanford parser also provides dependency types for each pair of adjacent words [22]. We extract the relations *caseNum* and *preconjNum* which are the number of type *case* and type *preconj* contained in a sentence. The *case* relation is used for any case-marking element which is treated as a separate syntactic word (including prepositions, postpositions, and clitic case markers) [2]. *Preconjunction* is the relation between the head of an NP and a word that appears at the beginning bracketing a conjunction and puts emphasis on it [10], such as "either", "both", "neither". The six features are utilized since based on our experiments, we observe that these six features have positive importance on the classification while the other elements contained in parse trees and dependencies have little positive importance or even have negative importance.

5.3 Algorithms and Evaluation

To train classifiers to categorize comments into different code entities and different perspectives, we leverage the following three algorithms: decision tree [64], random forest [18] and convolutional neural network (CNN) [38]. The three algorithms are frequently used to train classification models and they are proven to have high accuracy in classification [39, 74]. The decision tree algorithm and the random forest utilize the extracted features to train models while the CNN algorithm does not use any feature. As mentioned earlier, a comment may fall into different categories and thus we train multi-label classification models [83] for both perspectives and code entities. The multi-label classification problem is transformed into a set of binary classifications and each binary classification checks whether a comment can be classified into one category.

To evaluate the classifiers, we apply the standard 5-fold cross validation [42], namely we randomly select 20% comments collected in Section 4 as the testing set and the remaining comments as the training set. The performance of perspective/code entity classification is summarized in Table 3. The columns DTC, RFC and CNN respectively show the performance of decision tree, random forest and CNN. The four metrics we use are *Precision*, *Recall*, *F1 Score* and *Hamming Loss* which are calculated using sklearn metrics [11]. Specifically, *precision* measures the ability of the classifier correctly labels a comment, and it is calculated as $TP/(TP + FP)$ where TP is the total number of correctly classified comments and FP is the total number of comments that are classified into wrong categories. The metric *recall* measures the ability of the models to correctly

Table 3: Comment Classification Result

	Perspective			Code Entity		
	DTC	RFC	CNN	DTC	RFC	CNN
Precision	87.84%	87.78%	95.15%	97.39%	98.09%	89.33%
Recall	95.22%	91.39%	93.78%	99.27%	99.27%	75.28%
F1 Score	93.43%	93.17%s	94.46%	98.55%	98.90%s	81.71%
Hamming Loss	0.014583	0.014583	0.011979	0.010417	0.0007813	0.0674157

classify all the comments that belong to one category and it is calculated as $TP/(TP + FN)$ where FN is the number of comments that are not classified as one category while in fact they belong to that category. The *F1 score* is a weighted average of precision and recall and the *hamming loss* is the fraction of labels that are incorrectly predicted. For the first three metrics, the higher the better while for the hamming loss, the lower the better.

From this table, we can see that the three algorithms achieve high precision, recall and F1 score and low hamming loss in classifying comments into correct perspectives, which indicates the effectiveness of our classifiers. For the code entity classification, the decision tree and the random forest algorithms have high precision, recall and F1 score and low hamming loss. In contrast, CNN has relative lower precision, recall and F1 score and relative higher hamming loss compared with the other two algorithms. The classification of code entity is more sensitive to input features. CNNs performs their own feature abstraction, which may miss important features.

6 PROPAGATION

In this section, we will introduce the rules that are used to propagate comments based on their corresponding code entities and perspectives. These propagation rules achieve the goal of leveraging program analysis techniques to update existing comments, infer new comments and associate comments with code. We have different rules for different code entities and perspectives. Each rule is headed by its name, followed by a fraction with the nominator the conditions and the denominator the derived comment.

6.1 Property-comment Propagation

The *property*-comment propagation rules are summarized in Fig. 5(a), which involve rules for propagating comments of different levels of granularity, namely class-, method-, statement-, and variable-level.

Class-level Propagation. Rule Property-Instan defines the propagation between a class and its instantiation. That is, if a comment c is associated with a class o and a variable v instantiates class o, then the *property*-comment is propagated from class o to variable v. The expression $c[v/o]$ means the occurrence of o in c is replaced with v. For example, the *property*-comment "permits all element, including null" of the class *ArrayList* is propagated to the instance *all* declared at line 1 in Fig. 1. Rule Property-Inher specifies that if (1) a comment c is associated with superclass q, and (2) there is an inheritance relation between subclass p and superclass q, then *property*-comment c is propagated from superclass q to subclass p with the class name q (superclass) occurring in c substituted with the class name of p (subclass). Rule Property-Impl is analogous to rule Property-Inher, where the *property*-comment c associated with interface i is propagated to its implementation class o.

