@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies

Shin Hwei Tan
University of Illinois
Urbana, IL 61801, USA
stan6@illinois.edu

Darko Marinov
University of Illinois
Urbana, IL 61801, USA
marinov@illinois.edu

Lin Tan
University of Waterloo
Waterloo, ON N2L 3G1, Canada
lintan@uwaterloo.ca

Gary T. Leavens
University of Central Florida
Orlando, FL 32816, USA
leavens@eecs.ucf.edu

***Abstract*—Code comments are important artifacts in software. Java has standardized the writing of API specifications as Javadoc comments. API developers write Javadoc comments, and API users often read these comments to understand the API they use, e.g., reading a Javadoc comment for a method instead of reading the body of the method. An inconsistency between the Javadoc comment and body for a method indicates either a fault in the body or, effectively, a fault in the comment that can mislead the callers of the method to introduce faults in their code.**

We present a novel approach, called @tComment, for testing Javadoc comments, specifically for testing method properties about null values and related exceptions. Our approach consists of two components. Given source files for a Java project, the first component automatically analyzes the English text in each Javadoc comment to infer a set of likely properties for each method in the files. The second component generates random tests for these methods, checks the inferred properties, and reports inconsistencies. We evaluated @tComment on six open-source projects and found 24 inconsistencies between Javadoc comments and method bodies. We reported some of these inconsistencies, and 4 have already been confirmed and fixed by the developers.

I. Introduction

Source code comments are important artifacts in software and have been around for as long as code has been around. While comments are not compiled or executed, they aid in code comprehension, reuse, maintenance, etc. Comments can be broadly categorized into those that appear in the body of a method to describe its inner working and those that appear in the header of a method to describe its specification [23]. Java has standardized the writing of API specifications as Javadoc comments with tags such as `@param` to describe method parameters and `@throws` to describe what exceptions the method could throw. API developers write Javadoc comments to describe their classes and methods. API users often read these comments to understand the code; e.g., reading a Javadoc comment for a method instead of reading the body of the method.

A comment-code inconsistency between the Javadoc comment for a method and the code of that method's body is highly indicative of a fault. First, it can be the case that the comment is correct (in that it properly specifies what the code should do) but the method body has a fault (in that it improperly implements the specification). Second, it can be the case that the method body is correct (in that it properly implements the intended specification) but the comment is incorrect (in that it does not properly describe the intended specification). While the second case does not by itself have an executable fault, it can mislead the users of the method to introduce faults in their code [36].

Because comment-code inconsistencies are indicative of faults, it is important to check for such inconsistencies. However, automating such checking is challenging because it requires automated understanding of the natural-language text in the comments. While natural-language processing (NLP) techniques have made a lot of progress in the recent decades [24], they are still plagued by ambiguities that are inherent in understanding general text. For example, consider the Javadoc snippet "`@param chrono Chronology to use, null means default`", which describes a certain method parameter `chrono` that is an object of type `Chronology`. The part "`null means default`" is hard to understand, as it could specify that `null` is treated in some "default" manner (e.g., throwing a `NullPointerException`) or that `null` is used to represent some default type of `Chronology`.

The only currently viable solution for automated understanding of the natural-language text in the comments is to build *domain-specific analyses*. Tan et al. [36], [37] pioneered automated checking of comment-code inconsistencies based on NLP analysis. Their iComment [36] and aComment [37] projects focus on systems code written in C/C++, and analyze comments in the domains of locking protocols (e.g., "`the caller should grab the lock`"), function calls (e.g., "`function f must be called only from function g`"), and interrupts (e.g., "`this function must be called with interrupts disabled`"). Their tools extract rules from such comments and use *static analysis* to check source code against these rules to detect comment-code inconsistencies.

We present a novel approach, called *@tComment*, for *testing* comment-code inconsistencies in Javadoc comments and Java methods. We make the following contributions.

New Domain: We focus @tComment on a *new domain* in comments, specifically method properties for *null values and related exceptions* in Java libraries and frameworks. This domain was not studied in the previous work on detecting comment-code inconsistencies, but our inspection of several Java projects showed that this domain is important and widely represented in well-documented Java code. Detecting comment-code inconsistencies in this domain has unique challenges that require new solutions, as discussed below.

Dynamic Analysis: @tComment uses a *dynamic analysis* to check comment-code inconsistencies, unlike previous work that used static analysis. Specifically, our @tComment implementation builds on the Randoop tool [28] for random test generation of Java code. Randoop randomly generates method sequences of the code under test, checks if execution of these sequences violate a set of default *contracts* such as throwing an uncaught exception [28], and generates as tests those sequences that violate some constraint. We modify Randoop to check @tComment-inferred properties during random test generation and to report violations which correspond to comment-code inconsistencies. We refer to our modified Randoop as *@Randoop*.

We chose dynamic analysis to address the following challenges imposed by the new domain. First, even widely used tools for static checking of Java code, such as FindBugs [17], can have a large number of false alarms when checking properties related to `null` values and exceptions if these properties are available for only some parts of the code, which is the case when inferring properties from Javadoc comments that are not available for all methods. Second, we focus on Java libraries and frameworks, which have few calls to their own methods from their own code bases. Therefore, an analysis cannot focus on callers to these methods to see what parameters they pass in. Instead, a dynamic approach such as Randoop, which generates call sequences to test library methods, is particularly beneficial.

