information

Article

Source Code Documentation Generation Using Program Execution †

Matúš Sulír * and Jaroslav Porubän

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovakia; jaroslav.poruban@tuke.sk
* Correspondence: matus.sulir@tuke.sk; Tel.: +421-55-602-2518

† This paper is an extended version of our paper published in SLATE 2017: Generating Method Documentation Using Concrete Values from Executions.

Received: 30 September 2017; Accepted: 14 November 2017; Published: 17 November 2017

Abstract: Automated source code documentation approaches often describe methods in abstract terms, using the words contained in the static source code or code excerpts from repositories. In this paper, we describe DynamiDoc: a simple automated documentation generator based on dynamic analysis. Our representation-based approach traces the program being executed and records string representations of concrete argument values, a return value and a target object state before and after each method execution. Then, for each method, it generates documentation sentences with examples, such as "When called on [3, 1.2] with element = 3, the object changed to [1.2]". Advantages and shortcomings of the approach are listed. We also found out that the generated sentences are substantially shorter than the methods they describe. According to our small-scale study, the majority of objects in the generated documentation have their string representations overridden, which further confirms the potential usefulness of our approach. Finally, we propose an alternative, variable-based approach that describes the values of individual member variables, rather than the state of an object as a whole.

Keywords: documentation generation; source code summarization; method documentation; dynamic analysis; examples

1. Introduction

Developers frequently try to comprehend what a particular method or procedure in source code does or what are its inputs and outputs. In such cases, they often turn to documentation. For instance, in Java, the methods can be documented by comments specially formatted according to the Javadoc specification [1]. Similar standards and possibilities exist in other languages.

However, the API (application programming interface) documentation is often incomplete, ambiguous, obsolete, lacking good examples, inconsistent or incorrect [2]. In the worst case, it is not present at all.

Consider the following simplified excerpt from Java 8 API (https://docs.oracle.com/javase/8/docs/api/java/net/URL.html#getAuthority--): a method `getAuhority()` in the class `URL` and its documentation:

```
public class URL {
  ...
  /**
   * Gets the authority part of this URL.
   * @return the authority part of this URL
   */
```

```
 public String getAuthority() {...}
}
```

For a person who is not a domain expert in the field of Internet protocols, this documentation is certainly not as useful as it should be. To get at least partial understanding of what the method does, he/she must study the corresponding RFC (request for comments) documents, various web tutorials and resources or browse the rest of the lengthy API documentation.

Now, consider the same method, but with a documentation containing concrete examples of arguments and return values:

```
public class URL {
 ...
 /**
  * Gets the authority part of this URL.
  * @return the authority part of this URL
  * @examples When called on http://example.com/path?query,
  *  the method returned "example.com".<br>
  *  When called on http://user:password@example.com:80/path,
  *  the method returned "user:password@example.com:80".
  */
 public String getAuthority() {...}
}
```

This kind of documentation should give a developer instant "feeling" of what the method does and help him/her to understand the source code.

Creating and maintaining documentation manually is time-consuming and error-prone. There exist multiple approaches for automated documentation generation [3]. Many of these approaches work by combining static analysis with natural language processing (NLP) [4,5] or repository mining [6]. NLP-based and static analysis approaches have an inherent disadvantage: they only present information already available in the static source code in another way. Mining-based methods require large repositories of code using concerned APIs. Furthermore, none of the mentioned approaches provide concrete, literate string representations of arguments, return values and states from runtime: at best, they provide code examples that use the API. While there exist dynamic analysis approaches collecting run-time values of variables [7,8], they are not oriented toward textual documentation generation.

In this paper, we first describe a representation-based method documentation approach, devised in our conference article [9]: The program of interest is executed either manually or using automated tests. During these executions, a tracer saves string representations (obtained by calling a `toString()`-like method) of the methods' arguments, return value and object states before and after executing the method. For each method, a few sample executions are chosen, and documentation sentences similar to the exhibit shown above are generated. The generated Javadoc documentation is then written into the source files.

To show the feasibility of our approach, a prototype implementation named DynamiDoc was constructed. It is available at https://github.com/sulir/dynamidoc. We applied DynamiDoc to multiple open-source projects and performed qualitative evaluation: we described its current benefits and drawbacks.

Next, as an addition to our original article, we perform a small-scale quantitative evaluation of the representation-based approach. The lengths of the generated documentation sentences are analyzed and compared to the lengths of the methods they describe. Furthermore, we look at the proportion of objects in the generated documentation that have a custom (overridden) string representation, which is a prerequisite for this approach to be useful.

Finally, we devise a new (not yet implemented) variable-based approach, which describes individual member variable values during method execution. This brings more details to the generated documentation and reduces the need for overridden string representations.

2. Representation-Based Documentation Approach

Now, we will describe the representation-based documentation approach in more detail. Our method consists of three consecutive phases: tracing, selection of examples and documentation generation.

2.1. Tracing

First, all methods in the project we want to document are instrumented to enable tracing. Each method's definition is essentially replaced by the code presented in Listing 1. In our implementation, we used the bytecode-based AspectJ instrumentation; however, the approach is not limited to it.

```
 1 function around(method)
 2  // save string representations of arguments and object state
 3  arguments ← []
 4  for arg in method.args
 5   arguments.add(to_string(arg))
 6  end for
 7  before ← to_string(method.this)
 8
 9  // run the original method, save return value or thrown exception
10  result ← method(method.args)
11  if result is return value
12   returned ← to_string(result)
13  else if result is thrown exception
14   exception ← to_string(result)
15  end if
16
17  // save object representation again and write record to trace file
18  after ← to_string(method.this)
19  write_record(method, arguments, before, returned, exception, after)
20
21  // proceed as usual (not affecting the program's semantics)
22  return/throw result
23 end function
```

Listing 1: The tracing algorithm executed around each method.

