DScribe: Co-generating Unit Tests and Documentation

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

Mathieu Nassif
mnassif@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 By linking documentation to unit tests, DScribe ensures documentation accuracy as outdated documentation is flagged by failing tests.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:
Alexa Hernandez, Mathieu Nassif, and Martin P. Robillard. 2022. DScribe: Co-generating Unit Tests and Documentation. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516856

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 out-of-date 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.

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516856

[1] https://www.cs.mcgill.ca/~martin/DScribeUpdateSite

[2] https://github.com/prmr/DScribe

```
1   /** e.g., when $field$ is $value$, $expected$ is appended to the StringBuilder */
2   @Template("Format")
3   @Test
4   public void format_When$field$Is$value$_Append$expected$() {
5       LogEvent event = Log4jLogEvent.newBuilder().set$field$($value$).build();
6       StringBuilder sb = new StringBuilder();
7       $class$ converter = $class$.newInstance($params$);
8       converter.format(event, sb);
9       assertEquals($expected$, sb.toString());
10  }
```

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.

2.1 The Approach

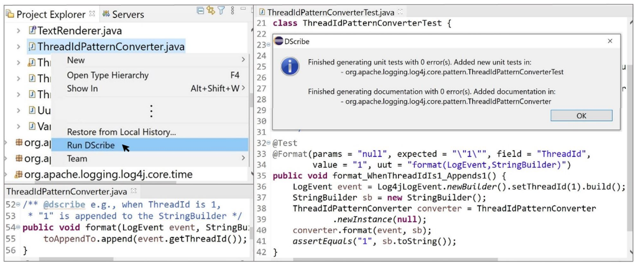
To use DScribe, developers create *templates* that jointly capture how to test and document a specification. DScribe templates associate a *code skeleton* with a *documentation fragment*. As an example, Figure 1 depicts a template to test and document Log4j2's format method. The code skeleton, spanning 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:

```
@Format(field="ThreadId", value="1", params="null", expected="\"1\"")
```

The above template invocation defines values for each placeholder in the Format 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.

DScribe integrates the generated assets into existing artifacts. To avoid corrupting manually-written documentation, DScribe marks the generated documentation fragments with the custom Javadoc tag @dscribe. This tag differentiates generated and manually-written fragments, making it easy to replace only generated documentation. Similarly, to avoid disrupting manually-written tests, DScribe automatically moves the annotation used to invoke the template from the focal method to the generated unit test. It also improves user experience by helping users discern the template invocation responsible for a test.


Project Explorer
Servers
TextRenderer.java
ThreadIdPatternConverter.java
Thı
Thı
Thı
Uu
Vaı
org.aı
org.aı
org.apache.logging.log4j.core.time
New
Open Type Hierarchy F4
Show In Alt+Shift+W
Restore from Local History...
Run DScribe
Team
ThreadIdPatternConverter.java
ThreadIdPatternConverterTest.java
DScribe
OK

```
/** @dscribe e.g., when ThreadId is 1,
 * "1" is appended to the StringBuilder */
public void format(LogEvent event, StringBu
    toAppendTo.append(event.getThreadId());
}
```

Finished generating unit tests with 0 error(s). Added new unit tests in:
- org.apache.logging.log4j.core.pattern.ThreadIdPatternConverterTest

Finished generating documentation with 0 error(s). Added documentation in:
- org.apache.logging.log4j.core.pattern.ThreadIdPatternConverter

```
class ThreadIdPatternConverterTest {

@Test
@Format(params = "null", expected = "\"1\"", field = "ThreadId",
        value = "1", uut = "format(LogEvent,StringBuilder)")
public void format_WhenThreadIdIs1_Appends1() {
    LogEvent event = Log4jLogEvent.newBuilder().setThreadId(1).build();
    StringBuilder sb = new StringBuilder();
    ThreadIdPatternConverter converter = ThreadIdPatternConverter
            .newInstance(null);
    converter.format(event, sb);
    assertEquals("1", sb.toString());
}
```

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.

2.2 The DScribe Eclipse Plug-in

There are different ways to run DScribe—as a Java jar file, as an Eclipse project, or using the more user-friendly DScribe Eclipse plug-in. After invoking templates, developers can use the plug-in to generate assets for a production class. To do so, one would right-click on the class in Eclipse's Project Explorer view and then select the "Run DScribe" command, as shown in Figure 2. To generate assets for multiple classes at once, one would select all of the classes in the Project Explorer view before performing the command.

When asset generation is complete, a message dialog appears to inform users (see Figure 2). The dialog lists all classes for which DScribe has added or modified assets, as well as any errors that occurred. The asset generation process is not CPU-intensive; for example, it generates assets for an invocation of the Format template without a perceivable delay on a Windows 10 laptop with Intel i7-10510U CPU and 16 GB RAM.

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

```
/** Throws $exType$ when $state$ */
@Template("AssertThrows")
@Test
public void $method$_When$state$_Throw$exType$() {
   assertThrows($exType$, () -> $factory$.$method$($params$));
}

/** Performs a shallow copy of the object. */
@Template("ShallowClone")
@Test
public void clone_ReturnShallowCopy() {
   $class$ initial = $factory$;
   $class$ cloned = initial.clone();
   assertNotSame(initial, cloned);
   assertEquals(initial, cloned);
}
```

Figure 3: Two of the six predefined templates.

fragments), developers can write short annotations requiring only use-case–specific information. Moreover, by linking documentation to tests, DScribe ensures that assets are consistent, as failing tests instantly flag outdated documentation. Our finding that 97% of the identified inconsistencies in Commons IO were preventable by DScribe confirms its ability to enhance asset consistency [12].