Improved Testing: @Randoop allows us not only to *detect comment-code inconsistencies* but also to *improve test generation* in Randoop. For detecting inconsistencies, @Randoop generates tests that Randoop would not necessarily generate otherwise because these tests need not violate the default contracts that Randoop checks. For improving test generation, @Randoop identifies Randoop false alarms, and ranks them lower so that developers can focus on the true faults. False alarms refer to tests that Randoop generates but that do *not* find fault in the code under test; these tests cause some methods to throw exceptions, but these exceptions are expected according to the Javadoc comments for those methods.

Evaluation: We applied @tComment on six open-source Java projects (`Apache Commons Collections`, `Apache Log4j`, `Apache Lucene`, `Glazed Lists`, `JFreeChart`, and `Joda-Time`), which have well-developed, well-tested, and well-documented code. We found 24 methods with inconsistencies between Javadoc comments and method bodies in these projects. We reported some of these inconsistencies, and 4 of them have already been confirmed and fixed by the developers (one by fixing the code and three by fixing the comment), while the rest await developer verification. We automatically inferred 4,672 properties regarding null values and their related exceptions from Javadoc comments with high accuracies of 94–100%. The high accuracies were achieved without using NLP techniques, largely due to the more structured Javadoc null-related comments with less paraphrases and variants.

II. Examples

We illustrate how @tComment can be used by showing three examples of comment-code inconsistencies that we found with @tComment in two projects, and one example of a Randoop false alarm identified by @tComment. The first three examples show progressively more complex cases: (1) inferring that *some* exception should be thrown when a method parameter is `null`; (2) inferring what *type* of exception should be thrown when *one* method parameter is `null`; and (3) inferring what *type* of exception should be thrown when *two* method parameters are `null`.

Consider first the `Joda-Time` project [8], a widely used Java library for representing dates and times. `Joda-Time` provides several classes that support multiple calendar systems. `Joda-Time` code is fairly well commented with Javadoc. We ran @tComment to infer properties for the methods in `Joda-Time` and to check these properties with our @Randoop. For each test that @Randoop generates, it marks whether the test, when executed, violates some @tComment-inferred property or a default Randoop contract.

Figure 1 shows an example test that violates a @tComment-inferred property. This test creates a `MutablePeriod` object `var2` and invokes several methods on it. The executions of `setSeconds` and `setValue` methods finish normally, but for `setPeriod`, @tComment reports that there is a likely comment-code inconsistency: the parameter `var9` is `null`, but the method execution throws no exception, which disagrees with the corresponding comment indicating that some exception should be thrown. Note that this test *passes* if some exception is thrown and fails otherwise.

Figure 1 also shows the relevant parts of the `setPeriod` method. It has two parameters, and the Javadoc comment provides a description for each of them. A typical Javadoc comment has the main, free-flow text (for brevity omitted in our examples) and specific *tags/clauses* such as `@param`, `@throws`, `@return`, etc. We call the entire block of comment before a method as *one comment* with several *comment tags*. Figure 1 shows one Javadoc comment with two `@param` tags and one `@throws` tag.

The key part here is "`not null`" for the `chrono` parameter. @tComment infers the property that whenever `chrono` is `null`, the method should throw *some* exception (although it does not know which exception because the tag for `ArithmeticException` is not related to `null`).

```
// In automatically generated test class:
void test1() throws Throwable {
  org.joda.time.MutablePeriod var2 =
    new org.joda.time.MutablePeriod(1L, 0L);
  var2.setSeconds(0);
  var2.setValue(0, 0);
  org.joda.time.Chronology var9 = null;
  try {
    var2.setPeriod(0L, var9);
    fail("Expected␣some␣exception␣for␣chrono==null");
  } catch (Exception expected) {}
}

// In a class under test:
/** ...
 * @param duration  the duration, in milliseconds
 * @param chrono  the chronology to use, not null
 * @throws ArithmeticException if the set exceeds
 *          the capacity of the period
 */
void setPeriod(long duration, Chronology chrono)
```

Figure 1. Example test generated by @Randoop. Method under test and its comment.

@Randoop finds that the shown test violates this property. Note that it may not be a comment-code inconsistency; the inference could have been wrong, e.g., "`not null`" could represent the method precondition—such that if `chrono` is `null`, the method could do anything and is not required to throw an exception—or "`not null`" could be a part of a larger phrase, say, "`not a problem to be null`"—such that the method must not throw an exception.

In this case, our inspection showed that @tComment performed a correct inference and detected a real comment-code inconsistency. In fact, @tComment also found a similar inconsistency for another overloaded `setPeriod` method. We reported both inconsistencies in the `Joda-Time` bug database [3], and `Joda-Time` developers fixed them by changing comments. It is important to note that Randoop would have *not* generated this example test because it does not throw an exception. More precisely, Randoop internally produces the method sequence but would not output it to the user as a possibly fault-revealing test. Indeed, @Randoop generates the test precisely because it does not throw any exception when some exception is expected.

Consider next the `Apache Commons Collections` project (called just `Collections` for short) [5], a popular library for representing collections of objects. Figure 2 shows two example tests, each of which violates a @tComment-inferred property, and the corresponding method declarations and their comments.