Method-level Propagation. Rule Property-Callee2St is applied to propagate a *property*-comment if a comment c contains properties regarding a callee method m and m is invoked by a statement s, then the *property*-comment c is propagated from the callee method m to the statement s with the formal parameters fp in c substituted with the actual parameters ap used in s. Rule Property-Method2St defines the scenario that a comment c associated with a method m is propagated to a statement s when the statement s is the only statement in m. For example, in Fig.2, the statement *return newArray(componentType, length)* at line 6 is the only statement of the method *newInstance()* and thus we can propagate the *property*-comment "@exception NegativeArraySizeException if the specified code length is negative." at lines 3-4 to the statement at line 6.

Statement-level Propagation. Rule Property-St2Callee specifies that if (1) a statement s invokes a callee method m, and (2) s has no additional operations other than returning the result of the callee m, then the *property*-comment c is propagated from the statement s to the callee method m with the actual parameters ap substituted with the formal parameters fp. Consider the aforementioned statement (line 6 in Fig.2) which invokes the method *newArray()* and does not have operations except returning the result of *newArray()*. Since the two conditions are met, we can infer a new *property*-comment for the native method *newArray()* by propagating "@exception NegativeArraySizeException if the specified code length is negative." from the statement to the callee *newArray()*. We can observe that comments of a caller method can be propagated from a callee method via the invocation statement based on Rule Property-Method2St and Rule Property-St2Callee. Rule Property-St2Method defines propagating comments from a statement s to a method m which contains s, under the condition that the set of actual variables ap contained in s is a subset of the parameters of m. Suppose that the *property*-comment c describes a property of a variable that is not a parameter of m and thus it would be inappropriate for c to be a comment of m.

Variable-level Propagation. Variable-level rules include two cases: definition-use and container-element. Rule Property-DefUse defines the case that if a comment c is associated with a variable v and v is used in code u, then the *property*-comment c is propagated from definition v to use u. Rule Property-Container specifies that if (1) a comment c is associated with container l and l has element e, then a element-related *property*-comment is propagated from container l to each element e.

6.2 What-comment Propagation

The rules to propagate *what*-comment are shown in Fig. 5(b). Similar to the rules of *property*-comment, they are also categorized based on classes, methods, statements and variables.

Class-level Propagation. Rule What-Inher is similar to Property-Inher, where a *what*-comment c associated with a superclass q is propagated to a subclass p. Rule What-Impl is also similar.

Method-level Propagation. Rule What-Callee2St specifies that if (1) a comment c is associated with a method m, and (2) there is a method invocation relation between a statement s and the callee method m, then the *what*-comment is propagated from the callee m to the statement s with the formal parameters fp in c substituted with the actual parameters ap. Rule What-Method2St denotes the propagation from a method m to a statement s inside. Two preconditions are required to be satisfied for the propagation. The first condition is that the statement s is the last statement of method

Class-level Propagation Rules:

Property-Instan $$\frac{propComment(o, c) \quad Instantiation(v, o)}{propComment(v, c[v/o])}$$

Property-Inher $$\frac{propComment(q, c) \quad inheritance(p, q)}{propComment(p, c[p/q])}$$

Property-Impl $$\frac{propComment(i, c) \quad implementation(o, i)}{propComment(o, c[o/i])}$$

Method-level Propagation Rules:

Property-Callee2St $$\frac{propComment(m, c) \quad invocation(s, m)}{propComment(s, c[ap/fp])}$$

Property-Method2St $$\frac{propComment(m, c) \quad onlySt(s, m)}{propComment(s, c)}$$

Statement-level Propagation Rules:

Property-St2Callee $$\frac{propComment(s, c) \quad invocation(s, m) \,\&\&\, noOtherOp(s)}{propComment(m, c[fp/ap])}$$

Property-St2Method $$\frac{propComment(s, c) \quad contain(m, s) \,\&\&\, ap \subseteq fp}{propComment(m, c[fp/ap])}$$

Variable-level Propagation Rules:

Property-DefUse $$\frac{propComment(v, c) \quad defUse(u, v)}{propComment(u, c)}$$

Property-Container $$\frac{propComment(l, c) \quad contain(e, l)}{propComment(e, c[e/l])}$$

(a) Property Propagation Rules

Class-level Propagation Rules:

What-Inher $$\frac{whatComment(q, c) \quad inheritance(p, q)}{whatComment(p, c[p/q])}$$

What-Impl $$\frac{whatComment(i, c) \quad implementation(o, i)}{whatComment(o, c[o/i])}$$

Method-level Propagation Rules:

What-Callee2St $$\frac{whatComment(m, c) \quad invocation(s, m)}{whatComment(s, c[ap/fp])}$$

What-Method2St $$\frac{whatComment(m, c) \quad lastSt(m, s) \,\&\&(preSts == \emptyset \,||\, preSts \subseteq exSts)}{whatComment(s, c)}$$

Statement-level Propagation Rules:

What-St2Callee $$\frac{whatComment(s, c) \quad invocation(s, m) \,\&\&\, noOtherOp(s)}{whatComment(m, c[fp/ap])}$$