The target project is then executed as usual. This can range from manual clicking in a graphical user interface of an application to running fully-automatized unit tests of a library. Thanks to the mentioned instrumentation, selected information about each method execution is recorded into a trace file. A detailed explanation of the tracing process for one method execution is below.

First, all parameter values are converted to their string representations (Lines 3–6 in Listing 1). By a string representation, we mean a result of calling the `toString()` method on an argument. (There are some exceptions; for example, on arrays, we call `Arrays.deepToString(arr)`.) For simple numeric and string types, it is straightforward (e.g., the number 7.1 is represented as `7.1`). For more complicated objects, it is possible to override the `toString()` method of a given class to meaningfully represent the object's state. For instance, `Map` objects are represented as `{key1=value1, key2=value2}`.

Each non-static method is called on a specific object (called `this` in Java), which we will call a "target object". We save the target object's string representation before actual method execution (Line 7). This should represent the original object state. In our example from Section 1, a string representation of a URL object constructed using `new URL("http://example.com/path?query");` looks like `http://example.com/path?query`.

We execute the given method and convert the result to a string (Lines 10–15). For non-void methods, this is the return value. In case a method throws an exception, we record it and convert to string; even exceptions have their `toString()` method. In the example we are describing, the method returned `example.com`.

After the method completion, we save again the string representation of the target object (`this`, Line 18) if the method is non-static. This time, it should represent the state affected by the method execution. Since the `getAuthority()` method does not mutate the state of a URL object, it is the same as before calling the method.

Finally, we write the method location (the file and line number) along with the collected string representations to the trace file. We return the stored return value or throw the captured exception, so the program execution continues as it would do without our tracing code.

To sum up, a trace is a collection of stored executions, where each method execution is a tuple consisting of:

- *method*: the method identifier (file, line number),
- *arguments*: an array of string representations of all argument values,
- *before*: a string representation of the target object state before method execution,
- *return*: a string representation of the return value, if the method is non-void and did not throw an exception,
- *exception*: a thrown exception converted to a string (if it was thrown),
- *after*: a string representation of the target object state after method execution.

2.2. Selection of Examples

After tracing finishes, the documentation generator reads the written trace file, which contains a list of all method executions. Since one method may have thousands of executions, we need to select a few most suitable ones: executions that will be used as examples for documentation generation. Each execution is assigned a metric representing its suitability to be presented as an example to a programmer. They are then sorted in descending order according to this metric, and the first few of them are selected. In the current implementation, this number is set to five. The higher the number of selected examples, the more comprehensive is the documentation. However, a too high number produces a lengthy documentation with a nontrivial reading effort.

In the current version of DynamiDoc, we use a very simple metric: execution frequency. It is the number of times that the method was executed in the same state with the same arguments and return value (or the thrown exception) and resulted in the same final state; considering the stored string representations. This means we consider the most frequent executions the most representative and use them for documentation generation. For instance, if the `getAuthority()` method was called three times on `http://example.com/path?query` producing `"example.com"` and two times on `http://user:password@example.com:80/path` producing `"user:password@example.com:80"`, these two examples are selected, in the given order.

2.3. Documentation Generation

After obtaining a list of a few example executions for each method, a documentation sentence is generated for each such execution. The generation process is template-based.

First, an appropriate sentence template is selected, based on the properties of the method (static vs. non-static, parameter count and return type) and the execution (whether an exception was thrown or a string representation of the target object changed by calling the method). The selection is performed using a decision table displayed in Table 1. For instance, the `getAuthority()` method is non-static (instance); it has zero parameters; it does not have a void type; its execution did not throw an exception; and the URL's string representation is the same before and after calling it (the state did not change from our point of view). Therefore, the sentence template "When called on {before}, the method returned {return}." is selected.

Table 1. A decision table for the documentation sentence templates.

Method Kind	Parameters	Return Type	Exception	State Changed	Sentence Template
static	0	void	no	no	-
		non-void			The method returned {return}.
		any	yes		The method threw {exception}.
	≥1	void	no		The method was called with {arguments}.
		non-void			When {arguments}, the method returned {return}.
		any	yes		When {arguments}, the method threw {exception}.
instance	0	void	no	no	The method was called on {before}.
				yes	When called on {before}, the object changed to {after}.
		non-void		no	When called on {before}, the method returned {return}.
				yes	When called on {before}, the object changed to {after} and the method returned {return}.
		any	yes	no	When called on {before}, the method threw {exception}.
				yes	When called on {before}, the object changed to {after} and the method threw {exception}.
	≥1	void	no	no	The method was called on {before} with {arguments}.
				yes	When called on {before} with {arguments}, the object changed to {after}.
		non-void		no	When called on {before} with {arguments}, the method returned {return}.
				yes	When called on {before} with {arguments}, the object changed to {after} and the method returned {return}.
		any	yes	no	When called on {before} with {arguments}, the method threw {exception}.
				yes	When called on {before} with {arguments}, the object changed to {after} and the method threw {exception}.