For the `synchronizedMap` method, @tComment *correctly* infers that the method should throw `IllegalArgumentException` when the parameter `map` is `null`; while this is explicit in the `@throws` tag, note that the `@param map` tag could be contradicting by allowing any behavior when `map` is `null`. Inferring a specific type of expected exception is unlike in the previous example when @tComment could only infer

```
// In automatically generated test class:
void test2() throws Throwable {
  java.util.Map var0 = null;
  try {
    java.util.Map var1 = org.apache.commons.collections.
        MapUtils.synchronizedMap(var0);
    fail("Expected␣IllegalArgumentException,␣" +
        "got␣NullPointerException");
  } catch (IllegalArgumentException expected) {}
}

void test3() throws Throwable {
  java.util.Collection var0 = null;
  java.util.Iterator[] var1 = new java.util.Iterator[]{};
  try {
    org.apache.commons.collections.CollectionUtils.
        addAll(var0, (java.lang.Object[])var1);
    fail("Expected␣NullPointerException␣" +
        "for␣collection==null");
  } catch (NullPointerException expected) {}
}

// In classes under test:
/** ...
 * @param map  the map to synchronize, must not be null
 * @return a synchronized map backed by the given map
 * @throws IllegalArgumentException  if the map is null
 */
static Map synchronizedMap(Map map)

/** ...
 * @param collection  the collection to add to, must not be null
 * @param elements  the array of elements to add, must not be null
 * @throws NullPointerException if the collection or array is null
 */
static void addAll(Collection collection, Object[] elements)
```

Figure 2. Two more example tests generated by @Randoop. Methods under test and their comments.

that *some* exception should be thrown. Indeed, inferring the type in this case is important because, when `map` is `null`, the method does throw an exception but of a different type. In this case, even the original Randoop can generate `test2`, because it throws an exception. However, Randoop also generates dozens of others tests that are not fault-revealing, so this comment-code inconsistency would be "the needle in a haystack" among the tests generated by Randoop. In contrast, @Randoop prominently highlights the inconsistency. We reported this inconsistency [1], and the `Apache Commons Collections` developers fixed it by removing the comment.

For the `addAll` method, @tComment *correctly* infers that the method should throw `NullPointerException` when either the parameter `collection` is `null` *or* the parameter `elements` is `null`. This is similar to the previous case where the specific exception type is inferred, but in this case two method parameters are involved. The inconsistency that @tComment finds is, in fact, related to the situation where only one parameter—`collection`—is `null` (while the array is empty), but the exception is not thrown as expected. We also reported this comment-code inconsistency [2], and it is under consideration.

For all examples presented so far, an exception is expected (according to the Javadoc comments), but the method

```
/** ...
 * @param id   the id to use
 * @throws IllegalArgumentException if the id is null
 */
protected DateTimeZone(String id)
```

Figure 3. Null-related Javadoc comment helps identify false alarms that Randoop would generate.

under test either does not throw an exception or throws an exception of a different type. We next discuss an example where an exception is thrown by the method under test, but it is expected as indicated by the relevant comment (Figure 3). This example illustrates a case where Randoop would generate a false alarm, but @tComment helps remove this false alarm automatically by lowering its ranking through analyzing the comment. @tComment infers that the constructor `DateTimeZone` should throw an `IllegalArgumentException` if `null` is passed to the `id` parameter. Randoop generates a test for `DateTimeZone` when `id` is `null`, because the execution of this test indeed throws an `IllegalArgumentException`. Randoop reports as potentially fault-revealing all tests that throw uncaught exceptions during execution. Since the exception is expected according to the comment, @tComment lowers the ranking of this false alarm to improve the testing accuracy.

III. @tComment Design

Our @tComment approach consists of two components. The first component takes as input the source code for a Java project, automatically analyzes the English text in the Javadoc comments in the project, and outputs a set of inferred likely properties for each method. The second component takes as input the same code and inferred properties, generates random tests for the methods in the code, checks the inferred properties, and reports inconsistencies.

Similar to prior work [36], [37], we build a domain-specific analysis, due to the difficulty of inferring arbitrary properties from general comments. In particular, we focus on null-pointer related comments because null-pointer deferences are common memory bugs [11], and a large percentage of Javadoc comments (25.4% in the six projects we evaluated) contain the keyword `null`.

A. Inferring Properties from Comments

Our goal is to infer from Javadoc comments null-related properties about method parameters. Table I shows the four kinds of properties that @tComment infers. For *each* method parameter of non-primitive type, @tComment infers one of the four kinds of properties: (1) *Null Normal*; (2) *Null Any Exception*; (3) *Null Specific Exception*; or (4) *Null Unknown*.

Intuitively, @tComment first searches for the keyword `null` in Javadoc comments with `@param` and `@throws` tags to identify potentially relevant comments. It infers *Null Normal* and *Null Any Exception* properties from the `@param` tags and *Null Specific Exception* properties from the `@throws` tags. It assigns *Null Unknown* to a method parameter if it cannot infer any of the other three properties about the parameter, i.e., there are neither `@param` nor `@throws` tags regarding that parameter, or there are such tags but none describes a null-related property about that parameter.

Table I shows four examples of comment tags and their corresponding properties inferred. For example, @tComment infers from the second tag "`@param collection the collection to add to, must not be null`", that if the method parameter `collection` is null, the method is expected to throw some exception, represented as $collection == null => exception$. Based on our experience with the null-related Javadoc comments, we found that this interpretation is generally what developers mean, and thus we adopted it for @tComment. The comment-code inconsistencies that we reported and developers confirmed, as well as the low false-positive rate of our detection for comment-code inconsistencies, confirm our decision. However, note that we focus on *library projects*, where the methods need not trust their callers. The interpretation may differ for applications with more of a design-by-contract mentality where callers were trusted more. As discussed earlier, this example tag could have another interpretation, describing a precondition such that passing null for `collection` allows the method to do anything, not necessarily throw an exception.

