

Generating Unit Tests for Documentation

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

Abstract—Software projects capture information in various kinds of artifacts, including source code, tests, and documentation. Such artifacts routinely encode information that is redundant, i.e., when a specification encoded in the source code is also separately tested and documented. Without supporting technology, such redundancy easily leads to inconsistencies and a degradation of documentation quality. We designed a tool-supported technique, called DScript, that leverages redundancy between tests and documentation to generate consistent and checkable documentation and unit tests based on a single source of information. DScript generates unit tests and documentation fragments based on a novel template and artifact generation technology. By pairing tests and documentation generation, DScript provides a mechanism to automatically detect and replace outdated documentation. Our evaluation of the Apache Commons IO library revealed that of 835 specifications about exception handling, 85% of them were not tested or correctly documented, and DScript could be used to automatically generate 97% of the tests and documentation.

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

1 Introduction

MATURE software frameworks and libraries are usually complemented by extensive test suites and reference documentation. For example, the Apache Commons Math project release 3.6 is supported by 4467 tests and 215 176 words of method reference documentation. Although unit tests and reference documentation serve different purposes, their creation involves expressing the same or similar information in different software artifacts, which must then be kept consistent. An example of a pervasive case is that of a function that throws a specific type of exception when supplied with an invalid argument. Normally, such behavior should be described in the function's documentation, and tested by a unit test. Ideally, the test and the documentation would be consistent.

As this simple scenario illustrates, current practices for testing and documenting reusable software assets exhibit three inter-related problems. First, manually-created tests and documentation are often *redundant*. In turn, this redundancy introduces the risk of *inconsistencies* between a documented specification and the exercise of the corresponding behavior in a test. Finally, in situations where many functions in a library exhibit similar constraints (e.g., on input validation), the redundancy between tests and documentation exacerbates the *repetitiveness* of the testing and documentation effort.

The goal of our research is to leverage the redundancy and repetitiveness of information in software artifacts to reduce the amount of developer effort, as well as the threat of inconsistencies. To advance towards this goal, we investigate a solution that explores a new synergy between template-based unit test and documentation generation.

Although, at an abstract level, generating unit tests may seem relatively straightforward, realizing this idea in practice required addressing many new technical challenges with original solutions. We explored this design space by fully

developing a prototype technique, called DScript, that can generate unit tests for Java systems.

DScript is a tool-supported technique for transforming facts about methods between different types of equivalent representations. DScript relies on a database of fact templates, and users invoke a template to instantiate a specific fact about a method into a unit test and corresponding block of documentation. To realize this functionality, the design of DScript incorporates, among others, a new template definition language and original algorithms for aggregating related fragments of documentation into a cohesive unit.

As a research project focused on engineering design, our assessment of DScript focused on gaining an understanding of the potential usefulness of the approach, its applicability, and its limitations. A study revealed that 85% of the specifications about exceptions thrown by the methods of the Apache Commons IO library are either untested, undocumented, or both. In addition, the investigation revealed that DScript could have prevented 97% of these inconsistencies. In a wider study of the applicability of DScript, we found that 42% of the tests in three additional Apache commons projects captured at least one unit of specification, which means that a significant amount of tests need to be kept consistent with documentation.

The main contribution of this paper is the complete design and implementation of a prototype technique for generating unit tests for documentation. Although this technique only represents one point in a wide design space, we also contribute numerous insights about the rationale for important design and implementation solutions that can inform future work in that direction. We also contribute three empirical studies that provide different insights on the general potential for generating unit tests for documentation. Although they leverage our work on DScript, the studies are not specific to the tool, and thus the observations they generated can provide insights that go much beyond the application of a given prototype.

This article is organized as follows. In Section 2 we provide a general overview of DScript, followed by sections that supply the details on the two key aspects of the approach: templates (Section 3) and generative technology (Section 4). Section 5 is

• 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 ...

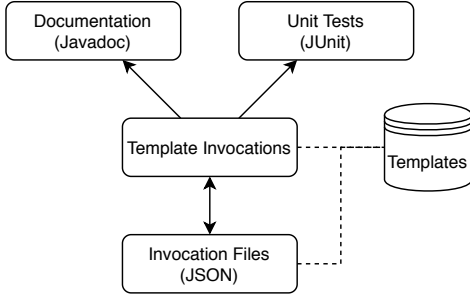


Fig. 1. Information representations in DScript. Rounded rectangles show different representations of facts, and arrows indicate transformations supported by DScript. Dashed lines indicates a dependency to a set of templates.

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

Research Artifacts

Our contributions are complemented by an on-line appendix, which contains the source code of DScript, details of the evidence collected as part of the studies, as well as additional details on the implementation of the technique.

<https://github.com/prmr/DScript-Research>

2 DScript Overview

At its core, DScript is an approach to transform *facts* about software elements between different types of equivalent representations. In the current implementation of DScript, facts must relate to a single public Java method, called the *focal method*.

