Generating Unit Tests for Documentation

Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P. Robillard

Abstract—Software projects capture redundant information in various kinds of artifacts, as specifications from the source code are also tested and documented. Such redundancy provides an opportunity to reduce development effort by supporting the joint generation of different types of artifact. We introduce a tool-supported technique, called DScribe, that allows developers to combine unit tests and documentation templates, and to invoke those templates to generate documentation and unit tests. DScribe supports the detection and replacement of outdated documentation, and the use of templates can encourage extensive test suites with a consistent style. Our evaluation of 835 specifications revealed that 85% were not tested or correctly documented, and DScribe could be used to automatically generate 97% of the tests and documentation. An additional study revealed that tests generated by DScribe are more focused and readable than those written by human testers or generated by state-of-the-art automated techniques.

Index Terms—Code documentation, Testing, Testing tools, Test generation, Maintainability, Specification management

1 Introduction

Mature software frameworks and libraries are usually complemented by extensive test suites and reference documentation. For example, Apache Commons Math 3.6 is supported by over 4400 tests and 215 000 words of method reference documentation. Although tests and documentation serve different purposes, they express similar information. For example, many functions throw an exception when supplied with an invalid argument. Ideally, such behavior is also documented and tested. Future maintenance tasks should ensure that tests and documentation remain consistent and accurate. Such maintenance tasks can require considerable effort [1], [2], and are often neglected [3].

Current practices for testing and documenting reusable software assets require the capture of *redundant* information. Redundancy introduces the risk of specifications being *inconsistently* captured between documentation and the exercise of the corresponding behavior in a test. This risk is significant as developers sometimes fail to update documentation [4] or only do so long after the corresponding code changes [5]. In situations where many functions exhibit similar constraints (e.g., a parameter that should be positive), the redundancy between tests and documentation exacerbates the *repetitiveness* of the testing and documentation effort. As a result, studies have found recurrent issues in documentation [6] and unit tests [7], [8], such as a lack of completeness and up-to-dateness.

The goal of our research is to leverage the redundancy and repetitiveness of information in software artifacts to reduce the effort required to create them and the threat of inconsistencies. We introduce a technique, called DScribe, that leverages a new synergy between template-based unit tests and documentation to efficiently create consistent and checkable documentation and unit tests. The technique is fully supported by a publicly available tool for Java.

DScribe provides developers with a method to create joint templates for unit tests and documentation that capture the structure to test and document a recurring concern. Testers can then invoke the templates to generate test code and documentation. With this approach, developers only need to write boilerplate code (i.e., templates) once, and can effectively update a large numbers of artifacts, e.g., when changing code style conventions. The template-based generation also ensures that tests and documentation follow a consistent style, and their association allows to detect outdated documentation when the corresponding tests fail.

The design of DScribe incorporates, among others, a new template definition language, a serialization format for templates and invocations, and algorithms for integrating generated unit tests and documentation with existing artifacts, without disrupting them.

As a research project focused on engineering design, we evaluated DScribe's ability to reduce testing and documentation effort for recurrent specifications and to prevent inconsistencies between artifacts. We analyzed qualitative and quantitative data from different perspectives to gain a deeper understanding of this ability: we surveyed recurrent specifications related to exceptions in production code, test code, and documentation; we gathered feedback from developers about the quality of the artifacts generated with DScribe; we assessed the degree to which tests capture information worth documenting; and we elicited factors acting as obstacles to our approach. Our studies revealed the potential benefits that DScribe could have on development practices. For example, 85% of the specifications about exceptions in the Apache Commons IO library are either untested, undocumented, or both, with 97% of these problems being directly preventable by DScribe. Moreover, our comparison study substantiated DScribe's ability to produce tests with a quality matching or surpassing existing baselines.

The main contribution of this paper is a novel technique to support the co-generation of unit tests and documentation. We also contribute four studies that provide insights on the potential of generating unit tests for documentation.

This article is organized as follows. In Section 2, we provide an overview of DScribe, followed by a description of its two key aspects: templates (Section 3) and generative technology (Section 4). Section 5 presents the empirical assessment of

• *The authors are with the School of Computer Science, McGill University, Montréal, Canada.*
E-mail: {mnassif, martin}@cs.mcgill.ca,
E-mail: {alexa.hernandez, ashvitha.sridharan}@mail.mcgill.ca

Manuscript received ...; revised ...

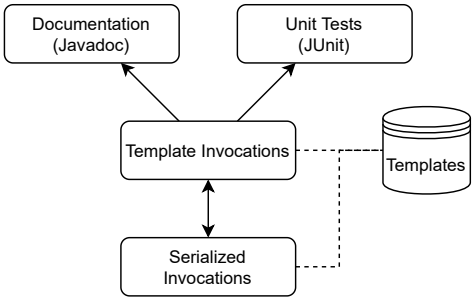

Fig. 1. Information representations in DScribe. Rounded rectangles show representations of information, and arrows indicate supported transformations. Dashed lines indicates a dependency to a set of templates.

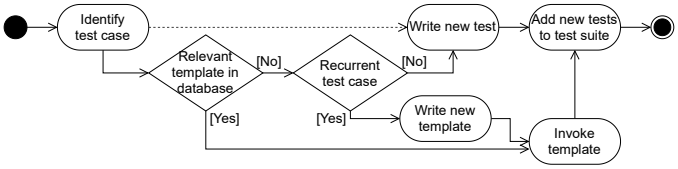

Fig. 2. Integration of DScribe in a development process. The activity diagram compares the creation of new unit tests in a typical scenario (dashed arrow) to a scenario using DScribe (solid arrows).

the work, followed by the details of a usefulness study (Section 6), a comparative quality assessment study (Section 7), a validation study (Section 8), and a qualitative study of the limitations of our approach (Section 9). Section 10 presents the related work and Section 11 concludes the paper.

Our contributions are complemented by an on-line appendix, which contains the source code of DScribe and details of the evidence collected as part of the studies, available at https://github.com/prmr/DScribe-Research (version 1.0).

2 DScribe Overview

DScribe is an approach to transform different representations of equivalent information about software elements. In the current implementation of DScribe, each piece of information must relate to a public Java method, called the *focal method*. For example, a focal method may throw a NullPointerException when its argument is null. DScribe allows developers to encode this specification once using a simple format, then generate different artifacts that capture the same information.

Figure 1 summarizes the information representations supported by DScribe. At the center are *template invocations*, which consist of the minimal amount of information necessary to describe a specification, including the focal method and details of the case to test and document.

Template invocations rely on *templates*. A template captures the structure of the tests and documentation for a specification. It includes placeholders to allow its reuse in different contexts. Table 1 presents three examples of templates, each composed of a documentation (doc.) and a test component.

Developers *invoke* the templates to express a particular specification about a method by writing JSON-formatted *invocation files*. DScribe uses this information together with templates to generate documentation and unit tests that capture the same information as the invocation files.

DScribe integrates into the development process as shown in Figure 2. When testing a method, a tester normally identifies a test case, then writes a new test. With DScribe, the tester instead selects a relevant template and only provides values for its placeholders. New templates can be created at any point, either opportunistically or as part of a testing plan. The generated assets are independent of the generation framework, so the use of DScribe does not require a project-wide adoption or other changes in the development environment.