What-St2Method $$\frac{whatComment(s, c) \quad lastSt(m, s) \,\&\&\, (preSts == \emptyset \,||\, preSts \subseteq exSts)}{whatComment(m, c)}$$

Variable-level Propagation Rules:

What-DefUse $$\frac{whatComment(v, c) \quad defUse(u, v)}{whatComment(u, c)}$$

(b) What Propagation Rules

Method-level Propagation Rules:

How-Clone $$\frac{howComment(m, c) \quad clone(m', m, 100\%)}{howComment(m', c)}$$

How-DiffType $$\frac{howComment(m, c) \quad clone(m', m, 90\%) \,\&\&\, diffType(m', m)}{howComment(m', c[t'/t])}$$

(c) How-it-is-done Propagation Rules

Note:

c	comment	u	code that uses a variable v
o	class	ap	actual parameters
p	subclass	fp	formal parameters
q	superclass	l	container variable
i	interface	e	element contained in the container l
m/m'	method	t/t'	type
s	statement	$preSts$	statements before the current statement
v	variable	$exSts$	exception-handling statements

Figure 5: Comment Propagation Rules

m. The second one can be either 1) there are no statements before *s*, namely *preSts* == ∅ or 2) all the previous statements are for exception handling (*preSts* ⊆ *exSts*). If the two conditions are met, the *what*-comment *c* can be propagated from the method *m* to the statement *s*. Consider the example shown in Fig. 2. The statement at line 6 is the last statement of the method *newInstance()* (lines 5-7), meaning the first condition is satisfied. Also the method body does not have statements before line 6, meaning the second condition holds. Hence the *what*-comment "Creates a new array with the specified component type and length." can be propagated from the method *newInstance()* to the statement at line 6.

Statement-level Propagation. Rule What-St2Callee describes propagation from a method invocation statement to the callee. Specifically, a statement *s* invokes a method *m* and has no additional operations other than returning the result of *m*. If a comment *c* is associated with *s*, then *c* can be propagated to *m* with the actual parameters *ap* substituted. For example, the statement at line 6 of method *newInstance()* in Fig. 2 invokes method *newArray()* and it does not involve other operations, and thus we can propagate the *what*-comment "Creates a new array with the specified component type and length." (propagated to the statement based on Rule What-Method2St) to *newArray()* at line 8. Rule What-St2Method is symmetric to rule What-Method2St and discussion is elided.

Variable-level Propagation. Variable-level propagation is defined by rule What-DefUse. That is, if variable *v* is associated with a comment *c* and there is a definition-use relation between *v* and *u*, then *c* is propagated from definition *v* to use *u*.

6.3 How-it-is-done-comment Propagation

The propagation rules for *How-it-is-done*-comment are given in Fig. 5(c) and they only involve method-level propagation. Comments can be propagated in other levels, but in practice, most *How-it-is-done*-comments are in method-level. The first rule How-Clone specifies the scenario that if (1) a *how-it-is-done*-comment *c* is associated with a method *m*, and (2) the method body of *m* is the same as the body of another method *m′*, then *c* is propagated from *m* to *m′*. The second rule How-DiffType specifies that if method *m′* is a code clone of method *m* but with different types of variables or formal parameters, then comment *c* is propagated from *m* to *m′* with the type information substituted.

7 EVALUATION

We implement a prototype CPC, leveraging the Eclipse JDT toolkit [6] and the code clone tool Nicad [68], and empirically evaluate it to address the following questions:

RQ1: How effective is CPC in propagating comments of different perspectives and code entites?
RQ2: How useful is CPC in helping developers?
RQ3: How effective is CPC in improving comments?
RQ4: How effective is CPC in detecting code bugs?

The evaluation was conducted on a machine with Intel(R) Core(TM) i7-8700K CPU (5.00GHz) and 32GB main memory. The operating system is macOS High Sierra 10.13.6, and the JDK version is 8.

Table 5: Comment Propagation Accuracy

Perspective	Accuracy		
	dist=0	dist<0.5	dist≥0.5
Property	100.00%	76.00%	85.00%
What	100.00%	71.00%	70.00%
How-it-is-done	100.00%	75.00%	70.00%