For example, a fact could be that if the focal method receives the value `null` as argument, it throws a `NullPointerException`. The method declaration encodes this fact by providing an implementation. However, the fact can also be redundantly encoded in other representations, such as unit tests or natural language documentation.

In addition to their redundant representations, some facts can have a repetitive structure when minor variations of the same fact apply to multiple methods. DScript comprises a database of fact *templates*, where each template captures one common kind of facts. For example, one template captures facts with the structure “method *X* throws an exception of type *Y* when the argument is *Z*.” A template includes *placeholders* that provide the flexibility to use the template in different contexts.

Users *invoke* a template to instantiate a specific fact about a method. A *template invocation* provides the values for the placeholders of the template for a given focal method. Template invocations serve as a common basis for all representations of facts. Figure 1 summarizes the representations supported by DScript and the relations between them. In practice, template invocations are employed to produce two commonly-used representations of facts: header comments used for *documentation* (Javadoc) and *unit tests* using the JUnit framework.

TABLE 1
Technical and implementation challenges involved in the development of DScript. The last column indicates the section of this article that discusses the challenge.

Component	Challenge	Sect.
<i>Technical</i>		
Templates	Capturing kinds of facts	3.1
Invocations	Capturing minimal information	3.2
Invoc. → Tests	Ensuring compilability	4.1
Invoc. → Doc.	Reducing clutter	4.3
<i>Implementation</i>		
Templates	Serializing template information	3.1
JSON	Designing a lossless readable format	3.2
Invoc. → Tests	Proper code style and integration with existing tests	4.2
Invoc. → Doc.	Traceability and integration with source code	4.4

This synergistic combination of automatically generated tests and documentation mitigates the respective weaknesses of both representations of information: The documentation is made checkable and traceable to source code (via its connection to unit tests), and the latent documentation captured by unit tests is made explicit and easily accessible (as documentation). A further benefit of DScript’s approach is that, once generated, the tests and documentation are well-formed artifacts fully independent from the generation framework.

Our research into the development of DScript required solving a number of design challenges, but also experimenting with alternative solutions to implementation challenges. Table 1 summarizes these challenges.

Defining the structure of templates and invocations were the two first challenges. The guiding principle behind their design was to facilitate the generation of tests and documentation while avoiding unnecessary or redundant information. Because these two components were novel aspects of DScript, they also involved significant implementation challenges.

Invocations and their JSON representations are roughly equivalent, so the bidirectional transition between them is straightforward. Given a template invocation, the generation of unit tests is, for the most part, an implementation rather than a design challenge. Nevertheless, ensuring that the generated unit tests are compilable is not trivial, and requires the definition of types for placeholders. Integrating the generated tests and documentation with other artifacts of the system is also challenging, especially with the constraint to minimize repetitiveness of the generated documentation. The solution to this latter challenge led to the design of a novel intermediate representation for documentation.

Because implementation challenges are less relevant to the research, we only mention them briefly for completeness. Our publicly available implementation of DScript provides a fully developed prototype solution to these implementation challenges.

A key design principle of DScript was to avoid any possibility for imprecise inference. Past research has proposed various inference techniques to extract and generate information, often using one representation to generate another [1]. These techniques are useful to *discover* information initially unavailable to developers. However, in the context of our research goal, their limitation is that they require developers

```

/** $method$ throws an exception of type $ex$
 * when $state$.
 */
@Template("Example")
@Types($ex$=EXCEPTION, $state$=EXPR, $factory$=METHOD)
@Test
public void test$method$_$state$() {
    $class$ instance = $factory$();
    try {
        instance.$method$();
        fail();
    } catch ($ex$ e) {}
}

```

Fig. 2. Example of a template.

to validate the outcome of the inference process. In contrast, DScibe’s aim is to effectively leverage information that has already been specified by developers precisely and unambiguously.

3 Templates and Invocations

Templates and their invocations are the main innovations of DScibe. They are a new form of documentation for software systems better suited to represent common facts without repetitiveness and unnecessary information. To achieve this objective, templates encapsulate as much as possible the information that would otherwise be repeated between methods and their associated artifacts (documentation and test), and template invocations encapsulate the remaining information that is method-specific.

3.1 Template Definition Language

DScibe’s templates are, literally, templates for tests and documentation: a DScibe template contains a partial abstract syntax tree (AST) of a unit test and a partial natural language description. These two elements are only partial because they contain placeholders, each identified by a unique name. Thus, a template is exactly the aggregation of a list of placeholders, an AST rooted at a method declaration, and a natural language description.

Figure 2 shows an example of a template. The Java code defines the template AST and the header comment is the template description, with placeholders identified by a surrounding pair of dollar signs (\$). The template expresses the fact that a method throws an exception when its implicit argument is in a specific state. The specific method, exception type, and state are all placeholders of the template, as well as the declaring type of the method and the name of a factory method.

