@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. Javadoc comments are widely used in Java for API specifications. API developers write Javadoc comments, and API users read these comments to understand the API, e.g., reading a Javadoc comment for a method instead of reading the method body. 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 method callers to introduce faults in their code.**

We present a novel approach, called @TCOMMENT, for testing Javadoc comments, specifically method properties about null values and related exceptions. Our approach consists of two components. The first component takes as input source files for a Java project and automatically analyzes the English text in Javadoc comments to infer a set of likely properties for a method in the files. The second component generates random tests for these methods, checks the inferred properties, and reports inconsistencies. We evaluated @TCOMMENT on seven open-source projects and found 29 inconsistencies between Javadoc comments and method bodies. We reported 16 of these inconsistencies, and 5 have already been confirmed and fixed by the developers.

***Keywords*-comment analysis; random testing; test generation**

I. INTRODUCTION

Source code comments are important artifacts in software and have been around as long as code has been around. While comments are not executed, they aid in many tasks such as code comprehension, reuse, or maintenance. 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 [27]. 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 [38].

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 much progress in the recent decades [28], they are still challenged by ambiguities inherent in understanding general text. For example, consider the Javadoc snippet "`@param chrono Chronology to use, null means default`", which describes a method parameter `chrono` that is an object of type `Chronology`. The part "`null means default`" is hard to understand; 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 value 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. [38], [39] pioneered automated checking of comment-code inconsistencies based on NLP analysis. Their iComment [38] and aComment [39] 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 comment analysis, specifically method properties for *null values and related exceptions* in Java libraries/frameworks.

This domain was not studied in the previous work on detecting comment-code inconsistencies, but our inspection of several Java projects showed this domain to be 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 [32] for random test generation of Java code. Randoop randomly explores 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 [32], 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 [21], 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 some tests as likely false alarms in Randoop so that developers can focus on the true faults. A false alarm is a test that Randoop generates but that does *not* find fault in the code under test, e.g., a test that causes a method to throw exception, but the exception is expected according to the Javadoc comment for the method.

Evaluation: We applied @tComment on seven open-source Java projects that have well-developed, well-documented, and well-tested code. We found 29 methods with inconsistencies between Javadoc comments and method bodies in these projects. We reported some of these inconsistencies; 5 of them have been already confirmed and fixed by the developers (one by fixing the code and four by fixing the comment), while the rest await confirmation by the developers. @tComment automatically inferred 2479 properties regarding null values and related exceptions from Javadoc comments with a high accuracy of 97–100%. The high accuracy was achieved without using NLP techniques, largely due to the Javadoc null-related comments being well structured with few paraphrases and variants.

```
// 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 when 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.

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 `JodaTime` project [9], a widely used Java library for representing dates and times. `JodaTime` provides several classes that support multiple calendar systems. `JodaTime` code is fairly well commented with Javadoc. We ran @tComment to infer properties for the methods in `JodaTime` and to check these properties with @Randoop. For each test that @Randoop generates, it marks whether the test, when executed, violated 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 text before a method *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). @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 JodaTime bug database [10], and JodaTime 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) [1], 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 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 generates test2 as it throws an exception. However, Randoop also generates dozens of