Table 4: Comment Propagation Summary

Perspe-ctive	Project					Similarity with Existing Comments					
						dist=0		dist<0.5		dist≥0.5	
		#c	#m	#ec	#pc	#cmt	%	#cmt	%	#cmt	%
Property	JDK	998	17727	21147	39274	9133	75.11%	2191	18.02%	835	6.87%
	Collections	247	2687	3151	4222	1301	73.30%	372	20.96%	102	5.75%
	Guava	518	6140	1940	8425	2718	88.28%	259	8.41%	102	3.31%
	Joda-Time	219	5011	2344	4393	1313	80.50%	111	6.81%	207	12.69%
	ApacheDB	193	3508	1898	2552	779	82.43%	57	6.03%	109	11.53%
What	JDK	628	10841	12927	5029	1368	39.66%	1550	44.94%	531	15.40%
	Collections	70	989	1472	330	105	44.30%	83	35.02%	49	20.68%
	Guava	205	2847	1347	1294	419	49.47%	333	39.31%	95	11.22%
	Joda-Time	83	1725	1949	885	237	29.40%	325	40.32%	244	30.27%
	ApacheDB	78	1426	1316	682	169	29.14%	366	63.10%	45	7.76%
How-it-is-done	JDK	261	974	1392	16285	15516	96.72%	394	2.46%	133	0.83%
	Collections	41	98	100	113	53	67.09%	22	27.85%	4	5.06%
	Guava	20	33	31	127	108	85.71%	16	12.70%	2	1.59%
	Joda-Time	15	22	29	130	32	35.20%	37	29.13%	58	45.67%
	ApacheDB	180	285	254	519	421	84.04%	58	7.39%	22	4.39%

7.1 Effectiveness in Comments Propagation

To answer RQ1, we propagate *property*-comments, *what*-comments and *how-it-is-done*-comments in five projects. The results are summarized in Table 4, which presents the comment perspective (column 1), the projects (column 2), the number of classes/methods whose comments are propagated (columns #c and #m), the number of existing comments/propagated comments (columns #ec and #pc), the similarity between an existing comment and an propagated comment (columns 7-12). Note that the comparison is conducted only when there is an existing comment. The similarity is measured using the Word Mover's Distance (WMD) algorithm [44]. A zero distance means the existing comment and the propagated comment are literally the same. If the distance is between 0 and 0.5, it means two comments are literally similar and if the distance is more than 0.5, it means two comments are literally different. The longest distance is 10. For each distance range, the columns #cmt and % present the number of propagated comments and the ratio between #cmt and the total number of propagated comment (column #pc).

From Table 4, we make a few observations. Firstly, the number of propagated comments is larger than that of existing comments since one comment may be propagated to different places. Secondly, the number of propagated *property*-comments is much larger than that of *what*-comments and *how-it-is-done*-comments. This is due to the fact that developers tend to comment on exception-related behaviors and one method may contain several different exception behaviors, and these exception-related comments belong to *property* comments. Thirdly, the number of propagated *how-it-is-done*-comments is relatively smaller due to two factors. The first one is that the number of code clones is small and the second one is that fewer comments are about implementation details (*how-it-is-done*). Fourthly, the percentage of propagated *property*-comments that are literally the same with existing comments (0 distance) is higher than the other two perspectives (on average 80% vs 56%). This is mainly because *property*-comments have limited contents with relatively fixed sentence patterns while the other comments describe various aspects and tend to be depicted using different sentences to express the same semantic. Fifthly, more than 88% propagated comments are literally similar with existing comments (distance less than 0.5), which indicates our propagation technique is feasible and efficient in manipulating comments as first-class objects. Lastly, the percentage of comments with distance larger than 0.5 in the project *Joda-Time* is much higher than the others. By checking the comments, we found that there are ten code snippets which share the same code and one of them has a comment that is literally different from the comments of the remaining ones, and there are some other similar cases. Such cases contribute a lot to the high percentage given the small number of propagated comments.

To further answer RQ1, we manually measure the accuracy of propagated comments for different distance ranges, summarized in Table 5. The first column gives the perspective of comments and the remaining columns show the accuracies of different distance ranges. Due to the large number of propagated comments, we cannot manually check all of them. Instead, for each distance range, we randomly sampled 500 comments of each perspective and manually checked whether the propagated comments are correct or not. If a propagated comment is inconsistent with the source code, it is considered as false positive.

Table 5 shows that we achieve 100% accuracy when the distance is 0 and an average of 75% accuracy when the distance is larger than 0. This demonstrates that our propagation technique is effective in inferring comments. Note that even though the distance is larger than 0 or even larger than 0.5, it does not mean the propagated comments are incorrect since the same semantics can be expressed using different sentences. For example, the comment "Returns the node; or null if not found" is propagated to method *remove()* of class *ConcurrentSkipListMap* and this method has an existing comment "Returns the previous value associated with the specified key; or null if there was no mapping for the key.". The two comments are literally quite different, but they have the same semantics.

7.2 Usefulness in Helping Developers

To answer RQ2, we conducted a user study involving 14 users (6 graduate students and 8 developers from industry) to participate. We randomly selected 80 code entities that have both existing comments and propagated comments (with a total of 160 comments). The generated comments are propagated from other places and must be syntactically different from the existing ones. They are mainly from Commons Collections, JDK, and Guava, and have even coverage for the three comment types. To diversify our selection, these code entities are selected from different source files. To avoid bias, we mix the propagated comments and the existing comments, and thus the users are unaware of whether a comment is propagated or existing. For each comment, we provide the corresponding code, and ask users to evaluate the comments from the following three perspectives: **Meaningfulness** (is a comment of high quality in helping developers understand code), **Consistency** (is a comment consistent with code), and **Naturalness** (does a comment effectively convey information as a natural language sentence).