The value of a template (i.e., the information that it captures) is more than the sum of its parts (partial AST and description). A template explicitly associates two representations of a specific kind of facts: how to describe the fact in documentation, and how to test it. It is this association that allows DScibe to transform different representations of the same information, without relying on inference techniques.

For a template to be effective, its AST and description must be self-contained, but also flexible. A developer should be able to understand the purpose of the template by reading the description and the AST, but also to apply this template to various contexts, or methods. A catalog of templates can

thus serve not only to generate tests and documentation, but also as a knowledge base for the development community.

An important design principle for templates was to avoid any reliance on a prescribed coding style, except for the requirement that each unit test focuses on a single test case about a single method. For example, with our design, templates can enforce any convention for unit test names.

Implementation Decisions

Templates are collected in a catalog that consists of a set of parsable Java files. Each method declared in these files and identified with the `@Template` annotation corresponds to a template, as in Figure 2. The AST of the method declaration becomes the AST of the template, and the header comment of the method becomes its description. Each placeholder is a legal Java identifier that begins and ends with a dollar sign (\$).¹ The `@Types` annotation declares the list of placeholders,² except for a few predefined placeholders that refer to the properties of the focal method, such as `$method$` (its name) and `$class$` (its declaring type). Template authors attribute the template’s name as the only argument of the `@Template` annotation (e.g., `Example` in Figure 2).

The motivation for expressing templates using legal Java code was for the template format to be familiar to Java developers. This makes templates more readable, and consequently the knowledge they capture more accessible. The format also allows template authors to leverage their usual tools to create and edit templates. Finally, this format facilitates the creation of new templates from existing tests and documentation: a developer only needs to clone the existing test and documentation, and replace specific values with placeholders.

3.2 Template Invocations

A template invocation records the application of a template to a focal method. Invocations require an *invocation context* that consists of the signature of the focal method and values for placeholders. The method signature is required to correctly link the generated assets (tests and documentation). It also provides the values of the few predefined placeholders (e.g., `$method$`), and a default package and Java type from which other placeholder values can be resolved.

The remainder of the invocation context is the set of values to assign to the template’s placeholders. Following the principle that the generation of tests and documentation should be as transparent for the user as possible, DScibe replaces placeholders with the user-provided values with as little transformation as possible to make the test compile or the documentation sensible. Thus, the values supplied to the template invocation are not expressions to be evaluated by the generation engine, but expressions to be substituted verbatim for the placeholders.

1. Although it is a legal character for identifiers, the Java Language Specification discourages the use of dollar signs in usual code [2, §3.8], thus reducing the probability of collisions between templates and actual code.

2. It also assigns a type to each placeholder. Placeholder types are designed mostly for unit test generation, so we discuss them in Section 4.1.

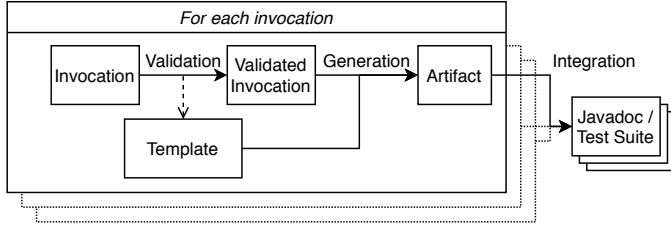


Fig. 3. High-level overview of tests and documentation generation

Implementation Decisions

We made the arbitrary decision to format invocation files using JSON. This format allows users to directly read, write, and edit invocation files with any text editor. Users can also use many existing tools to manipulate JSON objects more effectively. Finally, many JSON libraries support the implementation of a straightforward serialization and deserialization of invocations.

To keep invocation files as concise as possible, they contain only the necessary information. Hence, in contrast to template files, which are mostly self-contained, invocation files are not. They depend on the definition of the relevant templates to generate useful information. Future versions of DScript could include tools to make invocation files easier to create, read, and edit, but this implementation challenge is left for future work.

4 Unit Test and Documentation Generation

The ultimate goal of DScript is to manage external fact representations: unit tests and Javadoc documentation. The generation of both kinds of artifacts follows a similar three-step process summarized in Figure 3.

For each individual invocation, the first step is to identify the template being invoked and validate the values of the invocation. The validation step ensures that the invocation refers to a valid template, that its focal method exists, and that each placeholder value can be correctly substituted. Validating placeholder values is especially important, and challenging, for the generation of compilable unit tests. This challenge motivated the design of a small type system for placeholders that can guide developers in properly invoking templates (see Section 4.1).

If the invocation passes the validation step, the second step is to generate the artifact (i.e., unit tests or documentation fragment) by substituting each placeholder with the value that the invocation provides.