```
// 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 " +
         "when 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.

```
/** ...
 * @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.

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 comment-code inconsistency [2], and the 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

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 PROJECTS USED IN OUR EVALUATION.

also reported this comment-code inconsistency [3], and it is under consideration.

For all examples presented so far, an exception is expected (according to the Javadoc comments), but the method 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. @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`, and Randoop reports as potentially fault-revealing all tests that throw uncaught exceptions. In contrast, @TCOMMENT marks that the exception is expected according to the comment.

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 a 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 [38], [39], we build a domain-specific comment 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 [16], and a large percentage of Javadoc comments (24.2% in the seven 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. For a parameter of non-primitive type, @TCOMMENT infers one of these four kinds of properties: *Null Normal*, *Null Any Exception*, *Null Specific Exception*, or *Null Unknown*.

Table I shows four examples of comment tags and their corresponding inferred properties. 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, then 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 matches developers' intention, 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 reported 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 [12] 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 comment text ("`the collection to add to, must not be null`"). The method parameter, its type, and the full method signature are used later by @Randoop, the test-generation component of @TCOMMENT, to check the generated tests.

@TCOMMENT first extracts all Javadoc `@param` (for parameters of non-primitive types) and `@throws` tags that contain a non-empty free-form text, since a Javadoc tag with an empty free-form text does not describe a *Null Normal*, *Null Any Exception*, or *Null Specific Exception* property. Then, @TCOMMENT 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 neither its `@param` tag nor `@throws` tag describes any other property. In this paper, however, we do not count *Null Unknown* toward the 2479 inferred properties, since one cannot test against these *Null Unknown* properties.

@tComment uses three relatively simple heuristics to analyze the free-form text. First, 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.

Second, for `@throws` tags—say, for example, "`@throws IllegalArgumentException if the id is null`"—the Javadoc doclet parses the tag and outputs the specific exception (`IllegalArgumentException`) and the free-form text ("`if the id is null`"). If the text contains the keyword `null`, @tComment simply splits the text into words and searches each word in the list of all method parameter names generated by the Javadoc doclet. If a valid parameter name is found, —e.g., `id`, @tComment infers the property $id == null => IllegalArgumentException$.

Third, 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 [32], 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 Randoop algorithm, including discarding duplicates and filtering extensible sequences, are available elsewhere [32].

```
// inferredProperties is specific to @Randoop
GenerateSequences(classes, contracts, inferredProperties, 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, inferredProperties)
    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'
      // Mark the sequence as likely consistent
      newSeq.isCommentCodeInconsistency ← unlikely
    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 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 mark the sequence as a comment-code inconsistency, or (3) generate the same sequence as Randoop but mark the sequence as comment-code consistent.

Before we describe the five possible kinds of matches, we describe how @Randoop computes the *set* of expected

	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 parameters with `null`	(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.

exceptions. It handles multiple `null` values, which naturally arise for methods with several parameters of non-primitive types, e.g., `addAll` from Figure 2 has "`Collection collection`" and "`Object[] elements`". If only one parameter is `null`, @Randoop uses the property inferred for that parameter. If two or more parameters are `null`, @Randoop puts in the set all expected exceptions for these parameters, e.g., if we had $collection == null => NullPointerException$ and $elements == null => IllegalArgumentException$, @Randoop would assume that either of the two exceptions is expected. If some parameter with `null` value has the *Null Any Exception* property, then all types of exceptions are expected. Finally, @Randoop adds to the set exceptions that are not null-related but appear in the `@throws` tags for the Javadoc comment of the method under test. For example, the method `arrayIterator(Object[] array, int start, int end)` has one such tag ("`@throws IndexOutOfBoundsException if array bounds are invalid`") and one null-related tag ("`@throws NullPointerException if array is null`"); although `IndexOutOfBoundsException` does not correspond to a `null` input, @tComment always adds it to the set of expected exceptions.

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). To repeat, 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(s) 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 might find the inconsistency. However, these tests would be hard to identify among a large number of tests that Randoop generates (in our evaluation, only 8 tests were *Different Exception* among 1,285 tests that Randoop would generate). In contrast, @Randoop highlights these tests.

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 sequences. 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 marks these sequences as consistent. If @tComment inference is correct for these cases, they are false alarms that Randoop would generate; if the inference is incorrect, @tComment would 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 (non-randomly) generate such additional sequences. This extension remains as future work.

IV. Evaluation

We evaluate @tComment on seven 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 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

Project	Source	Description	Version	# LOC	# Classes	# Methods
`Collections`	[1]	Collection library and utilities	3.2.1	26,323	274	2,943
`GlazedLists`	[7]	List transformations in Java	1.8	5,134	35	456
`JFreeChart`	[8]	Chart creator	1.0.13	72,490	396	5,688
`JodaTime`	[9]	Date and time library	1.6.2	25,916	154	3,094
`Log4j`	[4]	Logging service	1.2	20,987	221	20,987
`Lucene`	[5]	Text search engine	2.9.3	48,201	422	4,017
`Xalan`	[6]	XML transformations	2.7.1	178,549	977	9,505