Next, the placeholders (enclosed in braces) in the sentence template are replaced by actual values. The meaning of individual values was described at the end of Section 2.1. An example of a generated sentence is: 'When called on http://example.com/path?query, the method returned "example.com".' Note that we use past tense since in general, we are not sure the method always behaves the same way; the sentences represent some concrete recorded executions.

Finally, we write the sentences into Javadoc documentation comments of affected methods. Javadoc documentation is structured: it usually contains tags such as `@param` for a description of a parameter and `@see` for a link to a related class or method. We append our new, custom `@examples` tag with the generated documentation sentences to existing documentation. If the method is not yet documented at all, we create a new Javadoc comment for it. When existing examples are present, they are replaced by the new ones. The original source code files are overwritten to include the modified documentation. Using a custom "doclet" [1], the `@examples` tag can be later rendered as the text "Examples:" in the HTML version of the documentation.

3. Qualitative Evaluation

We applied our representation-based documentation approach to three real-world open source projects and observed its strengths and weaknesses by inspecting the generated documentation.

The mentioned open source projects are:

- Apache Commons Lang (https://commons.apache.org/lang/),
- Google Guava (https://github.com/google/guava),
- and Apache FOP (Formatting Objects Processor; https://xmlgraphics.apache.org/fop/).

The first two projects are utility libraries aiming to provide core functionality missing in the standard Java API. To obtain data for dynamic analysis, we executed selected unit tests of the libraries. The last project is a Java application reading XML files containing formatting objects (FO) and writing files suitable for printing, such as PDFs. In this case, we executed the application using a sample FO file as its input.

A description of selected kinds of situations we encountered and observations we made follows.

3.1. Utility Methods

For simple static methods accepting and returning primitive or string values, our approach generally produces satisfactory results. As one of many examples, we can mention the method `static String unicodeEscaped(char ch)` in the "utility class" `CharUtils` of Apache Commons Lang. The generated sentences are in the form:

```
When ch = 'A', the method returned "\u0041".
```

For many methods, the Commons Lang API documentation already contains source code or pseudo-code examples. Here is an excerpt from the documentation of the aforementioned method:

```
CharUtils.unicodeEscaped('A') = "\u0041"
```

Even in cases when a library already contains manually written code examples, DynamiDoc is useful for utility methods on simple types:

- to save time spent writing examples,
- to ensure the documentation is correct and up-to-date.

The latter point is fulfilled when the tool is run automatically, e.g., as a part of a build process. Sufficient unit test coverage is a precondition for both points.

3.2. Data Structures

Consider the data structure `HashBasedTable` from Google Guava and its method `size()` implemented in the superclass `StandardTable`. The DynamiDoc-generated documentation includes this sentence:

```
When called on {foo={1=a, 3=c}, bar={1=b}}, the method returned 3.
```

Compare it with a hypothetical manually-constructed source-code-based example:

```
Table<String, Integer, Character> table = HashBasedTable.create();
table.put("foo", 1, 'a');
table.put("foo", 3, 'c');
table.put("bar", 1, 'b');
System.out.println(table.size()); // prints 3
```

Instead of showing the whole process of how we got to the given state, our approach displays only a string representation of the object state (`{foo={1=a, 3=c}, bar={1=b}}`). Such a form is very compact and still contains sufficient information necessary to comprehend the gist of a particular method.

3.3. Changing Target Object State

When the class of interest has the method `toString()` meaningfully overwritten, DynamiDoc works properly. For instance, see one of the generated documentation sentences of the method `void FontFamilyProperty.addProperty(Property prop)` in Apache FOP:

```
When called on [sans-serif] with prop = Symbol, the object changed to
[sans-serif, Symbol].
```

Now, let us describe an opposite extreme. In the case of Java, the default implementation of the `toString()` method is not very useful: it displays just the class name and the object's hash code. When the class of interest does not have the `toString()` method overridden, DynamiDoc does not produce documentation of sufficient quality. Take, for example, the generated documentation for the method `void LayoutManagerMapping.initialize()` in the same project:

```
The method was called on
org.apache.fop.layoutmgr.LayoutManagerMapping@260a3a5e.
```

While the method probably changed the state of the object, we cannot see the state before and after calling it. The string representation of the object stayed the same (and is not very meaningful).

Although this behavior is a result of an inherent property of our approach, there exists a way that this situation can be improved: to override `toString()` methods for all classes when it can be at least partially useful. Fortunately, many contemporary IDEs (Integrated Development Environments) support automated generation of `toString()` source code. Such generated implementations are not always perfect, but certainly better than nothing.

We plan to perform an empirical study assessing what portion of existing classes in open source projects meaningfully override the `toString()` method. This will help us to quantitatively assess the usefulness of DynamiDoc.

3.4. Changing Argument State

The current version of DynamiDoc does not track changes of the passed parameter values. For example, the method `static void ArrayUtils.reverse(int[] array)` in Apache Commons Lang modifies the given array in-place, which is not visible in the generated documentation:

```
The method was called with array = [1, 2, 3].
```

Of course, it is possible to compare string representations of all mutable objects passed as arguments before and after execution. We can add such a feature to DynamiDoc in the future.

3.5. Operations Affecting the External World

Our approach does not recognize the effects of input and output operations. When such an operation is not essential for the method, i.e., it is just a cross-cutting concern like logging, it does not affect the usefulness of DynamiDoc too much. This is, for instance, the case of the method `static int FixedLength.convert(double dvalue, String unit, float res)` in Apache FOP. It converts the given length to millipoints, but also contains code that logs an error when it occurs (e.g., to a console). A sample generated sentence follows:

```
When dvalue = 20.0, unit = "pt" and res = 1.0, the method returned
20000.
```

On the other hand, DynamiDoc is not able to generate any documentation sentence for the method `static void CommandLineOptions.printVersion()`, which prints the version of Apache FOP to standard output.

