DScribe: Co-generating Unit Tests and Documentation

Alexa Hernandez
alexahernandez@mail.mcgill.ca
School of Computer Science
McGill University
Montréal, QC, Canada

Mathieu Nassif
mnsassif@cs.mcgill.ca
School of Computer Science
McGill University
Montréal, QC, Canada

Martin P. Robillard
martin@cs.mcgill.ca
School of Computer Science
McGill University
Montréal, QC, Canada

ABSTRACT
Test suites and documentation capture similar information despite serving distinct purposes. Such redundancy introduces the risk that the artifacts inconsistently capture specifications. We present DScribe, an approach that leverages the redundant information in tests and documentation to reduce the cost of creating them and the threat of inconsistencies. DScribe allows developers to define simple templates that jointly capture the structure to test and document a specification. They can then use these templates to generate consistent and checkable tests and documentation. By linking documentation to unit tests, DScribe ensures documentation accuracy as outdated documentation is flagged by failing tests. DScribe’s template-based approach also enforces a uniform style throughout the artifacts. Hence, in addition to reducing developer effort, DScribe improves artifact quality by ensuring consistent content and style. Video: https://www.youtube.com/watch?v=CUKp3-MjMog

CCS CONCEPTS
- Software and its engineering → Documentation; Software testing and debugging; Maintaining software.

KEYWORDS
test generation, documentation generation, maintainability.

ACM Reference Format:

1 INTRODUCTION
Software projects encode information in multiple forms: not only source code but also extensive test suites and documentation. While each artifact serves a different purpose, the information they capture is similar. Source code implements the project’s specifications, while test suites and documentation validate and explain them, respectively. Such redundancy introduces the risk of artifacts inconsistently capturing specifications, a common problem that limits their usefulness [7, 16]. Redundancy also exacerbates the repetitiveness of testing [15] and documentation [8] effort, especially in cases where many functions exhibit similar specifications.

To illustrate the problem, we use the format(Object, StringBuilder) method from Log4j2’s PatternConverter interface which has over 35 implementations and more than 100 unit tests (release 2.14.1). The method transforms and appends information from an Object to a StringBuilder. For each implementation, at least one usage example should be tested and documented. These usage examples constitute a significant amount of redundant information that must be kept consistent. Without traceability links, this is an effort-intensive [14] and error-prone [2] process. As such, it is not surprising that documentation and unit tests are frequently outdated and incomplete [1, 4].

We developed DScribe, an approach that leverages redundant and repetitive information in artifacts to reduce the effort required to create them and the threat of inconsistencies. Rather than producing multiple representations of a specification, DScribe allows developers to define specifications once using simple templates. With a single line of code, developers can use templates to generate tests and documentation in a consistent and streamlined fashion. DScribe is fully supported by a publicly available tool for Java.

DScribe’s template-based approach enforces a uniform style throughout the tests and documentation, reducing the cognitive load required to understand them [11]. By generating documentation from the same source as tests, DScribe also solves the documentation traceability problem and ensures that documentation is accurate since outdated documentation is flagged by failing tests. DScribe does not aim to replace manual testing but to streamline testing and documenting boilerplate specifications, allowing developers to focus on complex, component-specific specifications.

This demonstration presents an overview of DScribe, which is described in detail in our prior work [12]. We also introduce a new Eclipse plug-in that facilitates using DScribe.1 We present two scenarios for using DScribe with universal and project-specific specifications. To support our scenarios, we released six predefined templates on DScribe’s GitHub site.2 These templates reduce the adoption cost of DScribe. Finally, we summarize four studies from our prior work that confirm DScribe’s potential to reduce developer effort and inconsistencies [12].

2 DScribe
We first present an overview of DScribe. Then, we describe an Eclipse plug-in that enhances the user experience of DScribe.