TABLE 1
Three examples of template. Each template consists of a unit test and documentation fragment, with placeholders marked by dollar signs ($).

ID	Template
1 (doc.)	$method$ will throw an exception of type $excep$ if the argument is arg.

1 (test)	`@Test`
	`public void test$method$_invalidArgument() {`
	`  $class$ obj = $factory$();`
	`  assertThrows($excep$.class, () -> obj.$method$($arg$));`
	`}`

2 (doc.)	If no argument is provided, the default value for the arguments of $method$ is $defVal$.

2 (test)	`@Test`
	`public void test$method$_defaultArgument() {`
	`  assertEquals( $method$() , $method$($defVal$) );`
	`}`

3 (doc.)	If the $class$ object is closed, $method$ returns val.

3 (test)	`@Test`
	`public void test$method$_afterClose() {`
	`  $class$ instance = $factory$();`
	`  instance.close();`
	`  assertEquals($val$, instance.$method$());`
	`}`

The realization of DScribe required solving a number of design and implementation challenges. A key design principle of DScribe was to make the generation process as transparent as possible for developers. Past research has proposed various inference techniques to generate information, based on various representations [9]. A limitation of these techniques is that they require developers to validate the outcome of the inference process. In contrast, DScribe makes effective use of unambiguous information specified by developers.

3 Templates and Invocations

We consider the following test:

```
@Test
public void testGet_WhenEmpty_ThrowsException() {
  Optional<?> instance = Optional.empty();
  assertThrows(NoSuchElementException.class,
      () -> instance.get());
}
```

The test checks that the method get throws an exception when called on an Optional instance that does not contain a reference. Some information, such as the state of the Optional instance, is specific to the current test case, but the code also captures information that is relevant to other contexts, such as a typical usage of the assertThrows method. This same structure can thus apply to a variety of test cases.

The design of DScribe hinges on the separation of this information, related to the general concern being tested (captured by the *template*), or related to details of the test case (provided by each *invocation*). Templates should be reusable across different contexts, while retaining a self-evident purpose. In contrast, invocations should require the minimal information to complete a template for a specific context.

3.1 Templates

A template is the association of a code skeleton for a specific kind of unit test, together with a documentation fragment that describes the tested specification. This association is the key to generating consistent and verifiable documentation.

The code skeleton is created as an abstract syntax tree (AST) with a set of nodes marked as *placeholders*, identified as tokens surrounded by dollar signs ('$', a legal but discouraged character for manually-written Java identifiers [10, §3.8]). Thus, code skeletons are written as syntactically valid code.

For example, the test shown at the beginning of this section can be adapted into a code skeleton by replacing test-case-specific elements with placeholders. In this example, this would require modifying (a) the Get and (b) Empty tokens in the test name, (c) the type and (d) instantiation of the instance variable, (e) the type of exception thrown, and (f) what method is called on the instance variable. Three of these values (a, c, and f) are derived directly from the focal method and its declaring class, for which DScribe provides two predefined placeholders ($method$ and $class$). Two other values (b and d) capture the state of the tested object that triggers the specification, allowing to use a single placeholder for both. Thus, the following code skeleton, extracted from the original unit test, only require two custom values and the focal method:

```
@Test
public void test$method$_When$state$_ThrowsException() {
  $class$ instance = $class$.$state$();
  assertThrows($exception$.class, () -> instance.$method$());
}
```

To help testers use templates correctly, we defined a type system that indicates how a placeholder is used within the code skeleton. For example, the type of the $exception$ placeholder specifies that its value should refer to an exception class name. Other types specify that a placeholder replaces either a Java type, a method, a field, a Java expression, or a list of Java expressions (e.g., arguments of a method call). DScribe includes straightforward validation rules for each type: values must be syntactically well-formed according to their type, and class names must also resolve to an existing class. As predefined placeholders receive their value from properties of the fully resolved focal method, they do not need a type.

To complete the transformation of a unit test into a DScribe template, template creators associate the new code skeleton with a text fragment that describes what the code checks. This documentation fragment can be any free form text and uses the same placeholders as the code skeleton. As for code skeletons, template creators can reuse existing documentation, replacing specific information with placeholders, to create the text fragment. In our running example, a suitable fragment could be *"$method$ will throw $exception$ if the implicit argument is $state$."* Creating this documentation fragment requires an effort no different from documenting a method, but the result can be reused to document all methods tested by the same template (see Section 4.2).

3.2 Invocations

Testers use template invocations to apply a template to a focal method. In addition to a reference to the template being invoked and the focal method it applies to, each invocation also provides specific values for the (non-predefined) placeholders of the template. Thus, the invocation to generate the test from our running example would specify the focal method Optional-.get(), refer to the template defined in the previous section, and provide the values empty and NoSuchElementException for the $state$ and $exception$ placeholders, respectively.

Following the principle that the generation of tests and documentation should be transparent for testers, DScribe replaces placeholders with the invocation values with as little transformation as possible, only in straightforward cases. For example, DScribe adds the new keyword when a placeholder for a method name is a constructor. Thus, the values supplied to the template invocation are not expressions to be evaluated, but to be substituted verbatim for the placeholders.

Testers invoke templates using invocation files. We made the arbitrary decision to format those files using JSON, which allows testers to directly read, write, and edit invocation files with any text editor. The many JSON libraries also support the implementation of a straightforward serialization and deserialization of invocations.

4 Asset Generation

DScribe generates two kinds of software assets: individual unit tests and documentation fragments inserted in the header comment (Javadoc) of the focal method. The general generation procedure consists of replacing placeholders with invocation values. However, integrating the assets within the system requires solving practical challenges to keep them up-to-date while avoiding corrupting manually-written assets.

4.1 Unit Test Integration

DScribe places the generated tests in new Java files, separated from the rest of the test suite, so that they can be deleted and generated again to update old tests. This strategy allows testers to leverage test scaffolding code by referring to helper methods to provide placeholder values. The scaffolding code only needs to be located in distinct utility files, to avoid being deleted when tests are updated.

Another concern for the generated tests is that they compile and run correctly. The transparent generation process helps testers generate tests that run as they intend, and the placeholder type system helps prevent simple compilation errors. However, DScribe cannot guarantee that generated tests are correct or even compilable. A generated test could, for example, contain unresolved references. This is the trade-off of the flexibility of DScribe: although template creators can design templates for virtually any testing environment, it also allows them to design bad templates. This limitation is consistent with the objective of DScribe, which is to help developers efficiently express and capture knowledge about a system while testing it, rather than dictate a testing style.

4.2 Documentation Integration

Template invocations already capture valuable information about the system under test. They summarize how a specification, defined by the template, applies to different focal methods, in a machine-readable format. This information can be leveraged to generate or improve documentation. As a proof of concept, we implemented one documentation generation

approach that inserts natural language fragments directly into the header comment (Javadoc) of the focal methods.

Contrary to generated tests, documentation cannot be generated in a different file, otherwise developers are likely to ignore it. However, integrating new generated fragments in existing documentation introduces the threat of corrupting existing documentation. DScribe must also update old fragments when invocations or templates are edited, again without affecting manually-written documentation.

To solve both issues, DScribe marks the generated fragments with the custom Javadoc tag `@dscribe`. This tag unambiguously identifies fragments to replace when updating documentation. It also indicates to developers which statements are automatically generated and backed by unit tests.