Table III
SUBJECT PROJECTS AND THEIR BASIC STATISTICS

Project	*Missing Exception* = TI + FA	*Different Exception* = TI + FA	*Unexpected Exception* = TI + FA	*Unknown Status*	*Expected Exception*	*Tested Properties*
`Collections`	12 = 12 + 0	4 = 3 + 1	6 = 0 + 6	94	36	115
`GlazedLists`	0 = 0 + 0	0 = 0 + 0	6 = 1 + 5	151	1	11
`JFreeChart`	1 = 1 + 0	0 = 0 + 0	2 = 2 + 0	127	6	42
`JodaTime`	3 = 3 + 0	0 = 0 + 0	13 = 0 + 13	37	3	31
`Log4j`	1 = 1 + 0	0 = 0 + 0	3 = 0 + 3	186	152	179
`Lucene`	4 = 0 + 4	0 = 0 + 0	2 = 1 + 1	368	2	12
`Xalan`	9 = 5 + 4	0 = 0 + 0	2 = 0 + 2	544	32	43
Total	30 = 22 + 8	4 = 3 + 1	34 = 4 + 30	1507	232	433

				Total
True Inconsistencies (TI)	22	3	4	29
False Alarms (FA)	8	1	30	39

Table IV
OVERALL RESULTS FOR THE DEFAULT CONFIGURATION OF OUR @RANDOOP (NULLRATIO=0.6, TIMELIMIT=3600S)

(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

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 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 split into True Inconsistencies and False Alarms. The last column also shows the number of @TCOMMENT properties that @Randoop checked during test generation.

In total, @Randoop generated 68 tests with potential comment-code inconsistencies. Randoop would not generate 30 of those (column '*Missing Exception*') where methods execute normally while exceptions are expected by the corresponding comments. @Randoop also generates 4 tests where an exception is thrown but different than specified by comments (column '*Different Exception*') and 34 tests where an exception is thrown but normal execution was observed (column '*Unexpected Exception*'). @Randoop generates 1507 tests that throw an exception for cases where no null-related properties were inferred (column '*Unknown Status*'). Finally, @Randoop identifies 232 tests as throwing exceptions expected by the comments (column '*Expected Exception*').

The cells with sums show the split of comment-code inconsistencies reported by @TCOMMENT into those truly inconsistent (summarized in row 'True Inconsistencies') and not (row 'False Alarms'). We inspected all the reports by carefully reading the comments and the code. A subset of reports was also independently inspected by two more students. @TCOMMENT detected 29 previously unknown comment-code inconsistencies and had 39 false alarms.

The sources of false alarms are incorrectly inferred properties for the method with reported violation itself (11 out of 39), missing properties (11 out of 39), and incorrect/missing properties for another method in the sequence (17 out of 39). As an example of the first source, @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 tag does not imply the parameter `filter` cannot be null. Section IV-B discusses inference accuracy in detail. The second source is when a method has at least one parameter missing property (*Null Unknown*) and at least one not missing (*Null Normal*, *Null Any Exception*, or *Null Specific Exception*). @Randoop reports an inconsistency if the method throws an unexpected exception, even if the parameter with *Null Unknown* caused it, because @Randoop does not identify the cause of exception. The third source results in some null values propagating through fields. For example, @TCOMMENT may not know that a constructor for some class should not have a null parameter. If a test passes null, the value is set to some field. Later on,

Project	@param	@throws	@param with null	@throws with null	Properties = *N* + *A* + *S*	Precision [%]			Recall [%]			Accuracy [%]
						N	*A*	*S*	*N*	*A*	*S*	
Collections	700	431	271	207	347=81+47+219	75	92	100	100	100	97	97
GlazedLists	110	55	14	17	19=14+0+5	100	100	100	100	100	100	100
JFreeChart	1808	75	902	3	902=362+537+3	100	100	100	100	100	100	100
JodaTime	802	726	529	96	553=445+23+85	100	75	100	100	100	78	98
Log4j	713	0	488	0	460=243+217+0	100	100	100	100	100	100	100
Lucene	498	373	39	15	67=13+25+29	100	67	100	80	100	100	99
Xalan	699	110	126	6	131=33+93+5	50	100	100	100	100	100	99
Total/Overall	5330	1770	2369	344	2479=1191+942+346	98	98	100	99	100	93	99

Table V
COMMENT ANALYSIS RESULTS. *N* IS FOR *Null Normal*, *A* FOR *Null Any Exception*, AND *S* FOR *Null Specific Exception*.

a method can dereference that field and throw an exception; if the method itself has some null parameters, @Randoop can falsely report an inconsistency (again because it does not identify the real cause of exception).