1https://www.cs.mcgill.ca/~martin/DScribeUpdateSite
2https://github.com/prmr/DScribe
To use DScribe, developers create templates that jointly capture how to test and document a specification. DScribe templates associate a code skeleton with a documentation fragment. As an example, Figure 1 depicts a template to test and document Log4j2’s format method. The code skeleton, spanning lines 3 to 10, defines a recurrent structure for tests. It includes the boilerplate code for the arrange, act, and assert sections [5]. Information that depends on a particular test case, such as the expected value, is abstracted using placeholders, e.g., $expected$. The documentation fragment on line 1 describes the behavior under test using free-form text in a header comment. In this case, it illustrates a usage example by referencing the code skeleton’s placeholders, e.g., $field$. In general, documentation fragments can capture any kind of documentation, including preconditions, exceptional behavior and behavior summaries, as they are defined using free-form text. Line 2 assigns “Format” as the template’s name.

To test and document a particular behavior for a method, developers invoke a template by providing the use-case-specific information required to capture the behavior completely. Developers invoke templates by adding a Java annotation to the focal method. Each template is associated with an annotation that has the same name and one member per placeholder. For example, to test and document an implementation of format that appends a LogEvent’s thread ID to the StringBuilder, one would annotate it with @Format. The above template invocation defines values for each placeholder in the Format annotation, except $class$. The $class$ placeholder, along with the $method$ placeholder, are predefined in DScribe. Their values are derived directly from the focal method and its declaring class, freeing users from specifying them. DScribe swaps placeholders for invocation values with minimal transformation, ensuring generation is as transparent as possible for developers. Hence, if the value provided is a String literal, as is the case for $expected$, it must be surrounded by double quotes.

The above template invocation defines values for each placeholder in the format_When$fieldName$Is$value$_Append$expected$(){} template, except $class$. The $class$ placeholder, along with the $method$ placeholder, are predefined in DScribe. Their values are derived directly from the focal method and its declaring class, freeing users from specifying them. DScribe swaps placeholders for invocation values with minimal transformation, ensuring generation is as transparent as possible for developers. Hence, if the value provided is a String literal, as is the case for $expected$, it must be surrounded by double quotes.

Figure 1: Template for the format method. The documentation fragment is on line 1 and the code skeleton spans lines 3 to 10. Surrounding dollar signs ($) denote placeholders.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/* e.g., when $fieldName$ is $value$, $expected$ is appended to the StringBuilder */</td>
</tr>
<tr>
<td>2</td>
<td>@Template(&quot;Format&quot;)</td>
</tr>
<tr>
<td>3</td>
<td>@Test</td>
</tr>
<tr>
<td>4</td>
<td>public void format_When$fieldName$Is$value$_Append$expected$(){}</td>
</tr>
<tr>
<td>5</td>
<td>LogEvent event = Log4jLogEvent.newBuilder().set$fieldName$(value).build();</td>
</tr>
<tr>
<td>6</td>
<td>StringBuilder sb = new StringBuilder();</td>
</tr>
<tr>
<td>7</td>
<td>$className$ converter = $className$.newInstance($params$);</td>
</tr>
<tr>
<td>8</td>
<td>converter.format(event, sb);</td>
</tr>
<tr>
<td>9</td>
<td>assertEquals($expected$, sb.toString());</td>
</tr>
<tr>
<td>10</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 2: DScribe Eclipse plug-in. On the left is the “Run DScribe” command (top) and the generated documentation fragment (bottom). On the right is the dialog that appears after running DScribe and the generated unit test.

2.1 The Approach

The above template invocation defines values for each placeholder in the format annotation, except $class$. The $class$ placeholder, along with the $method$ placeholder, are predefined in DScribe. Their values are derived directly from the focal method and its declaring class, freeing users from specifying them. DScribe swaps placeholders for invocation values with minimal transformation, ensuring generation is as transparent as possible for developers. Hence, if the value provided is a String literal, as is the case for $expected$, it must be surrounded by double quotes.

The above template invocation defines values for each placeholder in the format_When$fieldName$Is$value$_Append$expected$(){} template, except $class$. The $class$ placeholder, along with the $method$ placeholder, are predefined in DScribe. Their values are derived directly from the focal method and its declaring class, freeing users from specifying them. DScribe swaps placeholders for invocation values with minimal transformation, ensuring generation is as transparent as possible for developers. Hence, if the value provided is a String literal, as is the case for $expected$, it must be surrounded by double quotes.