Each fragment is prefixed by its own tag to avoid a large unreadable paragraph. However, generating many similar fragments for a single method can still clutter the documentation. For example, Java's `Math.log(double a)` method returns the value `NaN` if the argument is negative or `NaN` itself. Testing this specification requires two template invocations, which would in turn generate two sentences that differ only by one word: *"If the argument is [negative/NaN], 'log' returns NaN."*

To avoid generating such repetitive fragments, we designed an optional format for the documentation fragment of a DScribe template. The format consists of two components, a *condition* followed by a *consequence*, both consisting of three internal parts: a subject, a relation, and an object (similar to Resource Description Framework (RDF) triples [11]). For example, one documentation fragment for `Math.log` would be *the argument/is/negative/'log'/returns/NaN*. As this example shows, many specifications naturally follow this six-part structure. This format allows DScribe to merge fragments that differ only by some of their parts. In our running example, only the object of the condition differs in both fragments, so DScribe can add a single sentence to the documentation: *"If the argument is negative or NaN, the method returns NaN."*

5 Overview of the Empirical Assessment

We conducted a multi-pronged empirical investigation of key aspects relating to the co-generation of unit tests and documentation, using our implementation of DScribe. The investigation sought to answer two research questions:

- RQ1 In practice, what is the nature of inconsistencies between source code, unit tests, and documentation?
- RQ2 To what extent can developers leverage templates to automatically test and document focal methods?

We followed a four-stage process, with each stage constituting a cohesive study of its own (see Table 2). An in-depth *usefulness study* focused on the benefits of applying DScribe in a narrow context, i.e., testing thrown exceptions (Section 6). We compared the quality of the tests generated by DScribe to that of tests generated by humans and two automated techniques in a *comparison study* with external annotators (Section 7). We performed a multi-case *validation study* to assess the generalizability of the findings of the usefulness study beyond exception handling (Section 8). Finally, to better understand the limitations of our approach in diverse scenarios, we conducted an empirical *limitations study* (Section 9). For all four studies, the complete and detailed results are publicly available in our on-line appendix.

6 Usefulness Study

We investigated the usefulness of DScribe to prevent inconsistencies. The objective of the investigation was twofold. First, it aimed at assessing the amount of repetitive and redundant information between unit tests and documentation. Second, it aimed at assessing the potential of DScribe to prevent inconsistencies by automating the generation of assets.

6.1 Usefulness Study Design

To study information inconsistency and redundancy, it was necessary to define what constitutes a cohesive unit of information about a method. The information relevant to methods typically includes units of specification regarding, among others, exceptions, parameter types, edge cases, and return types. We chose as our unit of analysis such a *unit of specification*.

In general, a significant manual effort is required to isolate and make sense of even just a few units of specifications in unfamiliar code. To make this case study tractable, we narrowed the scope to a well-defined type of specification: units of specification about exceptions, which are relevant in almost all systems, yet often improperly implemented [12]. For a single method, its source code, associated unit tests, and documentation should present the same information about thrown exceptions. Thus, an *exception specification unit* (ESU) is inconsistent if there is any divergence in its associated artifacts (code, documentation, or tests). This definition includes the cases where an artifact omits the ESU.

As the subject of the case study, we selected the latest version (at the time of the study) of Apache Commons IO (version 2.6, commit 11f0abe). This library consists of utility functions and classes, each mostly independent of the others, with well-defined ESUs. It is also extensively documented and tested, mature (over 18 years old), actively developed (over 500 forks on GitHub, and multiple weekly commits), and popular (over 20k dependents on Maven). Because the library contains a total of 152 top-level Java types, an amount which precludes an in-depth analysis of each method, we focused only on the public types in the root package `org.apache.commons.io`. We also excluded deprecated, abstract, and exception types, which resulted in eleven remaining classes and a total of 293 public, non-deprecated methods. Table 3 presents an overview of these classes, including the number of identified ESUs and template invocations for each class.

For each method declared in the classes under study, barring deprecated and private ones, one of the authors identified all ESUs present in at least one of the documentation, test suite, and source code. The identified ESUs include not only exceptions directly thrown by the method under investigation, but also those thrown by nested calls, which explains the large effort involved in eliciting the ESUs. For each ESU, the investigator noted the type of exception thrown, the state that triggers the exception, which of the source code, test suite, and/or documentation captured the ESU, and which DScribe template could be used to generate a unit test and documentation for this exception, creating the template if necessary. For the latter, if no template could capture the ESU, the investigator recorded the reason instead. Of the 849 ESUs identified, the investigator was not able to verify the correctness of 14 with respect to the source code. These 14 cases are included in Table 3, but we omitted them from the rest of the study.

TABLE 2
Overview of the empirical assessment of unit test generation for documentation

Study	Sect.	Subject	Scope	Purpose	RQs
Usefulness	6	Commons IO (root package)	Exceptions	Assess DScribe's ability to prevent inconsistencies	1, 2
Comparison	7	Commons IO (root package)	Unit tests	Compare tests from different tools	2
Validation	8	Commons Math, Lang, Config.	Tested specifications	Validate the results of the usefulness study	1
Limitations	9	Five open source projects	All	Understand DScribe's limitations	2

TABLE 3
Number of methods, exception specification units (ESUs), and instantiated DScribe templates per class under investigation.

Class	Methods	ESUs	Instances
ByteOrderMark	8	6	6
ByteOrderParser	1	1	1
Charsets	4	2	2
EndianUtils	30	67	67
FileCleaningTracker	7	8	8
FileDeleteStrategy	5	4	4
FileUtils	95	403	386
FilenameUtils	33	30	28
HexDump	2	5	5
IOUtils	101	315	295
LineIterator	7	8	8
Total	293	849	810

TABLE 4
Presence of exception specification units (ESUs) in documentation and unit tests. For each value, the number after the "+" sign indicates the number of ESUs that are not present in the source code.

	In Doc.	Not in Doc.	Total
In Test	122+1	29+0	152
Not in Test	458+9	216+0	683
Total	590	245	835

6.2 Results and Discussion

To answer RQ1, Table 4 summarizes the degree to which identified ESUs are consistent across the artifacts of Commons IO, by comparing the number of ESUs captured by the documentation (In Doc.) and the test suite (In Test). The results highlight the pervasiveness of information inconsistencies in Commons IO: 85% of the identified ESUs are missing in at least one of the documentation, test suite, or source code. An even more concerning observation is that the overwhelming majority (82%) of ESUs are untested, which increases the risk of documentation becoming silently inaccurate. This risk of documentation becoming silently inaccurate is already exemplified from the 10 cases where ESUs in documentation are not traceable to the source code. In this case, use of DScribe would also remediate the 19% of tested ESUs that are absent from the documentation, presumably by accident.

In some cases, an ESU was only partially or vaguely described in the documentation. Of the 590 ESUs present in documentation, 22 (4%) did not include the type of exception thrown, and 115 (19%) only described the input state that triggers the exception in broad terms, or aggregated multiple invalid states. A recurring example of such broad documentation in the FileUtils class is "IOException - if source [file] is invalid".

TABLE 5
Number of times that each DScribe template was instantiated. The five templates are variations of Template 1 presented in Table 1.