The users are asked to evaluate each comment based on the widely-adopted five-point Likert scale [45], and the scores 1,2,3,4,5 separately represent strongly disagree, disagree, neither agree nor disagree, agree and strongly agree. Note that the numerical results of these questions are not important as they are dependent on the

Table 6: Comment Propagation Improvement

Perspective / Project	Property			What			How-it-is-done		
	#N	#I	#W	#N	#I	#W	#N	#I	#W
JDK	26862	11	243	1580	1	0	242	n.a.	0
Collections	2404	11	42	93	0	0	34	n.a.	0
Guava	5344	0	2	447	0	0	1	n.a.	0
Joda-Time	2757	0	5	79	0	0	3	n.a.	0
ApacheDB	1607	0	0	102	0	0	18	n.a.	0

Table 7: Code Bug Detection

Project	Version	#Bugs	Buggy Method	Confirmed
Collections	4.2	29	CompositeCollection.iterator()	Yes
			CompositeMap.removeComposited(final Map<K, V>)	Yes
			. . .	
Guava	28.0	6	Throwables.getRootCause(Throwable)	No
			. . .	
ApacheDB	3.2	2	Utilities.printClasspath()	Yes
			ConsoleFileOutput.getDirectory()	No

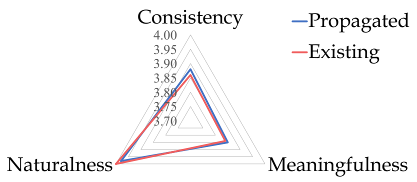


Figure 6: User Study Comparison Result

quality of the original comments (recall our propagated comments also originate from existing comments). Instead, the comparative results of the two kinds of comments are important. Fig. 6 shows the comparison results between propagated comments (blue) and existing comments (red): 3.88 vs 3.86 for consistency, 3.85 vs 3.84 for meaningfulness, and 3.98 vs 4.00 for naturalness. Overall, the results indicate propagated comments align well with existing ones in terms of quality. Further inspection shows that the slightly worse results regarding naturalness are due to our sampling bias: the propagated comments are 16.7% shorter than the existing comments and the users seem to consider longer comments are more natural.

7.3 Effectiveness in Improving Comments

To answer RQ3, we evaluate the effectiveness of our comment propagation in three aspects: 1) inferring new comments for code entities that do not have existing comments; 2) identifying incomplete comments which may be misleading for users or developers; and 3) detecting wrong comments that might lead to bugs. The result is shown in Table 6, and the columns #N, #I and #W present the number of new comments, incomplete comments and wrong comments, respectively. Here a new comment means the code entity where the comment is propagated to does not have any comment before. Note that we do not give the number of incomplete *how-it-is-done*-comments since it is unnecessary and impractical to comment all the implementation details.

Based on the number of new comments shown in Table 6 and the accuracy in Table 5, we can see that our technique can effectively generate new comments, which can be further used to facilitate understanding and maintain documentation. By manually checking some of the newly-generated comments, we find that many comments describe exceptional behaviors including the type of exception and the corresponding exception-trigger condition. Such comments are usually considered very important since a majority of bugs are caused by triggering exceptions. For example, we can generate the new comment "Throws IllegalArgumentException if the size is less than 1" for constructor *CircularFifoQueue(Collection)*, which conveys that the parameter *collection* should have a size larger than 0. Without such a comment, bugs are easily introduced since very few methods have such requirements and developers are insensitive to them. Moreover, among the new comments, precise functional comments are inferred for 87 native methods that have neither comments nor source code. These comments can serve as manuals for developers to leverage these native methods.

We also identify 11 incomplete comments which can be seen as inconsistencies between comments and code. Our propagated comments can be used to complement existing comments to address such inconsistencies and reduce the risk of introducing bugs when the code is used. For example, "Returns true if this list changed as a result of the call." is the existing comment of method *addAll()* in class *RoleList*, and "Returns true if the RoleList specified is null." is one propagated comment of the method. By analyzing the code, we can see the propagated comment is correct, meaning the existing one is incomplete. In addition, we detect many wrong comments that would be misleading and even lead to bugs. For example, in the project *Apache Commons Collection*, we generate the propagated comment "throws IndexOutOfBoundsException if index <0 or index >= size()" which is inconsistent with the existing comment "throws IndexOutOfBoundsException if index <0 or index > size()" of method *setIterator* in class *CollatingIterator*. We confirmed our propagated comment is correct, and *developers also confirmed this and corrected the existing wrong comment* [3].