After generating all artifacts individually, the third and final step is to combine them and integrate them with the rest of the system. Generated unit tests are integrated within the existing test suite, and generated documentation is integrated directly in the source code as header comments for documentation. In addition to inserting the generated artifacts into the project without disrupting the rest of the system, the integration step is also responsible for removing outdated generated artifacts, and aggregating similar generated artifacts to reduce repetitiveness. This aggregation is especially important for documentation, because unnecessary clutter will have a negative impact on readability. To solve this aggregation challenge, we designed a novel, easily interpretable structured format

```

$type$ x = $expr$;
Object y = x.$method$($exprlist$);
System.out.println(y.$field$);
throw new $exception$();

```

Fig. 4. Example of a partial AST template with placeholders of different types

to express facts, which also allows for a trivial yet effective aggregation of similar facts.

4.1 Placeholder Types

During the generation of tests, the value that each placeholder replaces is subject to different syntactic rules, depending on the location in which the placeholder appears in the template. For example, in the assignment `Object x = $p1$($p2$)`, the first placeholder, `$p1$`, can never be replaced by an integer literal. Placeholder types can help avoid such errors. Each different type defines a specific set of rules that apply to a placeholder based on its location in the template.

The placeholder types DScript supports are `TYPE`, `EXCEPTION`, `METHOD`, `FIELD`, `EXPR`, and `EXPR_LIST`. Figure 4 shows usage examples of a placeholder of each type.

Placeholder values of type `TYPE` and `EXCEPTION` must be the qualified name of an existing Java type in the build path of the system. Additionally, `EXCEPTION` placeholders must inherit from the `Throwable` class. A qualified name is necessary to resolve the Java type, but also to insert it in the template without an associated `import` statement. However, to reduce unnecessary effort, if the Java type is declared in the same package as that of the focal method, only the simple name is required. Placeholders of type `METHOD` and `FIELD` replace a method or field name, respectively. Placeholder values of type `EXPR` must be syntactically legal Java expressions. Similarly, the `EXPR_LIST` type can be used for placeholders that replace a variable number of expressions, usually for the arguments of a method invocation in the template (see the placeholder `$exprlist$` in Figure 4).

For the types `METHOD`, `FIELD`, `EXPR`, and `EXPR_LIST`, DScript does not resolve the identifiers used in the placeholder values, and so it does not verify that the method or field exists, or that the expressions refer to existing variables. Therefore, it is possible that DScript will generate unit tests with compilation errors due to unresolved symbols or incompatible types (e.g., if the value of `$expr$` is incompatible with the Java type `$type$` in Figure 4). This limitation is a necessary condition to allow developers to create templates reusable in various contexts, and it is mitigated by the fact that the compiler of the test suite will detect these errors.

The context of the invocation provides a few predefined placeholders, including `$method$`, `$class$`, and `$package$` for the focal method (or constructor) and its declaring type and package, respectively. These placeholders do not require an explicit value from the user, as the value is derived from the context. Therefore, they do not have a placeholder type.

4.2 Integration of Unit Tests with Existing Test Suite

Integrating the generated unit tests with an existing test suite is mostly an implementation challenge. Each template defines, using placeholders, the name of the class and package where to place the generated test. Thus, DScript groups together all

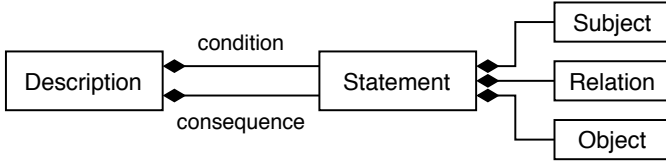


Fig. 5. UML description of the structure of documentation fragments

tests that go in the same file, and writes one file for each such test class. Because each unit test is independent, and because a large number of similar tests are not an issue for test suites (they are meant to be read by the testing framework), there is no further aggregation to perform.

To avoid corrupting the manually written testing code, but also remove outdated tests generated by previous executions, DScript places all generated tests in a separate folder, defined by the user. Therefore, manually and automatically generated test code can coexist in separate folders, and template invocations can leverage scaffolding, such as stub objects and helper methods, from the main test suite. When a user executes DScript again, old test classes (in DScript’s folder) are simply overwritten by the new ones, thus removing any outdated test.

4.3 Information Aggregation

Some methods can have multiple similar facts that need to be tested and documented. For example, Java’s `Math.log(double)` method returns the natural logarithm of its argument. However, the natural logarithm, as a real-valued function, is only mathematically defined for positive numbers, so the method needs to define special behaviors for other values. In particular, `Math.log` returns NaN for negative numbers and NaN itself. Thus, a DScript user would create two separate invocations for these two special cases, and DScript would then generate two similar documentation fragments, e.g., “If *a* is NaN, the result is NaN” and “If *a* is negative, the result is NaN”.

Because DScript inserts the documentation fragments directly into the header comment (Javadoc) of the focal method, such repetitive fragments are undesirable. Instead, it is preferable to use a single sentence to express both cases, as currently found in the documentation: “If the argument is NaN or less than zero, then the result is NaN” [3].