It is worth pointing out that we set all @TCOMMENT options (e.g., distance of negation words from `null`, treating preconditions as requiring exceptions, ignoring *Null Unknown*, etc.) by looking only at the first six projects. The experiments with `Xalan` were performed with the same options.

B. Comment Analysis Results

Table V shows the comment analysis results. Columns 'param' and 'param with null' show, for parameters of non-primitive types, the total number of `@param` tags and the number of `@param` tags that contain `null`, respectively. Similarly, columns 'throws' and 'throws with null' show the total number of `@throws` tags and the number of `@throws` tags that contain `null`, respectively. In total, 2713 `@param` and `@throws` tags contain `null` in the seven evaluated projects. @TCOMMENT inferred 2479 *Null Normal*, *Null Any Exception*, and *Null Specific Exception* properties from these comments. Of these 2479 properties, 433 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 `@param` (for parameters of non-primitive types) and `@throws` tags with non-empty free-form text from each project, and manually read them and the corresponding inferred properties to check if the inferred properties are correct. The *accuracy* is calculated as the number of correctly analyzed tags in a sample over the total number of tags in the sample. 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.

In addition, we present the standard precision and recall for *Null Normal*, *Null Any Exception*, and *Null Specific Exception* respectively. For example, the precision for *Null Normal* is the proportion of identified *Null Normal* properties that indeed are *Null Normal* properties. The recall for *Null Normal* is the proportion of true *Null Normal* properties in our sample that @TCOMMENT identifies.

Our analysis of the free-form comment text achieves a high accuracy of 97–100% (Column 'Accuracy %') without using NLP techniques as iComment did [38]. In addition, the precisions and recalls are in high nineties to 100% in most cases. One exception is that the precision for *Null Normal* in `Xalan` is only 50%, where only two *Null Normal* properties are inferred, and one was inferred incorrectly. The general high performance 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. While the general idea of detecting comment-code inconsistencies through testing should be applicable to C/C++ projects, the comment analysis component may need to leverage more advanced techniques as iComment did [38].

If some null-related properties are described in non-Javadoc style comments, e.g., with no `@param` tag, @TCOMMENT would not analyze them. As we do not anticipate many such comments, this paper focused on properly 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 budget allows, users can run @TCOMMENT with many `nullRatios` and `timeLimits` 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 seven projects and measured the most important metric, the number of *Missing Exception* tests. These are 5*11*7, a total of 385, 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, `nullRatio` 0.6 helps @Randoop identify the largest number of *Missing Exception* tests across all seven projects. Therefore, we chose it as the default value. We found that 0.3, 0.5, 0.7, 0.8, and 0.9 are the next best values. Note that 0.0 is clearly not good as it never selects `null`, but also 1.0 is not good as it always selects `null` and thus Randoop cannot "grow" bigger objects with non-`null` parameters.

To further understand the effect of `timeLimits`, we increased the `timeLimits` to up to two hours with `nullRatio` 0.6. 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 [31].

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 they produce almost identical numbers for the five kinds of matches, suggesting that if one runs @Randoop for an hour, one 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 faults [17], [18], [22], [30], [32], [44], e.g., based on random generation or using specifications. @tComment leverages an additional source—code comments—and modifies Randoop [32] to detect more faults (in both code and comments), and to identify false alarms generated by Randoop. It is quite conceivable to extend @tComment to improve other automated testing techniques.

Detecting Comment-Code Inconsistencies. iComment [38] and aComment [39] extract rules from comments and check source code against these rules *statically*. The differences between iComment/aComment and @tComment have already been discussed in the introduction, so 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 can find false alarms generated by Randoop.

A recent empirical study [24] 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. Doc Check [11] 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 [23], [27], [42], [43]. None of them automatically analyze comments to detect comment-code inconsistencies or improve automated testing.