7.4 Effectiveness in Bug Detection

To answer RQ4, we write a script to extract code whose propagated comments describe behaviors related to *NullPointerException* and *IndexOutOfBoundsException* based on buggy patterns. For *NullPointerException*, the buggy pattern is the code that does not check whether the return value of a method (whose comments state a null value may be returned) is null before dereferencing it. For *IndexOutOfBoundsException*, the buggy pattern is the code that does not check if the returned value of a method (whose comments state -1 may be returned) is -1 before using it to access an array.

Table 7 reports the bug detection results including, from left to right, the project, the project version, the number of detected bugs, the buggy method and whether the reported bug is confirmed. Due to the space limitations, not all the bugs are presented in the table. In total, our script reports 57 bugs. By manually checking them, we believe 37 of them are true bugs. We have reported the 37 bugs to the developers, among which, *30 bugs have already been confirmed and fixed by developers* [4, 5], while the remaining ones await confirmation. For the false positives, the main reason is that our analysis script is not context-sensitive and hence cannot identify cases in which users will never pass parameters that trigger the function to return null or -1. It is a limitation of our scanner, not comment derivation.

8 THREATS TO VALIDITY

The threat to construct validity is the bias that may be introduced during the manual labeling of comments (Section 4). To mitigate this threat, each comment was categorized by two developers independently, and a third developer would manually resolve all cases when two developers disagreed. We assessed the labeling reliability by measuring the inter-coder agreement (Section 4). In the future, we will further minimize the threat by inviting more developers to categorize comments. The threat to internal validity is the potential overfitting problem of the machine learning algorithms. To minimize this, we randomly selected 80% of the dataset as the training data and applied a five-fold cross validation. The threat to external validity is that it is plausible the classifiers produced on our training data would have low accuracy when categorizing comments from other projects. To alleviate this threat, we prepared labeled comments from four different software projects that cover different types of functionalities (e.g., Java collection framework and calendar system). While we believe that the comments from these software systems well represent comments in other software projects, we do not intend to draw any general conclusions. In the future, we will train the classifiers with more labeled comments of other kinds of systems to improve the generalizability. The user study was conducted with 14 users and 80 code entities. While we tasked each user with a lot of code and comment, we will extend the study to a larger user group.

9 RELATED WORK

Comment Classification. Researchers in [58] proposed a taxonomy based on meanings of comments and manually classified 1050 comments. They found 52.6% of these comments can be leveraged to improve software reliability and increase programmer productivity. In [52], researchers empirically studied API directives which are constraints about usages of APIs, and built a corresponding taxonomy. The authors of [46] leveraged grounded methods and analytical approaches to build a taxonomy of knowledge types in API reference documentation and manually classified 5574 randomly-sampled documentation units to assess the knowledge they contain. Based on this taxonomy, the researchers in [43] trained a classifier for each knowledge type and assigned only one label to each document unit based on nine features and their semantic and statistical combinations. In comparison, each classifier in our work classifies comments into different perspectives and code entities. The work [29] built a taxonomy of comments to investigate developers' commenting habits while the work [74] studied comment categorization to provide better quantitative insights about comment quality assessment. Researchers in [60, 61] produced a taxonomy of comments and investigated how often each category occurs by manually classifying more than 2,000 code comments. Unlike them, we develop the taxonomy to treat a comment as an attribute of a code entity and thus we can leverage program analysis techniques to infer, propagate, update and reason about comments. It is unclear how to propagate comments based on existing taxonomies.

Comment Generation. There are efforts of generating comments from source code/code changes, based on manually crafted templates [13, 19, 47, 48, 53, 54, 66, 71–73], information retrieval [27, 28, 32, 84, 85], and machine translation [14, 31, 33, 35, 62]. The techniques [84, 85] are most closely related and they generated comments for a code snippet by using comments of its code clone. However, they did not distinguish between comments of different perspectives and thus may generate many wrong comments. Also they did not utilize techniques like data flow analysis to propagate comments, which is our novelty. Our technique differs from comment generation in a few aspects. Comment generation produces comments from code. However, different projects have different coding and comment styles. A generation technique trained on a set of projects or based on rules may not generate good comments on other projects. Instead of generating comments from code, we propagate existing comments to code entities that are not commented by leveraging program analysis. Our technique is less sensitive to such styles as it only classifies comments instead of generating them. Comment propagation is deterministic and rigorous through program analysis. Secondly, generating comments for complex code that even humans can hardly understand is error-prone. For such cases, our technique can leverage existing comments (from other places). Thirdly, evaluating quality of generated comments, such as their naturalness, is a hard challenge. Our technique is largely immune to this. Finally, our technique can propagate comments to methods without code while existing work requires code as input. On the other hand, comment generation and comment propagation are complementary. Through propagation, we can produce a much larger training set for generation techniques. Generated comments can be propagated through our technique.