To support the aggregation of similar fragments, we designed a structure to express facts about method behavior. Figure 5 summarizes this structure. Each documentation fragment is divided into two statements, a *condition* and a *consequence*, with the interpretation that if the condition is true, then the consequence must also be true. For example, for `Math.log`, the condition “*a* is negative” is associated with the consequence “the result is NaN”. Furthermore, each statement is itself composed of three parts, a *subject*, a *relation*, and an *object*, similarly to Resource Description Framework (RDF) triples [4], with the interpretation that the relation applies from the subject to the object. Thus, the condition of the previous example would have “*a*” as subject, “is” as relation, and “negative” as object.

Altogether, this structure requires template authors to divide the natural language description associated with the template into six parts (i.e., the subject, relation, and object for both the condition and consequence) which map naturally

to most specifications. Each part can be any text (with placeholders), to allow as much flexibility as possible. Using this structure, DScript can aggregate similar fragments without the need for natural language processing techniques. If two or more fragments share the same condition (resp. consequence), DScript aggregates the consequences (resp. condition) of those fragments. When aggregating statements, DScript can also avoid the repetition of a common subject, relation, and/or object.

In our example, DScript would initially generate two fragments, “If *a* is NaN, then the result is NaN” and “If *a* is negative, then the result is NaN”. These two fragments have the same consequence (“the result is NaN”), so DScript will aggregate the conditions. Both conditions share the same subject (“*a*”) and relation (“is”), so the aggregated condition will become “*a* is NaN or negative”. Thus, both fragments become the single fragment “If *a* is NaN or negative, then the result is NaN”.

Although the six-part structure naturally maps to most method behaviors, to account for cases that are impossible to express with this structure, either or both statements can be replaced with free form text, or the complete fragment can be free-formed. Using these special provisions, however, limits the ability of DScript to aggregate similar fragments.

4.4 Integration of Documentation with Code

DScript inserts the generated documentation fragments into the Javadoc header comment of the focal method, creating the header comment if necessary. Thus, the documentation is explicitly linked to the code element (i.e., method) it applies to. This is made possible by the explicit link between a template (which captures the documented information) and its focal method.

DScript uses the custom `@dscript` Javadoc tag to mark generated fragments from those previously written by developers. Each (aggregated) fragment is prefixed by its own tag to avoid a single, large paragraph that is hard to read.

The custom `@dscript` tags not only clearly indicate to the user which statements are automatically generated, and backed by unit tests, it also allows DScript to keep the documentation up-to-date: when the user modifies the template invocations, DScript can remove all previous `@dscript` tags and regenerate them with the new information. This process does not impact the manually written documentation at all, unless users manually modify the content of generated tags.

5 Overview of the Empirical Assessment

Leveraging our implementation of DScript, we conducted a multi-pronged empirical investigation of key aspects relating to the generation of unit tests for documentation. The investigation sought to answer the following research questions.

- RQ1 To what extent is information in source code, unit tests, and documentation inconsistent?
- RQ2 To what extent can we leverage DScript templates to automatically test and document behaviors of focal methods?

In this investigation, we followed a three-stage process, with each stage constituting a cohesive study of its own:

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 DScript's ability to prevent inconsistencies	1, 2
Validation	7	Commons Math, Lang, Config.	Tested specifications	Validate the results of the usefulness study	1
Limitations	8	5 open source projects	All	Understand DScript's limitations	2

- 1) **Usefulness Study:** We conducted an *in-depth* study of the usefulness of DScript in a *narrow context* (Section 6);
- 2) **Validation Study:** We conducted a *multi-case* study to validate the findings of the first study in a *broader context* (Section 7);
- 3) **Limitations Study:** It is customary to discuss the limitations of proposed software engineering techniques. In our case we also conducted an empirical study to better understand the limitations of DScript in diverse scenarios (Section 8).

Table 2 provides an overview of the empirical work described in this article. For all three studies, the complete and detailed results are publicly available in our on-line appendix.

6 Usefulness Study

We investigated the usefulness of DScript to prevent inconsistencies. The objective of the investigation was twofold. First, it aimed at understanding the nature of the problem of information repetitiveness and redundancy. Second, it aimed at assessing the potential of DScript to avoid future inconsistencies by automating the generation of repetitive and redundant information.

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 the general case it requires a significant amount of manual effort to isolate and fully understand even just a few units of specifications in unfamiliar code. To make this case study tractable, we narrowed the scope to a particular type of unit of specification that is well-defined: units of specification about exceptions, which are relevant in almost all systems. 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 expression 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 chose the Apache Commons IO library (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. 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

TABLE 3
Number of methods, exception specification units (ESUs), and instantiated DScript 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

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 ESUs and templates identified 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 DScript 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.

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 described in the documentation (In Doc.) and tested by 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

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

TABLE 5

Number of times that each DDescribe template was instantiated

Template	Invocations	Not Invoked	
	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%)

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 DDescribe 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”. 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 DDescribe.

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 DDescribe’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 DDescribe’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. DDescribe helps standardize the consistent use of recurrent templates, thus enhancing the quality of test suites.