Our @tComment implementation leverages the Javadoc doclet [33] to parse Javadoc comments. For example, consider the above tag for `collection`; Figure 2 shows the comment for this tag and the corresponding method declaration. The Javadoc doclet parses this tag and outputs the name of the method parameter (`collection`), its type (`java.util.Collection`), the method's full signature (`...CollectionUtils#addAll(Collection collection, Object[] elements)`), and the free-form text ("`the collection to add to, must not be null`"). The method parameter, its type, and the full signature are used later by the test-generation component of @tComment to check tests generated by @Randoop.

@tComment uses relatively simple heuristics to analyze the free-form text. While the heuristics are not perfect, our empirical evaluation shows that they are highly accurate in practice. If negation words, such as "`not`" or "`never`", are found *up to three words* before or after the word `null`—e.g., "`the collection to add to, must not be null`" has "`not`" two words from `null`—@tComment infers the *Null Any Exception* property. If no negation words are found up to three words around `null`—e.g., the first tag in Table I—@tComment infers the *Null Normal* property.

For `@throws` tags, consider for example "`@throws IllegalArgumentException if the id is null`", the Javadoc doclet parses the tag and outputs the specific

Properties	Meaning	Comment Example	Notation
Null Normal	If the parameter is `null`, the method should execute normally (no exception).	`@param predicate the predicate to use, may be null`	predicate==null => normal
Null Any Exception	If the parameter is `null`, the method should throw *some* exception.	`@param collection the collection to add to, must not be null`	collection==null => exception
Null Specific Exception	If the parameter is `null`, the method should throw a *specific* type of exception.	`@throws IllegalArgumentException if the id is null`	id==null => IllegalArgumentException
Null Unknown	We do not know the expected behavior of the method when the parameter is `null`.	`@param array the array over which to iterate`	array==null => unknown

Table I
PROPERTIES TO BE EXTRACTED. THE COMMENT EXAMPLES ARE REAL COMMENTS FROM THE SIX EVALUATED PROJECTS.

exception (`IllegalArgumentException`) and the free-form text ("`if the id is null`"). @tComment simply splits the text into words and searches each word in the list of all valid method parameters generated by the Javadoc doclet. If a valid parameter name is found—e.g., `id`—@tComment infers the property $id == null => IllegalArgumentException$.

If the keyword "`or`" is in the `@throws` comment text, e.g., "`@throws NullPointerException if the collection or array is null`" in Figure 2, @tComment generates multiple properties, e.g., $collection == null => NullPointerException$ and $array == null => NullPointerException$. If both *Null Any Exception* and *Null Specific Exception* properties are inferred for the same method parameter, e.g., `collection`, @tComment keeps only the *Null Specific Exception* property.

B. Checking Properties in Test Generation

After @tComment infers likely method properties, it uses our modified Randoop, called *@Randoop*, to check these properties using random test generation. Figure 4 shows the simplified pseudo-code of the Randoop test-generation algorithm [28], together with our extension for checking @tComment-inferred properties.

We briefly summarize how Randoop works. It produces random *sequences* of method calls (including constructors) of the code under test. It maintains a set of error sequences (to be output as generated unit tests that are likely fault revealing) and a set of non-error sequences (to be used for creating longer sequences). In a loop, it first randomly selects a method m whose k parameters (including the receiver for non-static methods) have types $T_1 \ldots T_k$. It then selects sequences (previously generated) and values (e.g., "`0`", "`1L`", or "`null`") of appropriate type to use for the method parameters. It concatenates these sequences and adds a new call to m. It then executes the new sequence to check *contracts* (e.g., no uncaught exception during execution). If there is a violation, it adds the new sequence to the error sequences; otherwise, it adds the new sequence to the non-error sequences. More details of the original algorithm, including discarding duplicates and filtering extensible sequences, are available elsewhere [28].

```
// inferredNull is specific to @Randoop
GenerateSequences(classes, contracts, inferredNull, timeLimit)
  errorSeqs ← {} // These will be generated as unit tests
    // we add a comment-code inconsistency field to sequences
  nonErrorSeqs ← {} // These are used to build longer sequences
  while timeLimit not reached do
    // Create new sequence
    $m(T_1 \ldots T_k)$ ← randomPublicMethod(classes)
    ⟨seqs, val⟩ ← randomSeqsAndVals(nonErrorSeqs, $T_1 \ldots T_k$)
    newSeq ← extend($m$, seqs, vals)
    // Execute new sequence and check contracts.
    violated ← execute(newSeq, contracts)
    // Classify new sequence and outputs.
    if violated then
      errorSeqs ← errorSeqs ⋃ {newSeq}
    else
      nonErrorSeqs ← nonErrorSeqs ⋃ {newSeq}
    end if
    // Execute and check @tComment-inferred properties.
    match ← execute(newSeq, inferredNull)
    if match = 'Missing Exception' then
      // Add the new sequence, marked as inconsistency
      errorSeqs ← errorSeqs ⋃ { newSeq }
      newSeq.isCommentCodeInconsistency ← highlyLikely
    else if match = 'Different Exception' or
          match = 'Unexpected Exception' then
      // Mark an already added sequence as inconsistency
      newSeq.isCommentCodeInconsistency ← likely
    else if match = 'Unknown Status' then
      // Unknown inconsistency status
      newSeq.isCommentCodeInconsistency ← unknown
    else // match = 'Expected Exception'
      // Remove the new sequence if it matches the properties
      errorSeqs ← errorSeqs \ { newSeq }
    end if
  end while
  return ⟨ nonErrorSeqs, errorSeqs ⟩
```

Figure 4. Integration of @Randoop checking of @tComment-inferred properties into test generation.