Comment-Code Inconsistency Detection. Research has been conducted on improving API documentation maintenance such as reporting potential code-comment inconsistencies as code evolves [21, 67], detecting existing code-comment inconsistencies [77–79, 89, 91], and enriching documentation (e.g., with code samples) [26, 34, 37, 75, 76, 82]. They do not aim to explicitly propagate comments as first-class objects and thus our efforts are complementary.

10 CONCLUSION

We build a comprehensive comment taxonomy from different perspectives with various levels of granularity and propose using program analysis to propagate comments. We develop a prototype CPC. Our experiments show that CPC can generate 41573 new comments with 88% accuracy. The derived comments are used to detect 37 new code bugs in 5 real-world projects with 30 confirmed and fixed by developers. We also identify 304 defects in existing comments, including 12 incomplete comments and 292 wrong comments. Our user study confirms propagated comments align well with existing comments regarding quality.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part by NSF-China 61802166, 61972193 and 61832009, DARPA FA8650-15-C-7562, NSF 1748764, 1901242 and 1910300, ONR N000141410468 and N000141712947, and Sandia National Lab under award 1701331. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] 2019. Apache Commons Collections. https://commons.apache.org/proper/commons-collections/.

[2] 2019. Case Marking Relation. http://universaldependencies.org/docsv1/u/dep/case.html.

[3] 2019. Comfirmed and Corrected Comments. https://issues.apache.org/jira/browse/COLLECTIONS-727.

[4] 2019. Comfirmed and Fixed Bugs. https://issues.apache.org/jira/browse/COLLECTIONS-710.

[5] 2019. Comfirmed and Fixed Bugs. https://issues.apache.org/jira/browse/JDO-780.

[6] 2019. Eclipse Java development tools (JDT). https://www.eclipse.org/jdt/.

[7] 2019. Guava. https://opensource.google.com/projects/guava/.

[8] 2019. JDK. https://www.oracle.com/technetwork/java/javase/downloads/index.html.

[9] 2019. Joda Time. https://www.joda.org/joda-time/.

[10] 2019. Preconjunct Relation. https://nlp.stanford.edu/software/dependencies_manual.pdf.

[11] 2019. sklearn metrics. https://scikit-learn.org/stable/modules/classes.html.

[12] 2019. word2vec tool. https://github.com/dav/word2vec.

[13] Nahla J Abid, Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2015. Using stereotypes in the automatic generation of natural language summaries for c++ methods. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on.* IEEE, 561–565.

[14] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning.* 2091–2100.

[15] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning.* 2123–2132.

[16] J Viera Anthony and M Garrett Joanne. 2005. Understanding Interobserver Agreement: The Kappa Statistic. *Family medicine* 37 (06 2005), 360–3.

[17] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, 242–253.

[18] L. Breiman. 2001. Random Forests. In *Machine Learning.* Vol. 45. 5–32.

[19] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering.* ACM, 33–42.

[20] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[21] Barthélémy Dagenais and Martin P Robillard. 2014. Using traceability links to recommend adaptive changes for documentation evolution. *IEEE Transactions on Software Engineering* 40, 11 (2014), 1126–1146.

[22] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06).* European Language Resources Association (ELRA).

[23] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Kathia M de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information.* ACM, 68–75.

[24] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* ACM, 213–224.

[25] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java expressions from free-form queries. In *Acm Sigplan Notices*, Vol. 50. ACM, 416–432.

[26] Andrew Habib and Michael Pradel. 2018. Is this class thread-safe? inferring documentation using graph-based learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* ACM, 41–52.

[27] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 2.* ACM, 223–226.

[28] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on.* IEEE, 35–44.

[29] Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. 2011. How good is your comment? a study of comments in java programs. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on.* IEEE, 137–146.

[30] Carl S Hartzman and Charles F Austin. 1993. Maintenance productivity: Observations based on an experience in a large system environment. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1.* IBM Press, 138–170.

[31] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension.* ACM, 200–210.

[32] Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan Luo. 2017. Mining version control system for automatically generating commit comment. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* IEEE Press, 414–423.

[33] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 2073–2083.

[34] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *Proceedings of the 39th International Conference on Software Engineering.* IEEE Press, 38–48.

[35] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering.* IEEE Press, 135–146.

[36] Zhen Ming Jiang and Ahmed E Hassan. 2006. Examining the evolution of code comments in PostgreSQL. In *Proceedings of the 2006 international workshop on Mining software repositories.* ACM, 179–180.

[37] Jinhan Kim, Sanghoon Lee, Seung-Won Hwang, and Sunghun Kim. 2013. Enriching documents with examples: A corpus mining approach. *ACM Transactions on Information Systems (TOIS)* 31, 1 (2013), 1.

[38] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[39] Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).*

[40] Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Meeting of the Association for Computational Linguistics.*

[41] Dan Klein and Christopher D Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1.* Association for Computational Linguistics, 423–430.

[42] R. Kohavi. 1995. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAIâĂŹ95.*

[43] Niraj Kumar and Premkumar Devanbu. 2016. OntoCat: Automatically categorizing knowledge in API Documentation. *arXiv preprint arXiv:1607.07602* (2016).