We were not able to instantiate ESUs in only 25 cases (3%), due to two main reasons. The first one was the presence of inaccurate ESUs in the documentation, i.e., statements in the documentation that 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. We did not instantiate the other ESUs because 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. The majority of these cases were also not tested in the test suite.

6.3 Threats to Validity

Two of the authors performed all annotations. 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. Additionally, the methods of utility libraries are usually self-contained and can be understood without knowledge of the system as a whole. Nevertheless, the annotations may still reflect the investigators’ experience. 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 investigate 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. Nevertheless, the study demonstrates the usefulness of DDescribe 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.

7 Validation Study

The results presented in Section 6 clearly indicate that information inconsistency across source code, documentation, and unit tests is a clear issue for exception handling in the Commons IO project. To refine the answer to RQ1, we performed

TABLE 6

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

a multi-case study to validate and expand the findings of the initial case study.

7.1 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 chose to again study Apache Commons projects for the same reasons outlined in Section 6. 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* [5], 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 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 if 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.

7.2 Results and Discussion

Table 6 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.*).

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 6 should thus be regarded as lower bounds of the effective values.

The investigation of the sampled tests also revealed interesting use cases for DScribe outside the scope of this study. For example, the documentation of `StrBuilder.asTokenizer()`, from the Lang project, contains a usage example that is very similar to the test `StrBuilderTest.testAsTokenizer()`. In such situation, developers could also use DScribe to generate usage examples for which the correctness is guaranteed by passing unit tests. This study also revealed the presence of incorrect documentation, such as that of the method `Dfp.reciprocal()`. Its unit tests, however, capture the correct behavior. By generating unit tests and documentation together, DScribe reduces the amount of brittle documentation that silently becomes inaccurate.

7.3 Threats to Validity

As for the usefulness study, an 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. 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 are aware of 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.

TABLE 7
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

8 Limitations Study

The usefulness study showed evidence of the potential effectiveness of DScript in one particular context, in which 97% of the identified exception specification units could be captured by template invocations. However, to better answer RQ2 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.

8.1 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 7 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 [6] 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.

8.2 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 DScript in different contexts, they also revealed simple strategies to work around these limitations, which can improve the quality of the generated tests. Furthermore, these limitations can provide insights about new features to add to DScript in future work.

Generic Variable Names: A template-based approach requires the use of recurrent, generic names for local variables in unit tests (e.g., `input` and `expected`), as opposed to names specific to the test context (e.g., `baseString`, `encodedString`). Although only a minority of the studied tests used such generic identifiers, we believe generic names can have a beneficial impact on the readability of the test suite, as it allows unfamiliar readers to understand new tests quickly by identifying recurrent important aspects. Thus, despite being a limitation of template-based approach, this factor can be beneficial in the long term.

Structured Test Names: An important strength of a template-based approach is the ability to standardize and facilitate the use of conventions. In particular, although the name of unit tests does not impact its behavior, it is considered good practice to use meaningful names, 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)?`.

Recurrent Complex Operations: A common limitation of template-based approaches is the diverging implementations of similar operations. For example, many tests verify that the content of a generated object matches that of the expected result, but the implementation of this verification depends on the structure of the objects. Thus, although the tests follow the same high-level patterns, they cannot be generated from the same template. However, we observed that some tests encapsulated such recurrent complex operations into helper methods with generic but meaningful names, such as `assertContentEqual`. The use of such helper methods can mitigate this limitation, and increase the readability of test suites. Nevertheless, relying excessively on helper methods can be detrimental. As an extreme, but not unique, example, Tomcat's helper method `TestELParser.doTestParser` encapsulates all operations of multiple tests. This leads to a very complex logic that is harder to write and read than if the different tests were decoupled, and this helper method can only be used for testing a single class.

Complex Assertions: Some tests require complex assertion structures. For example, testing methods that rely on inversion of control may need to nest assertions inside mock objects, and to call seemingly unrelated methods to trigger the assertions. Such structures can severely limit the applicability of templates, and thus the usefulness of a template-based approach. Thus, these complex cases remain mostly outside the scope of DScript, or similar approaches. However, if the

same pattern of complex assertion is often needed, a template-based approach can encourage developers to create the necessary scaffolding and have a more systematic approach to test complex behaviors.

Testing Preconditions: Several tests include assertions to verify the input state of tested objects before the method under test is performed.³ Although there is no reason a template could not include these early assertions, in many cases, the assertions are specific to the tested objects and relevant to only some test case, so they are not well suited for template-based generation. A simple workaround this limitation, however, is the use of factory methods to create the tested objects in the right input state, and move any necessary early assertion to these methods.

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 need to perform additional setup and cleanup operations to avoid corrupting the resources, as well as special precautions to control errors originating from the constrained resources themselves (e.g., trying a second time to connect to a server if the first time fails). These operations often create deviations from recurrent templates, thus multiplying the number of required patterns to account for each possible deviation. Different mitigation strategies can limit the negative impact of these operations, such as an efficient use of “setup” and “teardown” methods (using JUnit’s `@Before` and `@After` annotations), but the right choice depend on the nature of the constrained resource.