3 USAGE SCENARIO

In this section, we present two usage scenarios of DScribe—namely, generating assets for universal and project-specific specifications.

3.1 Universal Specifications

DScribe facilitates testing and documenting universal specifications—specifications that appear in most software projects, regardless of their domain or features. Examples include specifications about exceptions, equals contracts, and clone contracts. In addition to recurring across projects, universal specifications can also exist in abundance within a project. For example, our previous work identified over 800 specifications about exceptions in just eleven classes of Apache Commons IO [12]. Creating and maintaining a unit test and documentation fragment for each is laborious and error-prone. Without tool support, developers often fail to evolve both artifacts consistently [2]. As evidence, 85% of the identified specifications about exceptions were not tested or correctly documented [12].

Developers can leverage DScribe to reduce the effort required to generate assets for universal specifications. Instead of creating, or cloning, 1,600 repetitive assets (800 tests and 800 documentation...
3.2 Project-Specific Specifications

As project-specific patterns emerge in manually-written tests, developers can design ad hoc templates to generate assets for them. Testing the format method introduces one such project-specific pattern. Even the Log4j2 developers noticed the repetition and applied two techniques to mitigate it. However, both have drawbacks that D Scriber avoids. Nevertheless, their use confirms the demand for tools to reduce the repetitiveness of testing activities.

3.2.1 Alternative Approaches. To reduce the repetition involved in testing the format method, developers used copy-pasting and helper functions. While both approaches expedite testing recurrent specifications, they do so at the expense of test quality.

The use of copy-pasting is apparent from the copy-paste-misadapt error [10] marked with a dashed underline in Figure 4. One may infer that the developer copied the test from the NanoTimePatternConverterTest class to the ThreadIdPatternConverterTest class and correctly adapted all code elements except the method name. This error is not uncommon as copy-pasting often introduces hard-to-spot semantic bugs [6] which degrade the code’s quality.

The second approach resembles D Scriber in that it uses helper functions, instead of templates, to abstract the common structure of similar tests. The helper functions also take as input test-case-specific information. An example helper function is testReplacement from the EqualsReplacementConverterTest class. Instead of duplicating code, various unit tests in EqualsReplacementConverterTest consist of a single call to testReplacement. By separating a test’s structure from its definition, this approach obscures the purpose of a test.

Unlike the above approaches, D Scriber reduces repetition while maintaining asset quality. D Scriber generates high-quality unit tests that are self-evident. It can even improve readability by ensuring that tests and documentation adhere to a consistent style. These benefits come in addition to D Scriber’s main value proposition: to generate consistent and checkable documentation at no extra cost.

3.2.2 Designing Templates. To create a D Scriber template, developers start by analyzing patterns in existing tests. First, they identify the code elements that are common across the tests. These elements will comprise the template’s code skeleton. For example, both format tests in Figure 4, along with several others, follow a similar recipe. They start by instantiating a LogEvent, StringBuilder, and PatternConverter instance. Then, they call format, passing the LogEvent and StringBuilder instances as parameters. Finally, they use JUnit’s assertEquals to validate the results. This sequence of code elements forms the code skeleton in the Format template (Figure 1).

Next, developers identify the code elements that vary between tests. These elements will become the code skeleton’s placeholders. For example, the LogEvent field that we set depends on the test (ThreadId vs. NanoTime). Thus, we replace it with the $field$ placeholder in the Format template. Finally, developers add a text fragment documenting the behavior that the code skeleton validates. The fragment in the Format template uses the code skeleton’s placeholders to capture a usage example of format.

In retrospect, the Format template could have captured 52% (59 of 114) of the format specifications. This would have significantly streamlined testing while adding helpful usage examples which do not currently exist. We could make the template more applicable by adding extra placeholders to generalize its structure—however, the more placeholders, the more work required to invoke templates. Also, general templates tend to require calls to helper functions as placeholder values, which may degrade readability. These are just a few of the design decisions developers must make when creating templates. By giving developers complete creative freedom, D Scriber empowers them to meet their needs effectively.