[44] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. 2015. From word embeddings to document distances. In *International Conference on Machine Learning.* 957–966.

[45] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[46] Walid Maalej and Martin P Robillard. 2013. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282.

[47] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension.* ACM, 279–290.

[48] Paul W McBurney and Collin McMillan. 2016. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119.

[49] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[50] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems.* 3111–3119.

[51] MB Miles, AM Huberman, and J Saldaña. [n.d.]. Qualitative data analysis: a methods sourcebook. 2013 Thousand Oaks.

[52] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17, 6 (2012), 703–737.

[53] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on.* IEEE, 23–32.

[54] Laura Moreno, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Jsummarizer: An automatic generator of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on.* IEEE, 230–232.

[55] Kimberly A Neuendorf. 2016. *The content analysis guidebook.* Sage.

[56] Anh Tuan Nguyen, Peter C Rigby, Thanh Van Nguyen, Mark Karanfil, and Tien N Nguyen. 2017. Statistical translation of English texts to API code templates. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on.* IEEE, 331–333.

[57] Chaparro Oscar, Lu Jing, Zampetti Fiorella, Moreno Laura, Di Penta Massimiliano, Marcus Andrian, Bavota Gabriele, and Ng Vincent. 2017. Detecting Missing Information in Bug Descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017).* ACM, New York, NY, USA, 396–407. https://doi.org/10.1145/3106237.3106285

[58] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. 2009. Listening to programmers Taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 331–341.

[59] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 815–825.

[60] Luca Pascarella. 2018. Classifying code comments in Java Mobile Applications. In *Conference on Mobile Software Engineering and Systems*.

[61] Luca Pascarella and Alberto Bacchelli. 2017. Classifying Code Comments in Java Open-source Software Systems. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. 227–237.

[62] Hung Phan, Hoan Anh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2017. Statistical learning for inference between implementations and documentation. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*. IEEE Press, 27–30.

[63] M.F. Porter. 1980. An algorithm for suffix stripping. *Program* 14, 3 (1980), 130–137.

[64] R. Quinlan and M. Kaufmann. 1993. *C4.5: Programs for Machine Learning*.

[65] Anna N Rafferty and Christopher D Manning. 2008. Parsing three German treebanks: Lexicalized and unlexicalized baselines. In *Proceedings of the Workshop on Parsing German*. Association for Computational Linguistics, 40–46.

[66] Sarah Rastkar, Gail C Murphy, and Alexander WJ Bradley. 2011. Generating natural language summaries for crosscutting source code concerns. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 103–112.

[67] Inderjot Kaur Ratol and Martin P Robillard. 2017. Detecting fragile comments. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 112–122.

[68] C. K. Roy and J. R. Cordy. 2008. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *2008 16th IEEE International Conference on Program Comprehension*. 172–181. https://doi.org/10.1109/ICPC.2008.41

[69] Cindy Rubio-González and Ben Liblit. 2010. Expect the unexpected: error code mismatches between documentation and the real world. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 73–80.

[70] C. Silva and B. Ribeiro. 2003. The importance of stop word removal on recall values in text categorization. In *Proceedings of the International Joint Conference on Neural Networks*. IEEE.

[71] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

[72] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 101–110.

[73] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Generating parameter comments and integrating with method summaries. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 71–80.

[74] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 83–92.

[75] Jeffrey Stylos, Brad A Myers, and Zizhuang Yang. 2009. Jadeite: improving API documentation using usage information. In *CHI'09 Extended Abstracts on Human Factors in Computing Systems*. ACM, 4429–4434.

[76] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 643–652.

[77] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 145–158.

[78] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 11–20.

[79] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 260–269.

[80] Ted Tenny. 1985. Procedures and comments vs. the banker's algorithm. *ACM SIGCSE Bulletin* 17, 3 (1985), 44–53.

[81] Ted Tenny. 1988. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering* 14, 9 (1988), 1271–1279.

[82] Christoph Treude and Martin P Robillard. 2016. Augmenting API documentation with insights from Stack Overflow. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 392–403.

[83] Grigorios Tsoumakas and Ioannis Katakis. 2007. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)* 3, 3 (2007), 1–13.

[84] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. 380–389.

[85] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. AutoComment: Mining Question and Answer Sites for Automatic Comment Generation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), New Idea*.

[86] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. DASE: Document-assisted Symbolic Execution for Improving Automated Software Testing. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE)*. IEEE, 620–631.

[87] Scott N Woodfield, Hubert E Dunsmore, and Vincent Yun Shen. 1981. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 215–223.

[88] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic model generation from documentation for Java API functions. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 380–391.

[89] Hao Zhong and Zhendong Su. 2013. Detecting API documentation errors. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 803–816.

[90] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 307–318.

[91] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 27–37.