Invocations		Not Invoked	
Template	Count	Reason	Count
Static	237 (28%)	Inaccurate Doc.	10 (1%)
NotStatic	2 (0%)	Unable to Test	15 (2%)
MessageStatic	547 (66%)		
MessageNotStatic	19 (2%)		
MessageConstructor	5 (1%)		
Total	810 (97%)	**Total**	25 (3%)

Here, an API user is left wondering about the various specific invalid input states that may trigger the exception, such as a file that does not exist or that is a directory. Such cases, which decrease the usefulness of documentation, would be avoided by DScribe.

To answer RQ2, the investigator created the necessary templates and invocations to capture as many ESUs as possible. Table 5 shows the resulting templates, and the number of invocations for each of them, as well as the reason why we could not invoke any template for some ESUs. The fact that 97% of the identified ESUs could be captured by a template invocation confirms DScribe's potential to avoid future information inconsistencies. Each such invocation would lead to a unit test and a documentation fragment. Failing unit tests would instantly flag invocations inconsistent with the source code, thereby alleviating the burden of having to maintain ESUs in multiple artifacts manually.

The results also show that DScribe's ESU templates are highly reusable. Almost all ESUs (94%) were supported by only two templates, Static and MessageStatic. Thus, the overall relative cost of template creation is low. In our case, 810 ESUs (97%) were instantiated using only five templates. The five templates vary depending on the different types of focal methods (static, non-static, and constructor), and whether to verify the message of the exception. The templates NotStatic and MessageNotStatic, designated for non-static focal methods, were used less often as most methods under investigation were static. Similarly, the MessageConstructor template was not widely used because few ESUs were identified for constructors.

In addition to these results, we observed the use of three alternative patterns to test exceptions. Namely, using a try-catch block with JUnit's fail method, using JUnit's assertThrows method, and using helper methods to verify the type and message of an exception. It is thus evident that developers leverage recurrent templates naturally, but inconsistently. This inconsistency hinders readability and, consequently, maintainability. DScribe helps standardize the consistent use of recurrent templates, thus enhancing the

quality of test suites.

We did not instantiate ESUs in only 25 cases (3%), due to two main reasons. In ten cases, an ESU in the documentation was inaccurate, i.e., its description did not reflect the actual behavior of the method. While it is possible to instantiate these ESUs, it would lead to failing unit tests and outdated documentation. For the 15 other ESUs, we could not produce input states that would trigger the target exception. For example, it is not possible to ensure that an InputStream instantiated inside a method, rather than passed as an input parameter, produces an IOException when it is read. Instantiating those ESUs would require considerable scaffolding to precisely manipulate the file system, which was beyond the scope of this study. The majority of these cases were also not tested in the test suite.

Findings: Among the sampled methods of Commons IO, only 15% of the specifications related to thrown exceptions (ESU) are both tested and documented. The remaining 85% of ESUs represent inconsistent information that would require a non trivial amount of effort to manually fix. DScribe can generate the tests and documentation for 97% of these ESUs, using only five templates.

6.3 Threats to Validity

Two of the authors performed all annotations, with no overlap between the annotators. It is possible that the investigators may have missed some ESUs, or misinterpreted the purpose of a test or behavior of a method, as they are not part of the development team for the library under test. We mitigated this threat by selecting a library that requires little specialized knowledge, with self-contained methods that can be understood without knowledge of the system as a whole.

Nevertheless, to assess this threat, we selected 20 methods from our sample at random to be independently annotated by two additional authors, providing three sets of annotations for these methods. In total, the annotators found 49 ESUs for these methods, ten of which were not found by the original annotator. These missed ESUs are exceptions thrown by methods deep in the call sequence of the target API, and that are not typically described in documentation, such as SecurityExceptions thrown when the file system does not have sufficient permission to perform some operations. In seven cases, the original and additional annotators agreed about the existence of an ESU for a method, but not about which of the source code, unit tests, or documentation captured it, mostly when the additional annotators failed to notice a part of an artifact. Thus, although the number of ESUs may actually be an underestimation, our conclusions about the amount of information inconsistency are reliable. To ensure verifiability, we include the complete results of our study in our on-line appendix.

A threat to external validity stems from our decision to focus on exception handling. We do not expect that this context would generalize to all types of units of specification. Moreover, we only investigated eleven classes from Commons IO. The results may not generalize to the library as a whole, let alone other systems. Similarly, our results are dependent on our selection of templates. Different templates may not be as reusable, and we did not evaluate the challenges developers have to use templates created by others. Nevertheless, the natural format in which a template is expressed (i.e., a syntactically valid code fragment coupled with a natural language description) favor effective reuse, and the study demonstrates the usefulness of DScribe in at least one realistic software development context, as we applied it to the popular Commons IO library, from which we can analytically generate to similar software components.

TABLE 6
Number of tests generated by each approach (including the original test suite). The percentage of branch and line coverage, respectively, is reported in parentheses.

Class	Original	DScribe	EvoSuite	Randoop
BOM	9 (100; 100)	6 (33; 30)	12 (100; 100)	3251 (88; 92)
BOP	3 (100; 100)	1 (50; 60)	3 (100; 75)	3 (50; 67)
Char	8 (75; 94)	2 (50; 47)	5 (100; 100)	4279 (50; 94)
EUtil	22 (100; 100)	67 (50; 8)	21 (100; 100)	6746 (0; 42)
FCT	9 (93; 98)	8 (40; 44)	8 (67; 79)	2922 (86; 85)
FDS	5 (100; 90)	4 (25; 53)	7 (100; 82)	3852 (13; 53)
FUtil	181 (86; 90)	386 (51; 45)	166 (73; 86)	668 (51; 60)
FNUtil	45 (94; 97)	28 (25; 26)	90 (76; 83)	6988 (45; 57)
HexD	1 (100; 96)	5 (83; 80)	6 (100; 100)	7982 (21; 37)
IOUtil	190 (85; 87)	295 (44; 42)	115 (68; 77)	3355 (64; 64)
LineIt	16 (100; 100)	8 (44;57)	8 (95; 100)	3 (7; 16)

7 Comparison Study

We assessed the quality of the tests generated by DScribe relative to three baselines: the original test suite of Commons IO, and tests generated by two automated approaches. The objective of this study was to assess whether it is possible to generate high-quality unit tests using DScribe with reasonable effort relative to other techniques. We did not evaluate the quality of the documentation fragments generated by DScribe, because templates for documentation are free form and can be trivially improved at any point.

7.1 Comparison Study Design

We used the unit tests produced during the usefulness study as the target population for this study. These tests are relevant to a popular software library, and were created following a systematic procedure with a well-defined scope. These 810 tests are also derived from only five templates, mitigating the threat of using overly specific templates to inflate the quality of the generated tests.

We compared the quality of DScribe's tests with the current test suite of the Commons IO project. We also used two state-of-the-art fully-automated test generation tools as additional baselines: EvoSuite [13] and Randoop [14]. Randoop is an approach to generate large regression test suites using random sequences of method calls. EvoSuite attempts to generate a test suite that optimizes several coverage criteria (line and branch coverage in our case) while minimizing the number of generated tests. We used the most recent version of both tools (EvoSuite 1.1.0 and Randoop 4.2.5) with their default parameters on each of the eleven Commons IO classes one by one (it was necessary to use the call-timeout option for Randoop on the FileUtils class). Table 6 shows the number of tests in each test suite, for each tested class, as well as the branch and line coverage for the class as computed by the EclEmma code coverage tool.