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. Currently, DScript assumes that the focus of each generated test is a method, so the generation of these other tests would be outside its scope. However, this is only a design decision for the prototype, and extensions of DScript could include other types of units under test.

Variety of Test Purpose: Tests in a test suite serve various purposes. Some simply test the usual behavior of a unit with specific examples, whereas others focus on exceptional behaviors or corner cases. Although it is important to test the former cases, it is likely that only the latter cases will need to be documented. Thus, a template-based approach should be able to generate documentation for only some templates to avoid clutter. More importantly, some tests verify the integration of various components in more complex scenarios (i.e., integration tests, which are not technically unit tests, but can still be found in the same test suites), and others are specifically tailored to a specific bug or case of regression. These kinds of tests are clearly outside the scope of template-based approaches, as they each require the execution of a specialized sequence of actions.

3. This practice is debated among developers, with some arguing that preconditions should be the object of separate tests. However, without taking a stance in this debate, given that at least some developers may want this feature, we consider its importance.

8.3 Threats to Validity

The case study relied on the identification of testing patterns, a subjective concept relative to the experience of every developer. Hence, the conclusions may reflect the personal experience of the investigator. This experimental design choice was necessary because the data analysis required a very high initial effort investment to study the systems, and a consistent point of view from one system to the other. Thus, the coding procedure could not be packaged into multiple sets of data to be labeled by independent coders. Furthermore, 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. To mitigate this risk, the on-line appendix contains several concrete examples supporting each of our conclusions.

9 Related Work

The difficulty of maintaining high-quality documentation [7], [8], [9], [10] led to a vast exploration of **automated documentation generation** approaches. Techniques proposed in prior work involve static [11] and dynamic [12] analysis of the body of methods, as well as their context [13], and different techniques are tuned to document either classes [14], methods [15], [16], or method parameters [11]. Techniques also differ in the kind of documentation they generate, such as specifications [17], [18], [19], program invariants [20], test summaries [21], [22], and usage scenarios and examples [23], [24]. However, a common limitation of such *fully*-automated techniques is that the correctness and usefulness of the generated documentation is limited by the underlying heuristics and the information these heuristics rely on. Furthermore, some manual selection is ultimately required as the unconstrained generation of large amounts of information can end up diluting important insights within more trivial information. Being semi-automated, DScript leverages developer effort, rather than replacing it, keeping developers 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 [25] and DART [26], 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. [27] generate test cases by extracting sequence of method calls to create relevant input states. Pacheco et al. [28] 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 [29] follow a more systematic random generation approach by leveraging mutation operators, and using genetic algorithms to optimize the test suite. Taneja and Xie [30] 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 [31], that apply to multi-threaded code [32], or that map to the system’s UML diagrams [33]. However, automated test generation techniques suffer from a similar problem as documentation generation techniques: In order to completely remove developers from the generation process, the techniques are susceptible to false positives, which in turn require human effort to filter out. In contrast, DScript

involves developers in the generation process so that no single person is required to sift through a large output after the generation.

The value proposition of DDescribe, however, extends beyond the generation of tests and documentation. An important benefit is the **automated traceability links** of the generated artifacts to the method they complement. Documentation traceability is a challenging problem [34], [35], but it is a prerequisite to validate the correctness of the documentation, another challenging problem [36], [37]. DDescribe offers a way to solve both problems by linking documentation not only to its focal method, but also to a unit test, such that if a change in the behavior of the method happens, the failing unit test will also flag the specific fragment of documentation as incorrect. This solution is similar to that of behavior-driven development (BDD) [38], a methodology derived from test-driven development [39]. BDD recognizes the documentation potential of testing code, and BDD frameworks such as JBehave [40] offer a way to integrate documentation fragments directly into unit tests, so that documentation is again backed by passing tests. However, developers are still responsible for writing both the testing code and documentation fragments, a repetitive and redundant effort.

Finally, our research is related to that 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 [23], [41], [42], and in particular violations of these implicit specifications, or design patterns [43], [44]. Future work can leverage a similar approach to automatically generate DDescribe templates, to further reduce the initial burden of developers.

10 Conclusion

Motivated by the observation that documentation and testing code often capture redundant and repetitive information, we designed a technique, called DDescribe, 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, while also encouraging the use of collectively agreed upon templates to reduce unnecessary variability in these artifacts.

A three-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. In addition, the investigation revealed that DDescribe could have prevented 97% of these inconsistencies in a favorable context. Finally, our empirical assessment of DDescribe includes rich descriptions of the technical characteristics of software projects that facilitate or hinder the application of DDescribe, thus providing insights on the potential costs and benefits of introducing the technique in different contexts.

Acknowledgments

This work was funded by NSERC.