Our @Randoop modification follows the similar approach that Randoop performs for checking contracts: @Randoop executes the sequence and checks the @tComment-inferred properties for method calls where one or more parameter have `null` values. We distinguish five kinds of matches between method execution (does it throw an exception and of what type) and @tComment-inferred properties (is an

	Comment-Code Inconsistent			Unknown	Comment-Code Consistent
	Missing Exception	*Different Exception*	*Unexpected Exception*	*Unknown Status*	*Expected Exception*
Exception Thrown	No	Yes	Yes	Yes	Yes
Properties that @tComment inferred about expected exceptions for method parameters with `null` values	*Null Any Exception* or *Null Specific Exception*	(1) at least one is *Null Specific Exception* & (2) thrown exception is not in the set of all specific exceptions	(1) at least one is *Null Normal* & (2) there is no *Null Specific Exception* or *Null Any Exception*	*Null Unknown* for all method parameters	(1) at least one is *Null Specific Exception* or *Null Any Exception* & (2) thrown exception is in the set of expected exceptions

Table II
CATEGORIES OF SEQUENCES THAT @RANDOOP CLASSIFIES BASED ON MATCHES FOR PARAMETERS WITH NULL VALUES.

exception expected and of what type). Based on the match, @Randoop can (1) generate a sequence that Randoop would not generate otherwise, (2) generate the same sequence as Randoop but potentially mark the sequence as a comment-code inconsistency, or (3) generate the same sequence as Randoop but mark the sequence as a false alarm.

Before we describe the five possible kinds of matches, we describe how @Randoop handles multiple `null` values. Note that these can naturally arise for methods with several non-primitive parameters, e.g., `addAll` from Figure 2 has "`Collection collection`" and "`Object[] elements`". If only one of these parameters is `null`, @Randoop uses only the property inferred for that parameter. If two or more parameters are `null`, @Randoop computes the *set* of all expected exceptions for these parameters. For example, if we had $collection == null => NullPointerException$ and $elements == null => IllegalArgumentException$, then @Randoop would assume that either of the two exceptions is expected, but some third exception (say, `ConcurrentModificationException`) would not be expected. If some parameter with `null` value has the *Null Any Exception* property, then all types of exceptions are expected.

Table II lists the five kinds of matches:

Missing Exception sequences throw no exception during execution, but the corresponding inferred properties specify that some exception is expected. These sequences should be generated as tests that are likely comment-code inconsistencies (although they could be false alarms if the inference obtained incorrect properties from the corresponding comments). Note that these tests would *not* be generated by the original Randoop because they throw no exception.

Different Exception sequences throw an exception that is different from the exception expected according to the inferred properties. These are also likely comment-code inconsistencies. These sequences would be generated by Randoop as potentially fault-revealing tests, and by inspecting them, the developer could find the inconsistency. However, these tests would be hard to identify among a large number of tests that Randoop generates (in our evaluation, only 8tests were *Different Exception* among 1,285tests that Randoop would generate). In contrast, @Randoop highlights these cases.

Unexpected Exception sequences throw an exception whereas @tComment explicitly expects normal execution with no exception. As for *Different Exception*, Randoop would also generate these sequences as tests, but @Randoop highlights them due to the inconsistency.

Unknown Status sequences throw an exception but @tComment inferred no property to tell if the exception is expected or not. Both Randoop and @Randoop generate these as error sequencess. While they may indicate a fault in the code, they do not show an inconsistency between code and comment (unless the inference incorrectly missed inferring some property about exceptions).

Expected Exception sequences throw an exception, but this exception is expected according to the properties inferred by @tComment from the relevant comments. Hence, @Randoop does not generate these sequences as tests. Effectively, they show the number of Randoop false alarms that @Randoop identifies, which improves testing. If the inference is wrong, @tComment may increase the time for developers to find the true fault-revealing tests.

Currently, @Randoop *modifies only the checking and not the random selection* performed by Randoop. Randoop randomly selects methods to test and parameter values for the methods, and @Randoop does not perform any additional selection that the original Randoop does not perform. It could be beneficial to bias the selection based on the properties inferred from comments. For example, if it is inferred that an exception should be thrown when a method parameter `p` is `null` (i.e., *Null Specific Exception* or *Null Any Exception*), but Randoop does not select any sequence where `p` is `null`, @Randoop could generate such additional sequences to check if the inferred property holds. This extension remains as future work.

IV. EVALUATION

We evaluate @tComment on six open-source Java projects. Table III lists information about these projects. We modified Randoop revision 652 to build @Randoop. Randoop provides several options that control random generation, and we consider two options that are the most important for @Randoop: (1) *`nullRatio`* specifies the frequency that the `null` value is randomly chosen as an input

Project	Source	Description	Version	LOC	# Classes	# Methods
`Collections`	[5]	Collection library and utilities	3.2.1	26323	274	2943
`Glazed Lists`	[38]	List transformations in Java	1.8	5134	35	456
`JFreeChart`	[16]	Chart creator	1.0.13	72490	396	5688
`Joda-Time`	[8]	Date and time library	1.6.2	25916	154	3094
`Log4j`	[6]	Logging service	1.2	20987	221	20987
`Lucene`	[7]	Text search engine	2.9.3	48201	422	4017