Given the prevalence of universal specifications, their templates can be created as a part of a testing strategy and shared across projects. To minimize duplicated effort and DScribe's learning curve, we created six templates for well-known universal specifications. Figure 3 displays two such templates, one to capture specifications about exceptions, the other to test and document a shallow implementation of clone. These templates reduce the adoption cost of DScribe and can easily be adapted to comply with a team's style conventions. Once developers master the predefined templates, they can design new ones tailored to project-specific concerns.

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 DScribe 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 DScribe 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, DScribe reduces repetition while maintaining asset quality. DScribe 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 DScribe's main value proposition: to generate consistent and checkable documentation at no extra cost.

3.2.2 Designing Templates. To create a DScribe 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, DScribe empowers them to meet their needs effectively.

4 EVIDENCE OF EFFECTIVENESS

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

```
public class NanoTimePatternConverterTest {
  @Test
  public void testConverterAppendsLogEventNanoTimeToStringBuilder() {
    final LogEvent event = Log4jLogEvent.newBuilder().setNanoTime(1234567).build();
    final StringBuilder sb = new StringBuilder();
    final NanoTimePatternConverter converter = NanoTimePatternConverter
        .newInstance(null);
    converter.format(event, sb);
    assertEquals("1234567", sb.toString());
  }
}
```

```
public class ThreadIdPatternConverterTest {
  @Test
  public void testConverterAppendsLogEventNanoTimeToStringBuilder() {
    final LogEvent event = Log4jLogEvent.newBuilder().setThreadId(1).build();
    final StringBuilder sb = new StringBuilder();
    final ThreadIdPatternConverter converter = ThreadIdPatternConverter
        .newInstance(null);
    converter.format(event, sb);
    assertEquals("1", sb.toString());
  }
}
```

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

[1] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. In *In Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*. 1199–1210. https://doi.org/10.1109/ICSE.2019.00122

[2] Barthélémy Dagenais and Martin P. Robillard. 2014. Using Traceability Links to Recommend Adaptive Changes for Documentation Evolution. *IEEE Transactions on Software Engineering* 40, 11 (2014), 1126–1146. https://doi.org/10.1109/TSE.2014.2347969

[3] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[4] Daniel Gaston and James Clause. 2020. A Method for Finding Missing Unit Tests. In *In Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution*. 92–103. https://doi.org/10.1109/ICSME46990.2020.00019

[5] Jeff Grigg. 2012. Arrange Act Assert. http://wiki.c2.com/?ArrangeActAssert

[6] Patricia Jablonski and Daqing Hou. 2007. CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE. In *In Proceedings of the 2007 OOPSLAWorkshop on Eclipse Technology EXchange*. 16–20. https://doi.org/10.1145/1328279.1328283

[7] Mira Kajko-Mattsson. 2005. A Survey of Documentation Practice within Corrective Maintenance. *Empirical Software Engineering* 10, 1 (2005), 31–55. https://doi.org/10.1023/B:LIDA.0000048322.42751.ca

[8] D.V. Luciv, D. V. Koznov, G. A. Chernishev, A. N. Terekhov, K. Yu. Romanovsky, and D. A. Grigoriev. 2018. Detecting Near Duplicates in Software Documentation.

Programming and Computer Software 44 (2018), 335–343. https://doi.org/doi.org/10.1134/S0361768818050079

[9] Robert C. Martin. 2009. *Clean Code – a Handbook of Agile Software Craftsmanship*. Prentice Hall.

[10] Petru-Florin Mihancea and Roger Scott. 2019. CodeSonar (R) Extension for Copy-Paste-(Mis) Adapt Error Detection. In *In Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution*. 386–389. https://doi.org/10.1109/ICSME.2019.00065

[11] Isaac Moreira Medeiros Gomes, Daniel Coutinho, and Marcelo Schots. 2019. No Accounting for Taste: Supporting Developers' Individual Choices of Coding Styles. In *In 19th International Working Conference on Source Code Analysis and Manipulation*. 86–91. https://doi.org/10.1109/SCAM.2019.00018

[12] Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P. Robillard. 2021. Generating Unit Tests for Documentation. *Transactions on Software Engineering* (2021).

[13] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *In Proceedings of the 29th IEEE/ACM International Conference on Software Engineering*. 75–84. https://doi.org/10.1109/ICSE.2007.37

[14] P. Runeson. 2006. A survey of unit testing practices. *IEEE Software* 23, 4 (2006), 22–29. https://doi.org/10.1109/MS.2006.91

[15] Brent van Bladel and Serge Demeyer. 2021. A comparative study of test code clones and production code clones. *Journal of Systems and Software* 176 (2021), 110940. https://doi.org/10.1016/j.jss.2021.110940

[16] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through Repository Mining. *Empirical Software Engineering* 16, 3 (2011), 325–364. https://doi.org/10.1007/s10664-010-9143-7