TABLE 7
Ratio of unit tests considered focused (F) and readable (R) by each annotator (An.). Asterisks mark scores that are statistically lower than DScribe's scores at the 0.05 (*) or 0.001 (**) significance level, using a one-tailed Mann-Whitney U test to compare the distribution of scores.

An.	DScribe		Original		EvoSuite		Randoop	
	F	R	F	R	F	R	F	R
A	1.00	0.85	1.00	0.90	1.00	0.65	0.80*	0.30**
B	1.00	1.00	0.80*	0.50**	0.00**	0.60**	0.00**	0.05**
C	1.00	0.80	0.50**	0.25**	0.80*	0.40*	0.50**	0.00**
D	1.00	0.30	0.15**	0.65	0.30**	0.85	0.35**	0.10*
All	1.00	0.74	0.61**	0.58*	0.53**	0.63*	0.42**	0.11**

To evaluate the quality of the unit tests, we relied on the design principles described by Martin [15, Chapter 9]. Specifically, unit tests should be *readable* (i.e., *clean tests*) and *focused* (i.e., *single concept per test*). Because these two properties rely on human comprehension, we asked four external annotators, all with at least three years of Java programming experience and prior experience in writing unit tests, to evaluate the tests. We selected 20 tests per suite per annotator, for a total of 320 evaluated tests (80 per test suite). After reading a definition of the *readable* and *focused* properties, the annotators rated both qualities independently for each test on a five-point ordinal scale (1=not readable/focused, 5=readable/focused).

We provided the set to annotate to each annotator with all tests in a random order, so that annotators were unaware of the origin of each test and the number of different generation tools. To further reduce the distinction between test suites, we ported test scaffolding to the same single helper class for all four suites, removed inline comments, formatted tests using a consistent style, and ported JUnit 4 types to JUnit 5. We also removed automated generation artifacts such as meaningless test names (e.g., `test0123`), fully qualified type names (replaced with import statements), unnecessary `throws` clauses, and `if` statements used for debugging. These adjustments only increased the quality of tests from EvoSuite and Randoop, for a fair comparison.

7.2 Results and Discussion

Despite the five-point scale, the annotators gave the highest score to a majority of tests, for both properties. Although this observation in itself is encouraging, showing that four independent annotators found the tests suitable in terms of scope and readability, it makes the comparison between generation approaches more difficult. To analyze the results, we transformed the five-point scale into a binary decision, where a test is either focused (resp. readable) if the annotator gave the highest score, or not if the score is lower.

Table 7 shows the proportion of tests rated with the highest score for each test suite and quality. Already, the differences between annotators demonstrate the subjectivity of the task (especially for readability, as was previously observed [16]), not only due to the scale, but also to each developer's interpretation of good unit testing practices. However, despite these differences, all annotators agree that DScribe can produce unit tests that are focused, and three out of four annotators (A, B, C) found the majority of DScribe's tests readable, on par with or exceeding even manually written tests. The last annotator commented that DScribe's tests should rely on more local variables to increase readability, and thus gave a score of four to most DScribe tests.

Findings: Overall, DScribe produced a higher ratio of tests marked as readable and focused than human contributors and two alternative test generation approaches.

7.3 Threats to Validity

There are two main threats to the validity of this study. First, the sample of tests under study is not sufficient to generalize our results beyond the context of the Commons IO project. The decision to use only ESUs for DScribe's tests also resulted in a different scope for DScribe compared the baselines, which could favorably bias DScribe's scores for the *focused* quality. However, DScribe is not intended as a single comprehensive test generation solution, but rather as a complement to other testing tools (or manually-written tests). Therefore, it is a reasonable scenario to use DScribe only for testing a small number of repetitive specifications.

The second threat relates to the subjectivity of the annotation task. To mitigate this threat, we provided the same definition of the two qualities to all annotators, and required all annotators to have their own prior experience of unit testing. However, beyond these definitions, we let each developer rate tests according to their own experience and judgment, instead of training them to adhere to our own guidelines, which would inevitably introduce biases. We report the results of each annotator independently, rather than only in aggregated form, to let readers interpret the subjectivity of the task.

8 Validation Study

The results presented in Section 6 indicate that information inconsistency across source code, documentation, and unit tests is a common issue for exception handling in the Commons IO project. We performed a multi-case study to validate and expand the findings of the initial case study.

8.1 Validation Study Design

Because identifying all units of specifications from the *source code* of a method is both effort-intensive and subjective (due to the ambiguity of what constitutes a single "unit"), this second study focused on the units of specifications found in *unit tests*. This design restricts the scope of the validation study to testable (and tested) specifications, but it is necessary to make the findings reliable.

As the subjects of the validation study, we selected the three Apache Commons projects with the most unit tests: Math (version 3.5, commit d7d4e4d, 3757 tests), Lang (version 3.8.1, commit 2ebc17b, 3086 tests), and Configuration (version 2.4, commit 61732d3, 2554 tests). We randomly sampled tests uniformly from the total population of 9397 tests. For each sampled test, one author manually identified the focal unit of the test, using the test's name, the *Last Call Before Assert* heuristic [17], and comments. Because we were focusing on specifications about methods in the production code, we rejected tests whose focal unit was not a single method (e.g., multiple methods, or a class or field), or if the focus was ambiguous. We also rejected degenerate cases (e.g., empty, deprecated, or auto-generated tests). We continued the

TABLE 8
Number of unit tests capturing at least one specification (documented or not). Percentages are computed with respect to each project.

Project	Information Present			No Info.	Total
	Doc.	Partial	Not doc.		
Config.	9 (9%)	3 (3%)	16 (15%)	76 (73%)	104
Lang	55 (39%)	10 (7%)	20 (14%)	57 (40%)	142
Math	17 (14%)	3 (2%)	24 (19%)	80 (65%)	124
Total	81 (22%)	16 (4%)	60 (16%)	213 (58%)	370

sampling until we gathered a set of 370 viable tests, rejecting a total of 93 unsuitable tests. This sample size is sufficient to support a generalization of proportions of tests computed on the sample to the whole population within a 5% confidence interval at the 0.95 level.

For each test in the sample, one author noted whether the test captured at least one unit of specification about the focal method (some complex tests actually tested multiple inputs), writing it down to ensure it was well-defined. Each specification was expressed as *If X, then calling the method will do Y*, to avoid considering all information (including, e.g., usage examples) as a specification. For each identified unit of specification, the investigator then noted whether it was described in the documentation, and if so, if the description was only partial and broad, or complete and explicit.

8.2 Results and Discussion

Table 8 presents, for each project, the number of tests that captured self-contained information about a specification of its focal method (*Information Present*), or not (*No Info.*), and whether the information was completely included in the documentation (*Doc.*), partially or broadly (*Partial*), or not mentioned at all (*Not doc.*). The information captured by the tests varied. As expected, many tests capture similar information, for example a special behavior of a focal method when an argument consists of a special value (e.g., null), and other tests captured more unique information, for example that the leading spaces of a string argument are discarded or that the return value of a method is immutable. However, the variety of testing practices resulted in many different ways to capture similar information, each involving small variations. Therefore, in our analysis, we only use a binary variable to indicate the presence of information as we cannot reliably categorize tests based on whether they capture the same information.