Table III
Subject projects and their basic statistics

Project	*Missing Exception*	*Different Exception*	*Unexpected Exception*	*Unknown Status*	*Expected Exception*	*Tested Properties*
`Collections`	4 = 4 + 0 + 0	4 = 2 + 0 + 2	23 = 2 + 16 + 5	105	60	109
`Glazed Lists`	0 = 0 + 0 + 0	4 = 0 + 0 + 4	9 = 2 + 5 + 2	149	10	26
`JFreeChart`	3 = 1 + 2 + 0	0 = 0 + 0 + 0	2 = 0 + 0 + 2	119	7	15
`Joda-Time`	2 = 2 + 0 + 0	0 = 0 + 0 + 0	21 = 0 + 0 + 21	36	8	33
`Log4j`	2 = 2 + 0 + 0	0 = 0 + 0 + 0	2 = 0 + 0 + 2	186	152	156
`Lucene`	12 = 6 + 6 + 0	0 = 0 + 0 + 0	12 = 6 + 0 + 6	364	5	43
Total	23 =15 + 8 + 0	8 = 2 + 0 + 6	69 =10 + 21 + 38	959	242	382

				Total
True Inconsistencies	15	2	10	27
False Alarms	8	0	21	29
Order of Exceptions	0	6	38	44

Table IV
Overall results for the default configuration of our @Randoop (nullRatio=0.6, timeLimit=3600s)

for a method parameter of some non-primitive type (e.g., a `nullRatio` of 0.6 instructs Randoop to use `null` 60% of the time); and (2) *`timeLimit`* specifies the number of seconds that Randoop should generate tests for one project. All experiments were performed on a machine with a 4-core Intel Xeon 2.67GHz processor and 4GB of main memory.

A. Comment-Code Inconsistency Detection Results

Table IV shows the results of @tComment with the default values for @Randoop options. (Section IV-C discusses the sensitivity of the results to the value of these options.) For each project, we tabulate the number of tests that @Randoop generated (or did not generate in comparison with Randoop) based on the five kinds of matches between inferred properties and method executions (Section III-B). For three kinds of matches that could have comment-code inconsistencies, the cells show the detailed breakdown of True Inconsistencies, False Alarms, and Order of Exceptions (as described later). The last column also shows the number of @tComment properties that @Randoop checked during test generation.

In total, @Randoop generated 23+8+69 tests with potential comment-code inconsistencies. Note that Randoop would not generate 23 of those (column '*Missing Exception*') where methods execute normally while exceptions are expected by the corresponding comments. @Randoop also generates 8 tests where an exception is thrown but normal execution was observed (Column '*Unexpected Exception*') and 69 tests where an exception is thrown but different than specified by comments (Column '*Different Exception*'). @Randoop generates 959 tests that throw an exception for cases where no null-related properties were inferred (Column '*Unknown Status*'). Last but not least, @Randoop identifies 242 tests as throwing exceptions expected by the comments (Column '*Expected Exception*').

The cells with sums show the detailed results of comment-code inconsistencies reported by @tComment. We inspected all the reports by carefully reading the comments and the code to determine if they are indeed inconsistent (Row 'True Inconsistencies') or not (Row 'False Alarms'). @tComment detected 24 previously unknown comment-code inconsistencies. The primary source of false alarms is inaccurate inference (Section IV-B). For example, @tComment inferred the property $filter == null => exception$ from "`@param filter if non-null, used to permit documents to be collected`", because the negation word `non` is next to `null`. However, this comment tag does not imply the parameter `filter` cannot be null. Advanced NLP analysis may be leveraged to analyze this tag correctly.

We also separately show (Row 'Order of Exceptions') the cases where the result depends on the order in which `@throws` tags are written in a Javadoc comment. For example, the method `arrayIterator(Object[] array, int start, int end)` has two `@throws` listed ("`@throws IndexOutOfBoundsException if array bounds are invalid`" and "`@throws`

Project	`@param`	`@throws`	`@param` with null	`@throws` with null	Properties	*Null Specific Exception*	*Null Normal*	*Null Any Exception*	*Null Unknown*	Accuracy %
`Collections`	2996	862	1104	414	702	190	83	54	375	99
`Glazed Lists`	596	120	60	38	124	7	14	0	103	100
`JFreeChart`	20240	150	7200	6	1815	0	387	515	913	100
`Joda-Time`	8690	1544	1912	192	829	89	445	21	274	94
`Log4j`	6562	0	4056	0	685	0	243	217	225	100
`Lucene`	4178	746	200	30	517	12	16	22	467	99
Total	43262	3422	14532	680	4672	298	1188	829	2357	99.1 average

Table V
COMMENT ANALYSIS RESULTS

`NullPointerException if array is null`". The execution throws `IndexOutOfBoundsException`, while @tComment expects `NullPointerException`. Semantically, if one were to consider `@throws` tags as a *set*, they would either be infeasible (a method should throw two or more different exceptions for some input) or non-deterministic (a method can throw either of the two or more exceptions). The Javadoc specification [30] states that "The exceptions thrown from a method need not be mutually exclusive." It appears that developers mostly treat these tags as a *list* and require that the exception be thrown for the *first* `@throws` tag that satisfies the condition. @tComment does not currently encode this ordering of exceptions, but we plan to add it in the future.