Comment Inference from Source Code. Several recent projects infer comments for failed test cases [45], exceptions [14], API function cross-references [26], software changes [15], and semantically related code segments [35], [36]. Comments automatically generated by these techniques are more structured than developer-written comments; therefore, it may be easier to leverage such automatically-generated comments for finding inconsistencies. However, it is still beneficial to improve the analysis of developer-written comments because (1) millions of 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 [38] to help us detect more faults. FailureDoc [45] augments a failed test with debugging clues, which could be extended to help explain why the tests generated by @tComment fail.

Analysis of Natural-Language Text for Software. Various research projects analyze natural-language artifacts such as bugs reports [13], [20], [25], [29], [34], [37], [40], [41], API documentation [46], and method names [19] for different purposes such as detecting duplicate bug reports or identifying the appropriate developers to fix bugs. @tComment analyzes comments written in a natural language to detect comment-code inconsistencies and to improve automated testing. Rubio-González et al. detect error code mismatches between code and manual pages in the Linux kernel by combining static analysis and heuristics [33]. Different from some of these studies [19], [46] 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 a high accuracy of 97–100%, partially due to the more structured Javadoc comments with less paraphrases and variants.

VI. Conclusions and Future Work

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 seven open-source projects discovered 29 methods with inconsistencies between Javadoc comments and bodies. We reported 16 inconsistencies, and 5 were already confirmed and fixed by the developers.

In the future, @Randoop can be extended to (1) modify the random selection performed by Randoop such that it biases the selection based on the properties inferred by @tComment; (2) identify some causes of exceptions to reduce the rate of false alarms; and (3) rank the reported inconsistencies. @tComment can be extended to handle other types of properties and to be integrated with other testing or static analysis tools.

Acknowledgments. We thank Sai Zhang for help with Randoop and Aditya Dasgupta and James Tonyan for help in labeling properties inferred by @tComment and comment-code inconsistencies reported by @tComment. The work is partially supported by the US National Science Foundation under Grant Nos. CCF-0916715, CCF-0916350, and CCF-0746856, the National Science and Engineering Research Council of Canada, and a Google gift grant.

References

[1] Apache Commons Collections. http://commons.apache.org/collections/.

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

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

[4] Apache Log4j. http://logging.apache.org/log4j/.

[5] Apache Lucene. http://lucene.apache.org/.

[6] Apache Xalan. http://xml.apache.org/xalan-j/.

[7] Glazed Lists. http://www.glazedlists.com/.

[8] JFreeChart. http://www.jfree.org/jfreechart/.

[9] Joda Time. http://joda-time.sourceforge.net/.

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

[11] Sun Doc Check Doclet. http://www.oracle.com/technetwork/java/javase/documentation/index-141437.html.

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

[13] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE*, 2006.

[14] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *ISSTA*, 2008.

[15] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *ASE*, 2010.

[16] M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz. Propagation of JML non-null annotations in Java programs. In *PPPJ*, 2006.

[17] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: Adaptive random testing for object-oriented software. In *ICSE*, 2008.

[18] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA'10*.

[19] 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*, 2008.

[20] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. In *MSR*, 2010.

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

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

[23] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in PostgreSQL. In *MSR*, 2006.

[24] N. Khamis, R. Witte, and J. Rilling. Automatic quality assessment of source code comments: the JavadocMiner. In *NLDB*, 2010.

[25] 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*, 2006.

[26] F. Long, X. Wang, and Y. Cai. API hyperlinking via structural overlap. In *ESEC/FSE*, 2009.

[27] 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*, 2008.

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

[29] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR*, 2009.

[30] C. Pacheco and M. D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In *ECOOP*, 2005.

[31] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .net with feedback-directed random testing. In *ISSTA*, 2008.

[32] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *ICSE*, 2007.

[33] C. Rubio-González and B. Liblit. Expect the unexpected: error code mismatches between documentation and the real world. In *PASTE*, 2010.

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

[35] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *ASE*, 2010.

[36] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *ICSE*, 2011.

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

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

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

[40] 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*, 2008.

[41] 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.

[42] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *ICSE*, 1981.

[43] 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*, 2005.

[44] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, 2011.

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

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