Overall, 42% of tests captured at least one unit of specification, which means that a significant amount of tests need to be kept consistent with documentation. This proportion is even higher (60%) for Lang. For Configuration and Math, undocumented specifications amount to over half of the tested specifications, a situation that the use of DScribe prevents. In the case of Lang, although the lack of consistency is less significant, the use of DScribe would reduce the effort required to produce and maintain the more extensive documentation.

Multiple factors can explain the absence of unit of specification in the remaining 58% of tests. In many cases, a test was simply verifying that under "usual" inputs, a method behave as it should. For example, the test KendallsCorrelationTest.testSimpleReversed() in Math simply validates that the correlation computed in a specific (normal) scenario is correct. Other cases, however, were more ambiguous: some tests captured at least a partial unit of specification, but the complete information was obscured by external references or ambiguous names. For example, TestDataConfiguration.testGetByteArray() in Configuration follows some recognizable patterns, but depends on values from configuration files referred to as byte.list1, byte.list2, etc. In such cases, the investigator used a conservative strategy and marked the test as capturing no specification. Nevertheless, refactoring the tests, or generating them with DScribe, could make them more self-contained, thus improving their quality. Numbers reported in Table 8 should thus be regarded as lower bounds of the effective values.

This study also revealed an interesting use case for DScribe outside the scope of this study: developers can use templates to add *usage examples* to documentation. For example, the documentation of StrBuilder.asTokenizer(), from the Lang project, contains a usage example that is almost equivalent to the test StrBuilderTest.testAsTokenizer().

Findings: Approximately 42% of the unit tests of Commons Configuration, Lang, and Math capture information that would be relevant to document, but that information is documented for only 52% of these tests. Developers can use DScribe to generate documentation from these tests more consistently and efficiently, especially for repetitive specifications.

8.3 Threats to Validity

As for the usefulness study, a single author performed all annotations, which leads to the same threats outlined in the Section 6.3. However, as it is common in case studies, this procedure was necessary to obtain detailed insights that require a degree of interpretation.

Nevertheless, we selected a random sample of 50 tests to be independently annotated by two other authors. This additional annotation shows the subjectivity of the task, as the additional annotators agreed only on 31 and 33 of the tests (Cohen's $\kappa = 0.25$ and 0.29 [18], interpreted as *fair* by Landis and Koch [19]), respectively, reinforcing the motivation to conduct our initial usefulness study on a narrow scope. Nevertheless, the independent results of all three annotators lead to the same general conclusion: that a non trivial amount of information in tests is not properly documented, even if precisely quantifying information is challenging. For verifiability, we include the complete results in our on-line appendix.

The target systems are a collection of mostly independent utility methods and classes, with extensive test suites, from the same organization as the usefulness study. We do not expect that this context would generalize to all systems. We recognize this limitation, and we scope our claims accordingly. Nevertheless, the evaluation shows evidence of a considerable amount of information inconsistencies in a realistic and significant software development context.

9 Limitations Study

The usefulness study showed evidence of the potential effectiveness of DScribe in one particular context, in which 97% of the identified exception specification units could be

TABLE 9
Five open source subjects of the limitation study

System	Prod. Files	Test Files	Inspected Files
Freemind	379	26	18
Eclipse	3933	1669	49
Weka	1614	253	20
Tomcat	1402	475	16
Hibernate	3845	5647	12

captured by template invocations. However, to better answer RQ2 about the use of templates in a general context, we performed a qualitative multi-case study specifically to elicit the strengths and limitations of a template-based approach for generating unit tests and documentation.

9.1 Limitations Study Design

To ensure a variety of contexts, we selected five open source projects that are at least 15 years old and that vary in their development style, target audience, and application domain: Freemind (version 1.1.0, commit 643c55c), Eclipse Platform UI (version 4.9.0, commit d6d8a6a), Weka (version 3.9.3, commit r14866), Apache Tomcat (version 9.0.11, commit r183513), and Hibernate ORM (version 5.3.2, commit 35806c9).

One author annotated a subset of the test suite of each project. For each test, the investigator answered the question *What are the technical factors that would enable or prevent the generation of similar unit tests from templates?* To help answer this question, the investigator noted the unit under test, purpose, format, and recurrent patterns for each test, in addition to the enabling and hindering factors.

To achieve maximal purposive sampling, instead of annotating a fixed subset of each project, the investigator iteratively included more unit tests to the sample until reaching saturation, which we defined as when three consecutive iterations generated no new noted observations. Each iteration consisted of selecting a package with at least three classes at random from the test suite of a project, then selecting three random classes (or more if the classes or package are small enough) from that package, and annotating all tests from these classes. The investigator analyzed one project at a time, moving to the next once saturation was reached for one. For Freemind, which only contains two test packages, the investigator annotated all unit tests from the root package of the test suite. Table 9 shows the number of production and tests Java files for each project, in the order they were annotated, as well as the number of inspected test files.

The investigator initially used an open coding process [20] to annotate each test. After completing the open coding, and after a preliminary analysis of the initial codes, the investigator systematically re-coded each test using a closed code catalog.

9.2 Results and Discussion

We identified eight technical factors that can impact the ability to generate unit tests from templates or the qualities of the generated tests. We discuss these factors at a high level in this section, but the interested reader can find multiple concrete examples of each factor in our on-line appendix. Although these factors outlined several limitations for using DScribe in different contexts, they also revealed simple strategies to work around these limitations, which can improve the quality of the generated tests.

Generic Variable Names: Recurrent generic names for local variables (e.g., input and expected) better support the reuse of templates than names specific to the test context (e.g., baseString, encodedString). Although only a minority of the studied tests used such generic identifiers, generic names allow unfamiliar readers to understand new tests quickly by identifying recurrent aspects. Thus, despite being a limitation of template-based approach, this factor can be beneficial in the long term.

Structured Test Names: A template-based approach can help standardize local conventions. For example, it is considered good practice to use meaningful names for unit tests, usually following a fixed convention. As an extreme example of a highly-structured name, all test names in Tomcat's class CheckOutThreadTest match the pattern test(DBCP|Pool)-Threads(10|20)Connections(10|20)(Validate)?(Fair)?. This local structure could be encouraged with a specialized template for this class.

Recurrent Complex Operations: A common limitation of template-based approaches is the diverging implementations of similar operations. For example, verifying that the content of two objects varies based on their internal structure. This variation can prevent tests that follow the same high-level patterns to be generated from the same template. Encapsulating recurrent complex operations into helper methods, especially with meaningful generic names such as assertContentEqual, can mitigate this limitation. This strategy can be abused: As an extreme, but not unique, example, Tomcat's helper method TestELParser.doTestParser encapsulates all operations of multiple tests, resulting in a complex logic that is hard to maintain. In fact, a template-based generation of the tests in this class would have achieved the same objective (i.e., writing the recurrent structure of many tests only once) while keeping each test decoupled and more readable.