B. Comment Analysis Results

@tComment analyzes all `@param` and `@throws` tags that contain the keyword `null`. Table V shows the detailed comment analysis results. Columns 'param' and 'param with null' show the total number of `@param` tags analyzed and the number of `@param` tags that contain `null`, respectively. Similarly, columns 'throws' and 'throws with null' show the total number of `@throws` tags analyzed and the number of `@throws` tags that contain `null`, respectively. In total, 15,212 `@param` and `@throws` tags contain `null` in the six evaluated projects. @tComment inferred 4,672 properties from these tags, of which 2,315 are *Null Normal*, *Null Any Exception*, and *Null Specific Exception* properties. Of these 2,315 properties, 382 are tested by @tComment to automatically detect comment-code inconsistencies and improve testing. As discussed in Section III-B, it would be beneficial to modify Randoop's random selection to actively test more of the inferred properties in the future.

To evaluate the accuracy of our automatic comment-analysis technique, we randomly sample 100 of the inferred properties from each project, and manually read them and the corresponding `@param` and `@throws` tags to check if the inferred properties are correct. (We sampled only 50 properties from `Glazed Lists` because the number of `@param` and `@throws` comments that contain the keyword `null` is small compared to other projects.) Note that the manual inspection is purely for evaluating the accuracy of our comment analysis; @Randoop *directly* uses the automatically inferred properties to detect comment-code inconsistencies and improve testing, and no manual inspection of the inferred properties is required.

Our analysis of the free-form comment text achieves high accuracy of 94–100% (Column 'Accuracy %') without using NLP techniques as iComment [36] did. The high accuracy is partially due to the Javadoc API comments being much more structured than the comments in systems code written in C. There is also less variance in paraphrases and sentence structures in the Javadoc comments than in the C/C++ comments in systems code. Our result suggests that automatic analysis of other Javadoc comments is likely to be highly accurate, which can help us find more bugs and further improve testing in the future.

In addition, it is unlikely that we missed any null-related properties described in `@param` and `@throws` Javadoc comments, because developers are unlikely to describe a null-related property without using or misspelling the keyword `null` (confirmed by our cursory examination), and we analyzed all `@param` and `@throws` tags that contain `null`. If any null-related properties are described in non-Javadoc style comments, e.g., without using the `@param` tag, @tComment would not analyze it. As we do not anticipate many such comments, we focused on such tagged Javadoc comments.

C. Sensitivity of @Randoop Options

We want to understand how different values for @Randoop options `nullRatio` and `timeLimit` affect our results of comment-code inconsistency detection and false alarm pruning. When time allows, users can run @tComment with many `nullRatios` and `timeLimits` to try to detect more inconsistencies.

We run @Randoop with 5 `timeLimits`—50sec, 100sec, 200sec, 400sec, and 800sec—and 11 `nullRatios`—from 0.0 to 1.0 in increments of 0.1—on all six projects and measured the most important metric, the number of *Missing Exception* tests. These are 5*11*6, a total of 330, sets of experiments. Despite the randomness in @Randoop, it identifies more *Missing Exception* tests (thus potentially detects more comment-code inconsistencies) as the `timeLimit` increases for all cases but one combination of the value and the project. We found that when running @Randoop for 800sec, the value of `nullRatio` 0.6 helps @Randoop identify the largest number of *Missing Exception* tests for each and every of the six projects. Therefore, it was chosen as the default `nullRatio`. We found that 0.3, 0.5, 0.7, 0.8, and 0.9 are the next best `nullRatios` for these six projects considering the number of *Missing Exception* tests identified.

To further understand the effect of `timeLimits`, we experimented with varying `timeLimits` with `nullRatio` 0.6 going up to one hour and two hours. We found that the number of the *Missing Exception* tests reaches a plateau at about one-hour mark, which is similar to the fact that the original Randoop reaches a plateau around one-hour mark [27].

To further understand the effect of `nullRatios`, we performed additional experiments with `timeLimit` one hour and `nullRatios` from 0.3 to 0.9. The results show that different `nullRatios` between 0.3 and 0.9 inclusive produce almost the same numbers for these five kinds of matches, suggesting that if one plans to run @Randoop for an hour, he or she can pick any `nullRatio` from 0.3 to 0.9 to obtain similar results.

V. Related Work

Automated Software Testing. Many automated software testing techniques are designed to detect software bugs [12], [13], [18], [26], [28], [43], e.g., based on random generation or using specifications. @tComment leverages an additional source—code comments—and modifies Randoop [28] to detect more bugs (in both code and comments), and to identify the false alarms generated by Randoop. It is quite conceivable to extend @tComment to improve other automated testing techniques.

Detecting Comment-Code Inconsistencies. iComment [36] and aComment [37] extract rules from comments and check source code against these rules *statically*. The differences between iComment/aComment and @tComment were already discussed in Introduction, and we only summarize them here: (1) @tComment leverages a new type of comments, related to null values; (2) @tComment employs a dynamic approach to check comments during testing; and (3) in addition to finding comment-code inconsistencies, @tComment detects false alarms generated by tools such as Randoop.

A recent empirical study [20] examines the correlation between code quality and Javadoc comment-code inconsistencies. It checks only simple issues, e.g., whether the parameter names, return types, and exceptions in the `@param`, `@return`, and `@throws` tags are consistent with the actual parameter names, return types, and exceptions in the method. Sun's Doc Check [34] detects Javadoc errors such as missing and incomplete Javadoc tags. Different from checking for these *style* inconsistencies, @tComment detects *semantic* comment-code inconsistencies related to null values.

Empirical Studies of Comments. Several empirical studies aim to understand the conventional usage of comments, the evolution of comments, and the challenges of automatically understanding comments [19], [23], [41], [42]. None of them automatically analyze comments to detect comment-code inconsistencies or improve automated testing.