3.6. Methods Doing Too Much

Our approach describes methods in terms of their overall effect. It does not analyze individual actions performed during method execution. Therefore, it is difficult to generate meaningful documentation for methods such as application initializers, event broadcasters or processors. An example is the method `void FObj.processNode(String elementName, Locator locator, Attributes attlist, PropertyList pList)`. An abridged excerpt from the generated documentation follows.

```
The method was called on ...RegionAfter@206be60b[@id=null] with
elementName = "region-after", locator = ...LocatorProxy@292158f8,
attlist = ...AttributesProxy@4674d90
and pList = ...StaticPropertyList@6354dd57.
```

3.7. Example Selection

The documentation of some methods is not the best possible one. For instance, the examples generated for the method `BoundType Range.lowerBoundType()` in Google Guava are:

```
When called on (5..+∞), the method returned OPEN.
When called on [4..4], the method returned CLOSED.
When called on [4..4), the method returned CLOSED.
When called on [5..7], the method returned CLOSED.
When called on [5..8), the method returned CLOSED.
```

This selection is not optimal. First, there is only one example of the OPEN bound type, but this is only a cosmetic issue. The second, worse flaw is the absence of a case when the method throws an exception (`IllegalStateException` when the lower bound is $-\infty$).

The method `Range encloseAll(Iterable values)` in the same class has much better documentation, which shows the variety of inputs and outputs (although ordering could be slightly better):

```
When values = [0], the method returned [0..0].
When values = [5, -3], the method returned [-3..5].
When values = [0, null], the method threw
  java.lang.NullPointerException.
When values = [1, 2, 2, 2, 5, -3, 0, -1], the method returned [-3..5].
When values = [], the method threw java.util.NoSuchElementException.
```

Improvement of the example selection metric will be necessary in the future. A possible option is to include the most diverse examples: some short values, some long, plus a few exceptions.

Furthermore, like in any dynamic analysis approach, care must be taken not to include sensitive information like passwords in the generated documentation.

4. Quantitative Evaluation

In this section, we will present two small-scale studies of the selected aspects that could affect the usability of the representation-based documentation approach in practice. First, we will look at the length of the generated documentation. Second, a study of the overridden vs. default string representation in the generated documentation is performed.

4.1. Documentation Length

In order to be useful, a summary of the method should be as short as possible, while retaining necessary information. Therefore, we decided to perform a quantitative evaluation regarding the length of generated summaries.

4.1.1. Method

First, we ran all tests from the package `org.apache.commons.lang3.text` from Apache Commons, while recording execution data. Next, we generated the documentation sentences (at most five for each method). Finally, we computed statistics regarding the length of the generated documentation sentences and the length of the methods' source code.

The following metrics were computed:

- the length of each sentence in characters,
- the "proportional length": the length of each sentence divided by the length of the corresponding method's source code (in characters).

Note that the proportional length was chosen only as an arbitrary relative measure to compare the length of the sentence to the length of the method. We do not try to say that one sentence is enough to comprehend the whole method; it is difficult to automatically determine what number of sentences is necessary to comprehend a method without performing a study with human subjects. In general, no number of documentation sentences can precisely describe possibly infinite scenarios encompassed in the method's source code.

4.1.2. Results

In Figure 1, there is a histogram of the lengths of individual documentation sentences. The average (mean) length of a sentence is 140 characters, and the median is 117.

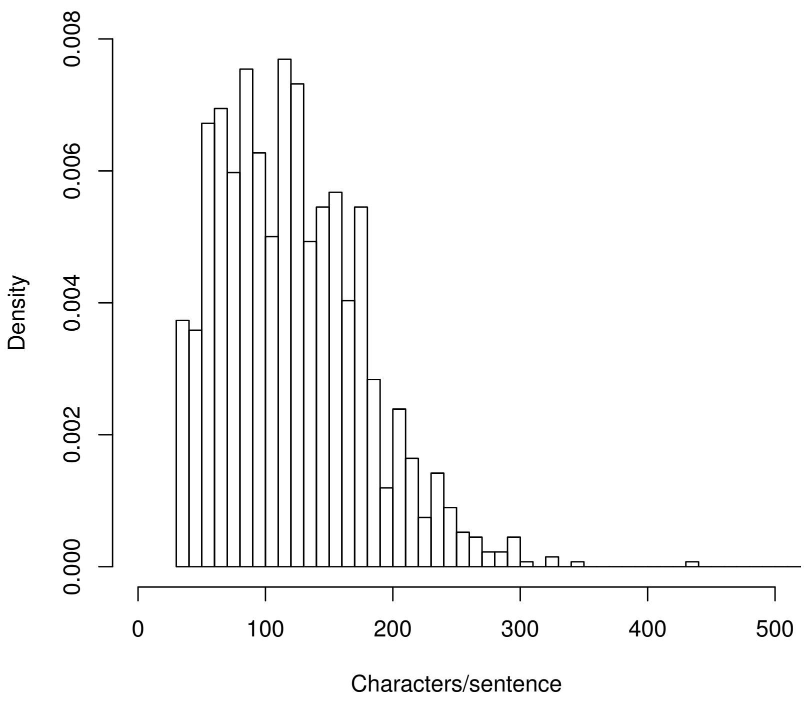

Figure 1. Length of a documentation sentence (histogram).