Complex Assertions: Some tests require complex assertion structures, such as nesting assertions inside methods of custom mock objects. Such structures can severely limit the applicability of templates, and thus the usefulness of a template-based approach. However, if the same pattern of complex assertion is often needed, a template-based approach can encourage testers to develop a more systematic approach to test complex behaviors, with the necessary scaffolding.

Testing Preconditions: Several tests include assertions to verify the input state of tested objects before the method under test is performed.[1] Although a template can include such early assertions, they are often specific to the tested objects or the test case, so they are not well suited for template-based generation. To mitigate this limitation, testers can use factory methods that include the necessary assertions to create the tested objects in the right input state.

1. This practice is controversial among testing communities. We do not take a stance in this debate, but recognize the importance of supporting this feature as some tester may desire it.

Constrained Resources: Tests that rely on constrained resources, such as connections to external servers, multiple threads, or even read and write operations to the file system, may involve unique operations to avoid corrupting the resources or control errors originating from the constrained resources themselves (e.g., trying twice to connect to a flaky server). These operations can prevent the use of templates that are not designed specifically for handling these resources. Mitigation strategies for this limitation include an efficient use of `setup` and `teardown` methods (using JUnit's `@Before` and `@After` annotations), and the creation of local templates.

Different Units Under Test: In the sampled set of tests, the unit under test was not always a single method. Some tests focused on a whole class, whereas others focused on validating a single field. For example, Freemind's test `HtmlConversionsTest.testEndContentMatcher` validates the expected behavior of a regular expression encoded in a constant field. Our approach assumes that each test targets a method, but this assumption could be modified in future work.

Variety of Test Purpose: Unit tests can serve various purposes, e.g., testing a typical behavior using arbitrary input values or a corner case with special inputs. Tests for typical behaviors are less likely to capture precise information that need to be documented, compared to corner cases that should be thoroughly described. Similarly, tests that verify the correct interaction of various components in complex scenarios and tests specifically tailored to a historical defect typically do not capture information valuable for documentation, and involve unique sequences of operations. Thus, a template-based approach would not completely remove the need for manually-crafted tests.

Findings: We identified eight technical factors that can improve or limit the effectiveness of DScribe in certain contexts. These factors relate to the testing constraints (e.g., complex assertion procedures) and contexts (e.g., reliance on an external server), as well as to the testing conventions (e.g., recurrent identifiers or number of use cases in the same test). Developers can consider those factors before deciding whether to use DScribe.

9.3 Threats to Validity

The case study relied on the identification of testing patterns, a concept influenced by the experience of each developer. Hence, our conclusions may reflect the personal experience of the investigator. This limitation was necessary because the data analysis required a very high initial effort investment, and a consistent point of view, to study the five systems. Thus, the coding procedure could not be packaged into multiple sets of data to be labeled by independent coders. Hypothetical external coders would have to be extensively trained to have the in-depth knowledge of the template-based approach required for the task, which would re-introduce the risk of bias. For transparency and independent verification of the quality of the coding, the on-line appendix contains several concrete examples supporting each of our conclusions.

10 Related Work

The difficulty of maintaining high-quality documentation [5], [21], [22], [23] led to a vast exploration of **automated documentation generation** approaches [24]. Techniques proposed in prior work involve static [25] and dynamic [26] analysis of the body of methods, as well as their context [27]. Different techniques are tuned to document either classes [28], methods [29], [30], or method parameters [25]. Techniques also differ in the kind of documentation they generate, such as specifications [31], [32], [33], program invariants [34], test summaries [35], [36], and usage scenarios and examples [37], [38]. These techniques contribute insights to automatically produce more documentation with minimal effort. However, an inherent limitation of *fully* automated techniques is the inevitable threat of false positives, and the possibility of diluting important insights within large amounts of trivial information generated automatically. These techniques thus require manual effort to filter documentation after its generation. DScribe proposes an alternative to this process by leveraging developer effort before the generation, keeping them in control of what information is added to the system.

A vast number of techniques have also been proposed to **automatically generate tests**. Notable early work includes CUTE [39] and DART [40], which introduced the concept of *concolic* testing. Concolic testing couples a symbolic and concrete execution of a program to explore the space of inputs that will trigger different responses from the program. Thummalapenta et al. [41] generate test cases by extracting sequence of method calls to create relevant input states. Pacheco et al. [14] proposed Randoop, a technique to generate test cases by randomly creating sequences of execution, with a feedback loop to inform the next generations. Fraser and Zeller [42] follow a more systematic random generation approach by leveraging mutation operators, and using genetic algorithms to optimize the test suite. Taneja and Xie [43] leverage the version history of a project to create test cases. Other techniques focus only on the generation of test cases that can crash a system [44], that apply to multi-threaded code [45], or that map to the system's UML diagrams [46]. Automated test generation techniques suffer from the same limitation than documentation generation techniques: by aiming to exclude developers from the generation process, they are susceptible to false positives, which in turn requires human effort to sift through their output. In contrast, DScribe aims to optimize early human effort to create trustworthy assets. Similarly, Gaston and Clause recently proposed a technique to suggest missing tests for a method based on existing tests for similar methods [8]. This technique could complement DScribe, as both leave the decision to generate a test to developers.

The value proposition of DScribe, however, extends beyond the generation of tests and documentation. An important benefit is the **traceability links** of the generated artifacts to the method they complement. Documentation traceability is a challenging problem [47], [48], but it is a prerequisite to validate the correctness of the documentation, another challenging problem [49], [50]. By generating both tests and documentation from the same invocations, DScribe solve both problems: following a change in a method specification, updating the invocation related to a failing unit test will also update the corresponding outdated documentation. This solution is similar to that of behavior-driven development (BDD) [51], a methodology derived from test-driven development [52]. BDD recognizes the documentation potential

of testing code, and BDD frameworks such as JBehave [53] integrate documentation fragments directly into unit tests, so that documentation is again backed by passing tests. However, contrary to DScribe, BDD frameworks require developers to write both the testing code and documentation fragments, a repetitive and redundant effort.

Finally, our research is related to the field of **code pattern mining**, which parses large corpora of source code to identify regularities in the usage of various type of code elements (e.g., functions). The objective of these techniques is to identify specifications [37], [54], [55], and in particular violations of these implicit specifications, or design patterns [56], [57]. Future work can leverage a similar approach to automatically generate DScribe templates, to further reduce the initial burden of developers.

11 Conclusion

Motivated by the observation that documentation and testing code often capture redundant and repetitive information, we designed a technique, called DScribe, to allow developers to decouple aspects of unit testing and documentation that relate to repetitive specifications from the aspects specific to each instance. This technique can partially relieve developers of the burden of maintaining a consistent and extensive documentation and test suite.

A four-phase investigation of the inconsistencies in selected mature software projects revealed their pervasiveness in testing code and method documentation, with 85% of the specifications about exceptions thrown by the Apache Commons IO methods either untested, undocumented, or both. The investigation revealed that DScribe could have prevented 97% of these inconsistencies in a favorable context, with generated tests that external annotators found both more readable and focused than three baselines. A systematic study of the limitations of DScribe provided rich descriptions of technical factors that facilitate or hinder the co-generation of tests and documentation in varied contexts.

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.

References

[1] B. Dagenais and M. P. Robillard, "Using traceability links to recommend adaptive changes for documentation evolution," *IEEE Trans. Softw. Eng.*, vol. 40, no. 11, pp. 1126–1146, 2014.