Comment Inference from Source Code. Some recent work infers comments for failed test cases [44], exceptions [9], API function cross-references [22], software changes [10], and semantically related code segments [31], [32]. Comments automatically generated by these techniques are more structured than developer-written comments, therefore we expect it would be easier to leverage such automatically-generated comments for bug detection. However, it is still beneficial to improve analysis of developer-written comments because (1) millions lines of developer-written comments are available in modern software, and (2) these developer-written comments bring in information that is not available in source code [36] to help us detect more bugs and false alarms generated in automated testing. FailureDoc [44] augments a failed test with debugging clues, which could be extended to help explain why the tests generated by @tComment fail.

Analysis of Software Natural Language Text. Various research projects analyze software natural-language text such as bugs reports [4], [15], [21], [25], [29], [35], [39], [40], API documentation [45], and method names [14] for various purposes including detecting duplicate bug reports, identifying the appropriate developers to fix bugs, etc. @tComment analyzes comments written in a natural language to detect comment-code inconsistencies and to improve automated testing. Different from some of these studies [14], [45] that use NLP techniques such as part-of-speech tagging and chunking, @tComment does not use NLP techniques because our simple comment analysis can already achieve high accuracy of 94–100% partially due to the more structured Javadoc comments with less paraphrases and variants (Section IV-B).

VI. Conclusions

Developers read, write, and modify comments together with code. An inconsistency between comment and code is highly indicative of program faults. We have presented a novel approach, called @tComment, for testing consistency of Java method bodies and Javadoc comments properties related to null values and exceptions. Our application of @tComment on six open-source projects discovered 24 methods with inconsistencies between Javadoc comments and bodies. We reported these inconsistencies, and 4 were already confirmed and fixed by the developers. Additionally, @tComment improves Randoop by lowering the ranking of false alarms. These promising results motivate us to extend @tComment for other types of properties and to integrate it with other testing tools.

References

[1] Apache Commons Collections Bug Report. https://issues.apache.org/jira/browse/COLLECTIONS-384.

[2] Apache Commons Collections Bug Report. https://issues.apache.org/jira/browse/COLLECTIONS-385.

[3] Joda Time Bug Report. http://sourceforge.net/tracker/?func=detail&atid=617889&aid=3413869&group_id=97367.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE'06*.

[5] Apache Software Foundation. Apache Commons Collections. http://commons.apache.org/collections/.

[6] Apache Software Foundation. Apache Log4j. http://logging.apache.org/log4j/.

[7] Apache Software Foundation. Apache Lucene. http://lucene.apache.org/.

[8] Apache Software Foundation. Joda Time. http://joda-time.sourceforge.net/.

[9] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. ISSTA, pages 273–282, 2008.

[10] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE, pages 33–42, 2010.

[11] M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz. Propagation of JML non-null annotations in Java programs. In *PPPJ 2006:Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 135–140, 2006.

[12] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: adaptive random testing for object-oriented software. ICSE, pages 71–80, 2008.

[13] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. ISSTA, pages 85–96, 2010.

[14] Z. P. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker. Analysing source code: looking for useful verb-direct object pairs in all the right places. *IET Software Special Issue on Natural Language in Software Development*, 2008.

[15] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. In *MSR'10*.

[16] D. Gilbert and T. Morgner. JFreeChart. http://www.jfree.org/jfreechart/.

[17] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE*, 2007.

[18] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: Object capture-based automated testing. In *ISSTA*, pages 159–170, 2010.

[19] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in PostgreSQL. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*.

[20] N. Khamis, R. Witte, and J. Rilling. Automatic quality assessment of source code comments: the javadocminer. NLDB, pages 68–79, 2010.

[21] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? – An empirical study of bug characteristics in modern open source software. In *ASID'06*.

[22] F. Long, X. Wang, and Y. Cai. API hyperlinking via structural overlap. ESEC/FSE, pages 203–212, 2009.

[23] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. E. Hassan. Understanding the rationale for updating a function's comment. In *ICSM*, pages 167–176, 2008.

[24] C. D. Manning and H. Schütze. *Foundations Of Statistical Natural Language Processing*. The MIT Press, 2001.

[25] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR'09*.

[26] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, 2005.

[27] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .net with feedback-directed random testing. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA, pages 87–96, 2008.

[28] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. ICSE, pages 75–84, 2007.

[29] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE*, pages 499–510, 2007.

[30] K. A. Smith and D. Kramer. Requirements for writing Java API specifications. http://www.oracle.com/technetwork/java/javase/documentation/index-142372.html.

[31] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. ASE, pages 43–52, 2010.

[32] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. ICSE, pages 101–110, 2011.

[33] Sun. The standard doclet. http://download.oracle.com/javase/1,5.0/docs/guide/javadoc/standard-doclet.html.

[34] Sun. Sun doc check doclet. http://java.sun.com/j2se/javadoc/doccheck/docs/index.html.

[35] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *ICSE*, pages 45–54, 2010.

[36] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *SOSP*, October 2007.

[37] L. Tan, Y. Zhou, and Y. Padioleau. aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs. In *ICSE*, May 2011.

[38] University of Waterloo and Universitaet Koblenz-Landau and Trusted Computer Solutions. Glazed Lists. http://www.glazedlists.com/.

[39] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, pages 461–470, May 2008.

[40] J. woo Park, M. woong Lee, J. Kim, S. won Hwang, and S. Kim. CosTriage: A cost-aware triage algorithm for bug reporting systems. In *AAAI*, 2011.

[41] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[42] A. T. 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 *MSR*.

[43] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, 2011.

[44] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *ASE*, Nov. 2011.

[45] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, 2009.