In the histogram, we can see the vast majority of documentation sentences are 30–300 characters long.

Figure 2 displays a histogram of proportional lengths. On average, one documentation sentence has 0.104 of the length of the method it describes. The median value is 0.096.

It is visible that a vast majority of documentation sentences is shorter than 20–30% of the method it describes.

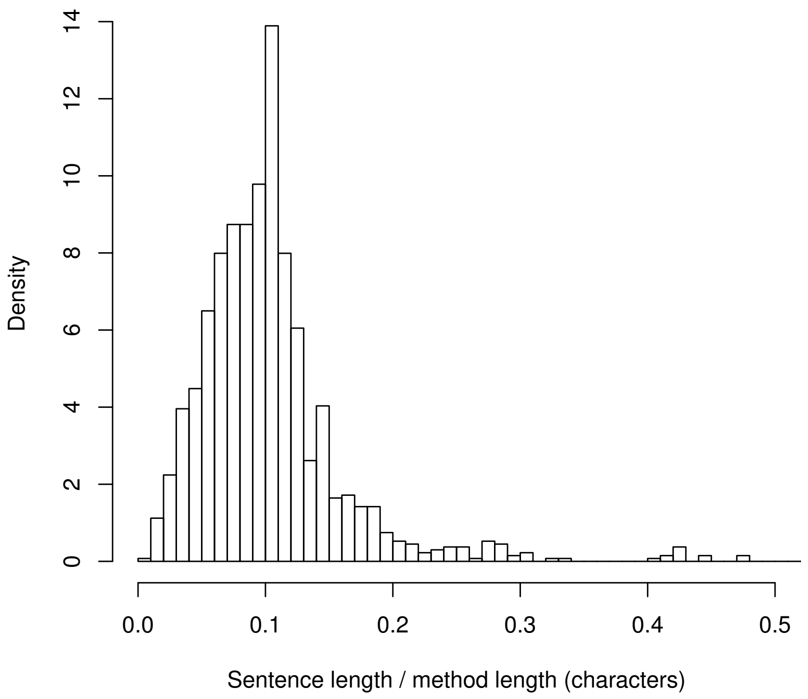

Figure 2. Length of a documentation sentence divided by the length of the method it describes (histogram).

4.1.3. Threats to Validity

It is disputable whether the length of a sentence is a good measure of comprehensibility. However, it is clear that too long sentences would require more time to read. By ensuring the sentences are short enough, we filled the first requirement of a good summary.

While documentation sentences are written in natural language, the methods' source code is written in a programming language. Therefore, comparing these two lengths might not be fair. However, our aim was only to roughly estimate the reading effort, rather than perform a comprehension study.

Regarding external validity, the presented results were obtained by an analysis of a single package of tests in a specific library. However, we inspected also other packages, which provided results not too different from the presented ones.

4.1.4. Conclusions

By measuring the length of sentences and methods, we tried to estimate the reading effort of the documentation and the method source code itself.

According to the results, an average sentence is around 10-times shorter than the method it describes. Of course, reading one example (sentence) is rarely sufficient. Nevertheless, even if the programmer reads three sentences per method, it would be only 30% of the full method's source code length.

In this study, we did not try to determine whether the generated documentation sentences contain enough information to be useful. This would require performing an experiment with human participants, and it is left for future work.

4.2. Overridden String Representation in Documentation

For our documentation to be meaningful, the `toString` method should be overridden in as many classes as possible. In the following small-scale study, we tried to estimate what proportion of objects contained in the generated documentation (target objects, parameters, return values) had this method overridden and what proportion relied on the default implementation.

4.2.1. Method

We used the documentation of Apache Commons Lang, Google Guava and Apache FOP, generated during the qualitative analysis. From each of these three projects, we randomly selected 10 automatically documented methods. For every method, we selected the first documentation sentence.

Next, we manually inspected these 30 sentences. Each of the sentences contained one or more representations of target objects, parameters or return values. For every string representation, we determined whether it is default (only the class name and the hash code) or overridden (any representation other than default).

4.2.2. Results

The results are presented in Table 2. We can conclude that in our study, 87.5% of the documented objects had the `toString` method meaningfully overridden. The other 12.5% relied on the default implementation, which displays only the class name and the object's hash code.

Table 2. The numbers of objects with overridden/default string representations in randomly-sampled documentation excerpts.

Project	Overridden	Default
Apache Commons Lang	27	3
Google Guava	22	5
Apache FOP	21	2
Total	70 (87.5%)	10 (12.5%)

Although the number of objects with an overridden implementation is relatively high, note that not all overridden implementations are actually useful. Some of the string representations we encountered showed too little of the object's properties, so they did not help in the comprehension of a specific method.

5. Variable-Based Documentation Approach

Although the representation-based documentation approach presented so far generates terse sentences, it may be disadvantageous if the objects it describes do not have meaningful `toString` methods. Therefore, in this section, we would like to present an alternative approach, when each method execution is broken into reads and writes of prevalently primitive-type values. In Java, primitive values include Booleans, characters, integers and floating-point numbers. However, for the purpose of this discussion, we will consider also string and enumeration types primitive.