4 EVIDENCE OF EFFECTIVENESS

In previous work [12], we conducted four studies to assess D Scriber’s ability to reduce testing and documentation effort and prevent inconsistencies between artifacts. This section summarizes those studies, focusing on findings relevant to D Scriber’s applicability.

Usefulness Study: We performed a case study to evaluate the degree of information inconsistency in a mature project, Apache Commons IO, and assess D Scriber’s potential to avoid them. We used a unit of specification as our unit of analysis and focused solely on thrown exceptions to make the study tractable. We considered an exception specification unit (ESU) to be inconsistent if there was any divergence in its associated artifacts (code, documentation, or tests), including cases where an artifact omitted the ESU.

We analyzed all 293 public, non-deprecated methods in the root package. For each, we identified all ESUs present in the source code,
Figure 4: Two Log4j2 test classes with differing code elements underlined. The dashed underlined marks a copy-paste error.

documentation, and/or test suite. For each ESU, we noted whether it was consistent and invoked a template to capture it.

The study showed that information inconsistencies are prevalent in Commons IO, with 85% of the identified ESUs being inconsistent. Dscribe templates could capture 97% of the identified inconsistencies using only five templates. Hence, templates can be highly reusable, leading to a low cost of creation. Overall, the study substantiated Dscribe’s potential to avoid future inconsistencies.

**Comparison Study:** We recruited four annotators to assess Dscribe’s ability to yield high-quality tests. The annotators compared the quality of the unit tests produced by Dscribe in the usefulness study against three baselines: the original Apache Commons IO test suite and tests produced by two state-of-the-art test generation techniques—EvoSuite [3] and Randoop [13]. We measured the quality of a test using two well-established properties—readable and focused [9]. Each annotator evaluated 20 tests per test suite for a total of 320 tests. The study revealed that Dscribe generates tests that are more readable and focused than the baselines.

**Validation Study:** Using a multi-case study, we assessed the generalizability of the findings from the usefulness study beyond exception handling. Specifically, we assessed the extent to which unit tests from three Apache Commons projects (Math, Lang, and Configuration) capture information worth documenting. We randomly sampled 370 tests uniformly from the population of 9397 tests. For each test, we identified the focal method and noted whether the test captured any units of specification about it. For each unit of specification, we noted whether the documentation captured it.

Overall, 42% of the sampled tests captured information worth documenting, but about half of that information was undocumented. Dscribe prevents this by generating documentation from the same source as unit tests (i.e., template invocations). The study also unveiled a novel use case for Dscribe, which we use in this demonstration: using templates to document usage examples.

**Limitations Study:** We performed a qualitative multi-case study to elicit the limitations of a template-based approach for test generation. We analyzed five open source projects that differ in their development style, target audience, and application domain: FreeMind, Eclipse Platform UI, Weka, Apache Tomcat, and Hibernate ORM. For each test, we identified technical factors that impact the generation of similar tests from templates.

The study revealed eight such factors, one of which is Different Units Under Test: the current version of Dscribe assumes that each test targets a single method, making it impossible to generate tests that focus on a class or field. Another factor is Constrained Resources: tests that rely on constrained resources often include unique operations to handle the resource, impeding the use of templates that are not specifically designed for the resource. One mitigation strategy is to use setup and teardown methods. Our prior work [12] discusses these factors in detail to help developers decide whether Dscribe meets their needs.

5 CONCLUSION
We introduced Dscribe, a novel template-based approach to co-generate unit tests and documentation, along with an Eclipse plug-in that facilitates Dscribe’s use. Developers can use Dscribe to streamline testing and documenting recurrent specifications, whether universal or project-specific. In addition to reducing developer effort, the template-based generation enhances the quality of a project’s artifacts by ensuring consistency in content and style.

ACKNOWLEDGMENTS
We are grateful to the external annotators for helping with the comparison study. This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds de recherche du Québec – Nature et technologies (FRQNT).

REFERENCES