[2] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, 2015.

[3] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empir. Softw. Eng.*, vol. 10, no. 1, p. 31–55, 2005.

[4] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A survey," in *Proc. ACM Symp. Document Eng.*, 2002, p. 26–33.

[5] T. C. Lethbridge, J. Singer, and A. Forward, "How Software Engineers Use Documentation: The State of the Practice," *IEEE Softw.*, vol. 20, no. 6, pp. 35–39, 2003.

[6] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 1199–1210.

[7] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *J. Syst. Softw.*, vol. 138, pp. 52–81, 2018.

[8] D. Gaston and J. Clause, "A method for finding missing unit tests," in *Proc. IEEE Int. Conf. Softw. Maintenance and Evolution*, 2020, pp. 92–103.

[9] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Softw. Eng.*, vol. 39, no. 5, pp. 613–637, 2013.

[10] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman. (2020) The Java language specification. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se14/html/index.html

[11] O. Lassila and R. R. Swick, "Resource description framework (RDF) model and syntax specification," W3C, W3C Recommendation, 1999. [Online]. Available: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

[12] E. A. Barbosa, A. Garcia, M. P. Robillard, and B. Jakobus, "Enforcing exception handling policies with a domain-specific language," *IEEE Trans. Softw. Eng.*, vol. 42, no. 6, pp. 559–584, 2016.

[13] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[14] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 75–84.

[15] R. C. Martin, *Clean Code – a Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

[16] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 107–118.

[17] B. Van Rompaey and S. Demeyer, "Establishing Traceability Links between Unit Test Cases and Units under Test," in *13th IEEE European Conf. Softw. Maintenance and Reengineering*, 2009, pp. 209–218.

[18] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[19] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.

[20] M. B. Miles, A. M. Huberman, and J. Saldana, *Qualitative data analysis*. Sage, 2013.

[21] B. Fluri, M. Würsch, and H. C. Gall, "Do code and comments co-evolve? On the relation between source code and comment changes," in *Proc. Work. Conf. Reverse Eng.*, 2007, pp. 70–79.

[22] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*iComment: Bugs or Bad Comments?*/," in *Proc. ACM Symp. Operating Syst. Princ.*, 2007, pp. 145–158.

[23] I. K. Ratol and M. P. Robillard, "Detecting Fragile Comments," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 112–122.

[24] K. Nybom, A. Ashraf, and I. Porres, "A systematic mapping study on API documentation generation approaches," in *Proc. 44th Euromicro Conf. Softw. Eng. and Adv. Appl.*, 2018, pp. 462–469.

[25] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *Proc. IEEE Int. Conf. Program Comprehension*, 2011, pp. 71–80.

[26] M. Sulír and J. Porubän, "Generating Method Documentation Using Concrete Values from Executions," in *Proc. Symp. Lang., Appl. and Technol.*, 2017, pp. 3:1–3:13.

[27] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 279–290.

[28] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proc. 21st Int. Conf. Program Comprehension*, 2013, pp. 23–32.

[29] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2010, pp. 43–52.

[30] N. J. Abid, N. Dragan, M. L. Collard, and J. I. Maletic, "Using stereotypes in the automatic generation of natural language

summaries for c++ methods," in *Proc. IEEE Int. Conf. Softw. Maintenance and Evolution*, 2015, pp. 561–565.
[31] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proc. 29th ACM SIGPLAN-SIGACT Symp. Princ. of Program. Lang.*, 2002, pp. 4–16.
[32] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia, "Static Specification Mining Using Automata-Based Abstractions," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 651–666, 2008.
[33] C. Le Goues and W. Weimer, "Specification mining with few false positives," in *Proc. Int. Conf. Tools and Algorithms for the Construction and Anal. of Syst.*, 2009, pp. 292–306.
[34] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin, "Dynamically discovering pointer-based program invariants," in *Proc. Int. Conf. Softw. Eng.*, vol. 373, 1999.
[35] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 547–558.
[36] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 625–636.
[37] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications," in *Proc. 6th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. Found. of Softw. Eng.*, 2007, pp. 25–34.
[38] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 782–792.
[39] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for c," in *Proc. 10th European Softw. Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2005, p. 263–272.
[40] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. and Implementation*, 2005, p. 213–223.
[41] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," in *Proc. 7th joint meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 193–202.
[42] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 278–292, 2012.
[43] K. Taneja and T. Xie, "Diffgen: Automated regression unit-test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2008, pp. 407–410.
[44] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for java," *Softw.: Pract. and Exp.*, vol. 34, no. 11, pp. 1025–1050, 2004.
[45] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 727–737.
[46] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," in *UML'99 – The Unified Modeling Language*, 1999, pp. 416–429.
[47] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proc. 25th Int. Conf. Softw. Eng.*, 2003, pp. 125–135.
[48] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, 2002.
[49] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs Documentation and Code to Detect Directive Defects," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2017, pp. 27–37.
[50] E. Ben Charrada, A. Koziolek, and M. Glinz, "Identifying outdated requirements based on source code changes," in *Proc. IEEE Int. Requirements Eng. Conf.*, 2012, pp. 61–70.
[51] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *Proc. Int. Conf. Modelling Techn. and Tools for Comput. Perform. Eval.*, 2012, pp. 269–287.
[52] K. Beck, *Test-driven development: by example*. Addison-Wesley, 2003.
[53] JBehave.org. (2017) What is JBehave? [Online]. Available: https://jbehave.org/
[54] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 472–483.
[55] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *Proc. Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. Found. Softw. Eng.*, 2005, pp. 306–315.
[56] J. Dong, Y. Zhao, and T. Peng, "A Review of Design Pattern Mining Techniques," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, no. 06, pp. 823–855, 2009.
[57] A. Pandel, M. Gupta, and A. K. Tripathi, "DNIT – A new approach for design pattern detection," in *Proc. Int. Conf. Comput. and Commun. Technol.*, 2010, pp. 545–550.

Mathieu Nassif is a Ph.D. student in Computer Science at McGill University, under the supervision of Martin Robillard. His research focuses on the extract, representation, and manipulation of knowledge in software systems to optimize the contribution of developers to the system. Mathieu received his M.Sc. in Computer Science from McGill University and his B.Sc. in Mathematics from Université de Montréal.

Alexa Hernandez is an M.Sc. student in Computer Science at McGill University, under the supervision of Martin P. Robillard. Her research aims to understand the structure of and relationship between software documentation and tests to facilitate their creation and maintenance. Alexa received a B.A. in Computer Science at McGill University.

Ashvitha Sridharan is a software engineer optimizing the edge network at Shopify. Her research interests include software design, maintenance, and evolution. Sridharan received a B.Sc. Computer Science at McGill University, Montreal, where she worked under the supervision of Martin P. Robillard.

Martin P. Robillard is a Professor of Computer Science at McGill University. His research investigate how to facilitate the discovery and acquisition of technical, design, and domain knowledge to support the development of software systems. He served as the Program Co-Chair for the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012) and the 39th ACM/IEEE International Conference on Software Engineering (ICSE 2017). He received his Ph.D. and M.Sc. in Computer Science from the University of British Columbia and a B.Eng. from École Polytechnique de Montréal.