5.1. Example

First, let us describe the approach on an example. Consider the following two classes (simplified):

```
public class Ticket {
 private String type;

 public void setType(String type) {
  this.type = type;
 }
}
```

```
public class Person {
 private int age;
 private Ticket ticket = new Ticket();

 public int setAge(int age) {
  this.age = age;
  ticket.setType((age < 18) ? "child" : "full");
 }
}
```

For a call of `somePerson.setAge(20)`, the following sentence would be generated:

```
When age was 20, this.age was set to 20 and this.ticket.type was
set to "full".
```

5.2. Approach

Now, we will describe the approach in more detail.

5.2.1. Tracing

First, a system is executed, recording the following events:

- the reads of the method's primitive parameters,
- all reads and writes of primitive variables in all objects/classes,
- the reads of non-primitive objects, but only if they are a part of object comparison (==, `!=`),
- and writes (assignments) of non-primitive objects (`x = object;`).

We try to break objects into the smallest pieces possible: their primitive (and string/enumeration) member variables, so a `toString` method describing each object as a whole is no longer necessary. However, the last two mentioned points are necessary to record actions that cannot be broken into smaller pieces.

For instance, in our example, we are not interested in the read of a non-primitive object `ticket`, which is used only to call its method `setType`. We record only the write of a string variable `type` inside this method.

For each variable, its name is determined as follows:

- For a method's parameter, the name is the parameter's name as it is written in the source code.
- If the variable is a member of this instance, the name is `this.memberName`.
- If the variable is a member of another object, references are gradually recorded to obtain a connection with the current method/object and the other object. In this case, the name is denoted as a chain of references: either `this.member_1. ... .member_n` (if the reference chain begins with a member variable of this instance) or `local_variable.member_1. ... .member_n` (if the reference chain begins with a local variable or a parameter).
- If the variable is static, its name is `ClassName.variableName`.
- Array elements are denoted by square brackets: `array[index]`.

An instance variable contained in another object may be referenced through multiple different reference chains; one variable can have multiple names. In such cases, the names are unified, so each variable has only one name.

5.2.2. Duplicate Removal

Some variables are read or written many times during an execution of a method. Including all such events in the generated sentences would make them too long. Therefore, only the following events are preserved in the trace:

- the first read of each variable; unless the variable was written before it was read for the first time; in this case, we ignore all reads of this variable
- and the last write of each variable.

All other occurrences are deleted; we do not want to describe multiple reads and intermediate changes. This way, we obtain the values of all read variables (which could possibly have an effect on the execution outcome) before the method was executed; and the values of all written variables after the method execution (the outcome).

Here is an example of a recorded trace with "read" and "write" events, the names of the variables and their values:

```
write x 1, read x 1, read y 2, write y 3, write y 4
```

The sample trace is reduced to this one:

```
write x 1, read y 2, write y 4
```

5.2.3. Selection of Examples

Similarly to the original approach, the most representative (currently, the most frequent) executions are selected.

In the future, it would be beneficial if the most diverse values were selected instead of the most frequent ones. Alternatively, the user could select the input values of his/her choice. However, this would make this approach less automated.

5.2.4. Documentation Generation

For each selected execution, the recorded events are divided into read and write events. A documentation sentence in the following format is generated:

```
When {read_1_name} was {read_1_value}, ...
and {read_n_name} was {read_n_value},
{write_1_name} was set to {write_1_value}, ...
and {write_m_name} was set to {write_m_value}.
```

If the method does not read or write any variables, the corresponding part is omitted.

5.3. Discussion

Compared to the representation-based documentation approach, the variable-based one is more detailed. Therefore, the generated sentences may be too long. On the other hand, it has multiple positive properties when compared to the original method.

First, the variable-based approach is less reliant on meaningful `toString` methods. Considering the example from this section, the generated sentence for the original approach would be similar to this: `The method was called on Person@1a2b3c with age = 20`. This is because `Person` has no `toString` method, and its string representation is the same before and after the method call. The variable-based documentation can tackle such situations.

Second, we can see the changes of the parameters' properties. For example, if the parameter `ticket` was passed to some method and it changed its type, we could see the excerpt "ticket.type was changed to ..." in the generated documentation.

Note that even the variable-based approach is not completely immune to meaningless string representations. When non-primitive objects are compared or assigned, their `toString` method is used to determine a string representation.

6. Related Work

In this section, we will present related work with a focus on documentation generation and source code summarization. The related approaches are divided according to the primary analysis type they use: static analysis (sometimes enhanced by repository mining) or dynamic analysis.

6.1. Static Analysis and Repository Mining

Sridhara et al. [4,10] generate natural-language descriptions of methods. The generated sentences are obtained by analyzing the words in the source code of methods; therefore, it does not contain examples of concrete variable values, which can be obtained often only at runtime. In [11], they add support for parameter descriptions, again using static analysis only.

McBurney and McMillan [5] summarize method context also: how the method interacts with other ones. While the approach considers source code outside the method being described, they still use only static analysis.

Buse and Weimer [12] construct natural language descriptions of situations when exceptions may be thrown. These descriptions are generated for methods, using static analysis.

Long et al. [13] describe an approach that finds API functions most related to the given C function. If adapted to Java, it could complement DynamiDoc's documentation by adding `@see` tags to Javadoc.

The tool eXoaDocs [6] mines large source code repositories to find usages of particular API elements. Source code examples illustrating calls to API methods are then added to generated Javadoc documentation. In Section 3.2, we described the difference between source code-based examples and examples based on string representations of concrete variable values. Furthermore, the main point of source code examples is to show how to use the given API. Therefore, the descriptions of results or outputs are often not present in the examples.