References

- [1] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.
- [2] 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>
- [3] Oracle. (2018) Math (Java SE 11 & JDK 11). [Online]. Available: <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>
- [4] 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/>
- [5] B. Van Rompaey and S. Demeyer, "Establishing Traceability Links between Unit Test Cases and Units under Test," in *13th IEEE European Conference on Software Maintenance and Reengineering*, 2009, pp. 209–218.
- [6] M. B. Miles, A. M. Huberman, and J. Saldana, *Qualitative data analysis*. Sage, 2013.
- [7] T. C. Lethbridge, J. Singer, and A. Forward, "How Software Engineers Use Documentation: The State of the Practice," *IEEE Software*, vol. 20, no. 6, pp. 35–39, 2003.
- [8] B. Fluri, M. Wüsch, and H. C. Gall, "Do code and comments co-evolve? On the relation between source code and comment changes," in *Proceedings of the Working Conference on Reverse Engineering*, 2007, pp. 70–79.
- [9] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*iComment: Bugs or Bad Comments?*/," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2007, pp. 145–158.
- [10] I. K. Ratol and M. P. Robillard, "Detecting Fragile Comments," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 112–122.
- [11] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *Proceedings of the IEEE International Conference on Program Comprehension*, 2011, pp. 71–80.
- [12] M. Sulír and J. Porubán, "Generating Method Documentation Using Concrete Values from Executions," in *Symposium on Languages, Applications and Technologies*, 2017, pp. 3:1–3:13.
- [13] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 279–290.
- [14] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 23–32.
- [15] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 43–52.
- [16] 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 *IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 561–565.
- [17] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 4–16.
- [18] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia, "Static Specification Mining Using Automata-Based Abstractions," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 651–666, 2008.
- [19] C. Le Goues and W. Weimer, "Specification mining with few false positives," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 292–306.
- [20] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin, "Dynamically discovering pointer-based program invariants," in *Proceedings of the International Conference on Software Engineering*, vol. 373, 1999.
- [21] 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 *Proceedings of the 38th*

- International Conference on Software Engineering*, 2016, pp. 547–558.
- [22] B. Zhang, E. Hill, and J. Clause, “Towards automatically generating descriptive names for unit tests,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 625–636.
- [23] M. Acharya, T. Xie, J. Pei, and J. Xu, “Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications,” in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007, pp. 25–34.
- [24] R. P. Buse and W. Weimer, “Synthesizing API usage examples,” in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 782–792.
- [25] K. Sen, D. Marinov, and G. Agha, “CUTE: A concolic unit testing engine for c,” in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, p. 263–272.
- [26] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, p. 213–223.
- [27] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, “Mseegen: Object-oriented unit-test generation via mining source code,” in *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering*, 2009, pp. 193–202.
- [28] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 75–84.
- [29] G. Fraser and A. Zeller, “Mutation-driven generation of unit tests and oracles,” *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2012.
- [30] K. Taneja and T. Xie, “Diffgen: Automated regression unit-test generation,” in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 407–410.
- [31] C. Csallner and Y. Smaragdakis, “JCrasher: an automatic robustness tester for java,” *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.
- [32] 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 *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 727–737.
- [33] J. Offutt and A. Abdurazik, “Generating Tests from UML Specifications,” in *UML’99 – The Unified Modeling Language*, 1999, pp. 416–429.
- [34] A. Marcus and J. I. Maletic, “Recovering documentation-to-source-code traceability links using latent semantic indexing,” in *Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 125–135.
- [35] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, “Recovering traceability links between code and documentation,” *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.
- [36] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, “Analyzing APIs Documentation and Code to Detect Directive Defects,” in *IEEE/ACM International Conference on Software Engineering*, 2017, pp. 27–37.
- [37] E. Ben Charrada, A. Koziolk, and M. Glinz, “Identifying outdated requirements based on source code changes,” in *IEEE International Requirements Engineering Conference*, 2012, pp. 61–70.
- [38] M. Soeken, R. Wille, and R. Drechsler, “Assisted behavior driven development using natural language processing,” in *Proceedings of the International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 2012, pp. 269–287.
- [39] K. Beck, *Test-driven development: by example*. Addison-Wesley, 2003.
- [40] JBehave.org. (2017) What is JBehave? [Online]. Available: <https://jbehave.org/>
- [41] M. Allamanis and C. Sutton, “Mining idioms from source code,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 472–483.
- [42] Z. Li and Y. Zhou, “PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code,” in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005, pp. 306–315.
- [43] J. Dong, Y. Zhao, and T. Peng, “A Review of Design Pattern Mining Techniques,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 06, pp. 823–855, 2009.
- [44] A. Pandel, M. Gupta, and A. Tripathi, “DNIT – A new approach for design pattern detection,” in *Proceedings of the International Conference on Computer and Communication Technology*, 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 incoming M.Sc. student in Computer Science at McGill University. Her research interests include software design, maintenance, and evolution. Alexa received a B.A. in Computer Science at McGill University, where she worked under the supervision of Martin P. Robillard.



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.