Buse and Weimer [14] synthesize API usage examples. Compared to Kim et al. [6], they do not extract existing examples from code repositories, but use a corpus of existing programs to construct new examples.

The APIMiner platform [15] uses a private source code repository to mine examples. The generated Javadoc documentation contains "Examples" buttons showing short code samples and related elements.

6.2. Dynamic Analysis

Hoffman and Strooper [16] present an approach of executable documentation. Instead of writing unit tests in separate files, special markers in method comments are used to mark tests. Each test includes the code to be executed and an expected value of the expression, which serves as specification and documentation. Compared to our DynamiDoc, they did not utilize string representations of objects; the expected values are Java source code expressions. Furthermore, in the case of DynamiDoc, the run-time values can be collected from normal program executions, not only from unit tests.

Concern annotations [17] are Java annotations above program elements such as methods, representing the intent behind the given piece of code. AutoAnnot [18] writes annotations representing features (e.g., `@NoteAdding`) above methods unique to a given feature obtained using simple dynamic analysis. While AutoAnnot describes methods using only simple identifiers, DynamiDoc generates full natural language sentences containing concrete variable values.

ADABU (ADABU Detects All Bad Usages) [19] uses dynamic analysis to mine object behavior models. It constructs state machines describing object states and transitions between them. Compared to our approach, which used class-specific string representations, in ADABU, an object state is described using a predicate like "isEmpty()". Furthermore, they do not provide examples of concrete parameter and return values.

Lo and Maoz [7] introduce a combination of scenario-based and value-based specification mining. Using dynamic analysis, they generate live sequence charts in UML (Unified Modeling Language).

These charts are enriched with preconditions and post-conditions containing string representations of concrete variable values. However, their approach does not focus on documentation of a single method, its inputs and outputs; rather, they describe scenarios of interaction of multiple cooperating methods and classes.

Tralfamadore [8] is a system for analysis of large execution traces. It can display the most frequently-occurring values of a specific function parameter. However, it does not provide a mapping between parameters and a return value. Furthermore, since it is C-based, it does not present string representations of structured objects.

TestDescriber by Panichella et al. [20] executes automatically-generated unit tests and generates natural language sentences describing these tests. First, it differs from DynamiDoc since their approach describes only the unit tests themselves, not the tested program. Second, although it uses dynamic analysis, the only dynamically captured information by TestDescriber is code coverage; they do not provide any concrete variable values except that present literally in the source code.

FailureDoc [21] observes failing unit test execution. It adds comments above individual lines inside tests, explaining how the line should be changed for the test to pass.

SpyREST [22] generates documentation of REST (representational state transfer) services. The documentation is obtained by running code examples, intercepting the communication and generating concrete request-response examples. While SpyREST is limited to web applications utilizing the REST architecture, DynamiDoc produces documentation of any Java program.

@tComment [23] is a tool using dynamic analysis to find code-comment inconsistencies. Unlike DynamiDoc, it does not produce new documentation; it only checks existing, manually-written documentation for broken rules.

7. Conclusions and Future Work

We described DynamiDoc: an automated documentation generator for methods that uses dynamic analysis. Two alternative sub-approaches were presented.

The original, representation-based approach traces program execution to obtain concrete examples of arguments, return values and object states before and after calling a method. Thanks to `toString()` methods, the values are converted to strings during the program runtime, and only these converted values are stored in a trace. For each method, a few execution examples are selected, and natural-language sentences are generated and integrated into Javadoc comments.

We evaluated the representation-based approach using two small-scale studies. We found out that the generated sentences are considerably shorter than the methods they describe. Furthermore, the majority of objects described in the generated sentences have an overridden (non-default) string representation.

An alternative, variable-based approach describes read and written values of individual member variables and parameters during a method execution. This partially alleviates the effects of meaningless string representations.

DynamiDoc should facilitate program comprehension. The potential users of DynamiDoc are software developers who could read the generated documentation to leverage understanding of existing programs during the development. Therefore, it is important to perform a controlled experiment with human subjects and using real comprehension tasks in the future. This experiment should determine whether the documentation produced by our tool really improves program comprehension efficiency.

We do not have empirical findings of what are the attributes of a "useful example". Therefore, we used only a very simple metric of execution frequency to sort the example executions and select the best ones. We would like to investigate, both qualitatively and quantitatively, what examples are considered the best by developers and modify the selection metric accordingly.

Integration with other tools that analyze variable values at runtime, such as RuntimeSearch [24], could make the data collection process during the program execution more efficient. The values would be collected only once and then shared between multiple tools.

Currently, DynamiDoc is a purely textual approach. In the future, we can develop an IDE extension that would visualize the collected dynamic information instead of only generating textual documentation.

Acknowledgments: This work was supported by Project KEGA 047TUKE-4/2016 Integrating software processes into the teaching of programming.

Author Contributions: Matúš Sulír designed parts of the original and new approach, performed the experiments and wrote the paper. Jaroslav Porubän designed parts of the original and new approach.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kramer, D. API Documentation from Source Code Comments: A Case Study of Javadoc. In Proceedings of the 17th Annual International Conference on Computer Documentation, SIGDOC'99, New Orleans, LA, USA, 12–14 September 1999; ACM: New York, NY, USA, 1999; pp. 147–153.
2. Uddin, G.; Robillard, M.P. How API Documentation Fails. *IEEE Softw.* **2015**, *32*, 68–75.
3. Nazar, N.; Hu, Y.; Jiang, H. Summarizing Software Artifacts: A Literature Review. *J. Comput. Sci. Technol.* **2016**, *31*, 883–909.
4. Sridhara, G.; Pollock, L.; Vijay-Shanker, K. Automatically Detecting and Describing High Level Actions Within Methods. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, Waikiki, Honolulu, HI, USA , 21–28 May 2011; ACM: New York, NY, USA, 2011; pp. 101–110.
5. McBurney, P.W.; McMillan, C. Automatic Documentation Generation via Source Code Summarization of Method Context. In Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, 2–3 June 2014; ACM: New York, NY, USA, 2014; pp. 279–290.
6. Kim, J.; Lee, S.; Hwang, S.W.; Kim, S. Adding Examples into Java Documents. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, Auckland, New Zealand, 16–20 November 2009; IEEE Computer Society: Washington, DC, USA, 2009; pp. 540–544.
7. Lo, D.; Maoz, S. Scenario-based and Value-based Specification Mining: Better Together. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, Antwerp, Belgium, 20–24 September 2010; ACM: New York, NY, USA, 2010; pp. 387–396.
8. Lefebvre, G.; Cully, B.; Head, C.; Spear, M.; Hutchinson, N.; Feeley, M.; Warfield, A. Execution Mining. In Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, VEE '12, London, UK, 3–4 March 2012; ACM: New York, NY, USA, 2012; pp. 145–158.
9. Sulír, M.; Porubän, J. Generating Method Documentation Using Concrete Values from Executions. In *OpenAccess Series in Informatics (OASIcs), Proceedings of the 6th Symposium on Languages, Applications and Technologies (SLATE'17), Vila do Conde, Portugal, 26–27 June 2017*; Dagstuhl Research Online Publication Server: Wadern, Germany, 2017; Volume 56, pp. 3:1–3:13.
10. Sridhara, G.; Hill, E.; Muppaneni, D.; Pollock, L.; Vijay-Shanker, K. Towards Automatically Generating Summary Comments for Java Methods. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, Antwerp, Belgium, 20–24 September 2010; ACM: New York, NY, USA, 2010; pp. 43–52.
11. Sridhara, G.; Pollock, L.; Vijay-Shanker, K. Generating Parameter Comments and Integrating with Method Summaries. In Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, ICPC '11, Kingston, ON, Canada, 20–24 June 2011; IEEE Computer Society: Washington, DC, USA, 2011; pp. 71–80.
12. Buse, R.P.; Weimer, W.R. Automatic Documentation Inference for Exceptions. In Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, Seattle, WA, USA, 20–24 July 2008; ACM: New York, NY, USA, 2008; pp. 273–282.

13. Long, F.; Wang, X.; Cai, Y. API Hyperlinking via Structural Overlap. In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '09, Amsterdam, The Netherlands, 24–28 August 2009; ACM: New York, NY, USA, 2009; pp. 203–212.
14. Buse, R.P.; Weimer, W.R. Synthesizing API usage examples. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 782–792.
15. Montandon, J.a.E.; Borges, H.; Felix, D.; Valente, M.T. Documenting APIs with examples: Lessons learned with the APIMiner platform. In Proceedings of the 2013 20th Working Conference on Reverse Engineering (WCRE), Koblenz, Germany, 14–17 October 2013; pp. 401–408.
16. Hoffman, D.; Strooper, P. Prose + Test Cases = Specifications. In Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS '00, Santa Barbara, CA, USA, 30 July–3 August 2000; IEEE Computer Society: Washington, DC, USA, 2000; pp. 239–250.
17. Sulír, M.; Nosáľ, M.; Porubän, J. Recording concerns in source code using annotations. *Comput. Languages, Syst. Struct.* **2016**, *46*, 44–65.
18. Sulír, M.; Porubän, J. Semi-automatic Concern Annotation Using Differential Code Coverage. In Proceedings of the 2015 IEEE 13th International Scientific Conference on Informatics, Poprad, Slovakia, 18–20 November 2015; pp. 258–262.
19. Dallmeier, V.; Lindig, C.; Wasylkowski, A.; Zeller, A. Mining Object Behavior with ADABU. In Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, WODA '06, Shanghai, China, 23 May 2006; ACM: New York, NY, USA, 2006; pp. 17–24.
20. Panichella, S.; Panichella, A.; Beller, M.; Zaidman, A.; Gall, H.C. The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, Austin, TX, USA, 14–22 May 2016; ACM: New York, NY, USA, 2016; pp. 547–558.
21. Zhang, S.; Zhang, C.; Ernst, M.D. Automated Documentation Inference to Explain Failed Tests. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, Lawrence, KS, USA, 6–10 November 2011; IEEE Computer Society: Washington, DC, USA, 2011; pp. 63–72.
22. Sohan, S.M.; Anslow, C.; Maurer, F. SpyREST: Automated RESTful API Documentation Using an HTTP Proxy Server. In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE '15, Lincoln, NE, USA, 9–13 November 2015; IEEE Computer Society: Washington, DC, USA, 2015; pp. 271–276.
23. Tan, S.H.; Marinov, D.; Tan, L.; Leavens, G.T. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, Montreal, QC, Canada, 17–21 April 2012; IEEE Computer Society: Washington, DC, USA, 2012; pp. 260–269.
24. Sulír, M.; Porubän, J. RuntimeSearch: Ctrl+F for a Running Program. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana Champaign, IL, USA, 30 October–3 November 2017; pp. 388–393.