2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

Constructural Software Documentation

Mathieu Nassif and Martin P. Robillard
School of Computer Science
McGill University
Montréal, Canada
{mnassif, martin}@cs.mcgill.ca

***Abstract*—Software projects capture information in artifacts that include production code, test suites, and documentation. Because different artifacts serve different purposes, some artifacts can include redundant information, encoded in different formats. To mitigate this redundancy, we propose an approach to explicitly encode in unit tests information that will be automatically extracted and added to the documentation of the software. We implemented this approach in the form of an Eclipse plug-in that binds unit tests written with JUnit to the header comments of the tested methods.**

***Index Terms*—Constructural documentation, Unit testing, Automatic documentation, Documentation extraction, Javadoc**

I. Introduction

Writing tests and documentation often involves encoding information that is already present in production code, but in a different format. For example, consider Google Guava [1]'s `Optional` class, which either holds a non-null reference, or is empty (the reference is *absent*). An associated testing class, `OptionalTest`, contains the unit test `testGet_absent`, shown in Figure 1, which tests the method `get()`. This unit test conforms to many testing conventions: the last segment of the test name, `absent`, identifies the state of the instance under test, and the `try-catch` statement is a testing pattern to ensure that a method throws a specific type (or subtype) of exception.

```java
public void testGet_absent() {
  Optional<String> optional = Optional.absent();
  try {
    optional.get();
    fail();
  } catch (IllegalStateException expected) {
  }
}
```

Fig. 1. Example of a unit test from Google Guava

Simply by looking at this test, a reader can learn that when `get()` is called on an `Optional` instance that is `absent`, an `IllegalStateException` is thrown. This information is documented in the header comment of `get()`: *@throws IllegalStateException if the instance is absent [...]*

While writing the same information in two different formats is redundant effort, it is necessary to support those using the API without access to the production or testing code. Yet, the manually-written constraint is brittle: if the behavior of the method changes, the modification will be detected by the failing test, but the documentation can silently become inconsistent. Hence, redundant information represents an additional burden on two levels: when creating the redundant artifacts, and when maintaining them.

To alleviate the burden due to redundant information in documentation, we propose an approach to generate documentation based on some explicit structure in unit tests. The approach leverages well-defined constructs, such as conventions for test names as seen in the example above. This approach both provides a framework for developers to actively embed information by using these specific constructs, and supports the generation of human-readable documentation. Thus, it integrates writing tests and documentation into a single, streamlined approach: a combination we call *constructural documentation*. By removing the need to write low-level information that is already captured by unit tests, constructural documentation can allow documentation writers to focus their effort on less trivial information, such as high-level abstractions or architectural designs.

Unlike many other documentation generation approaches [2]–[6], our approach does not infer information from already written code. In contrast, it provides a way for developers to explicitly incorporate information using agreed-upon constructs, an idea closer to literate programming [7], but at a strictly verifiable level. Some behavior-driven development [8] frameworks, such as JBehave, achieve a similar result by inserting documentation elements in testing code, but none of them directly identify arbitrary code constructs as a form of documentation. With our approach, we introduce the idea of constructural documentation in the context of unit tests and API documentation. We believe this idea should be explored further, and our work aims to be a first step in this direction, outlining the potential of this idea.

II. Preliminary Study

To understand the potential of a constructural documentation approach in different contexts, we performed a multi-case study on the testing code of five Java software systems from different domains and organizations: Freemind, Eclipse Platform UI, Weka, Apache Tomcat, and Hibernate ORM.

This study aimed at finding the technical factors that act as enablers or obstacles for embedding constructural documentation in unit tests, in order to better estimate the benefits or drawbacks of using our approach in different scenarios. We focused on *technical* aspects, i.e., properties of the code of a snapshot of the systems. We left other factors, such as the development process, out of the scope of this study.

The results of our study show several factors contributing to or hindering constructural documentation. For example, we found that recurring local variable names, with a more general meaning, are more informative in the context of constructural documentation than very specific names. In contrast, code that diverges from the well-established structures, and thus hinders constructural documentation, is often found in tests relying on constrained resources, such as a connection to an external server. A more ambiguous finding was the use of helper methods: in some contexts, they allowed developers to hide complex assertion structures, while in other contexts, too much information was hidden, negatively impacting what information could be extracted from the unit test.

III. Overview of the Approach

The development of our approach was guided by the following two principles. First, the documentation generated from unit tests should not be inferred. Rather, it must be explicitly embedded by developers. This ensures that the generation process is transparent, thus easy to control for false positives or negatives, and that the generated documentation can be traced back to its source. Furthermore, a transparent process can also motivate test writers to use a more consistent coding style. The second design principle is for the approach to be usable with any coding style. The approach should avoid prescribing a fixed style that would not match the conventions of a team or become a burden in the evolution of the system when new conventions are adopted.

As part of our approach, we introduce a new terminology to discuss constructural documentation: A **shape** represents a well-defined, parameterized structure of code, and a **shape instance** is its instantiation in a particular fragment of code. For example, in Figure 1, the `try-catch` statement, with an empty `catch` block, is an instance of a common shape to verify that a statement throws an exception. Just like geometric shapes can be assembled to form meaningful figures, code shapes can be assembled into a **construct**, which is a set of one or more shapes that represents a self-contained meaning. In Figure 1 shows a construct composed of the exception-handling shape mentioned above combined with the another shape instantiated in the test's name, that indicates when an exception can be thrown by `Optional.get`. This information, embedded in source code constructs, is called **constructural information**. Thus, **constructural documentation** is both the action of embedding constructural information in source code and its result, i.e., the documentation generated from this information.

Given our design principles for constructural documentation, our approach requires an initial configuration that defines the different shapes and constructs to target. This configuration encodes the different conventions prescribed by the development team, and their meaning. Once configured, a tool scans all unit tests of a test suite to identify the shapes present in each test, as well as the focal methods [9] of the tests. If possible, the shapes are assembled into one or more constructs, each of which is associated with a function to generate an *information fragment* (whose format depend on the implementation of the approach), based on the shape instances captured. So far, this first phase generates a set of information fragments to be associated with different methods in production code. Finally, the information fragments are grouped by focal method and aggregated together, then synthesized into proper documentation fragments to inject into the documentation.

IV. Proof of Concept: DScribe

To validate the potential of our approach, we implemented it in the form of an Eclipse plug-in, called DScribe, which targets unit tests written with the JUnit framework. DScribe generates human-readable documentation injected directly in the header comments of the production code methods. This implementation also explores some of the implementation tradeoffs of constructural documentation.

Readability was a main priority that guided our implementation. Hence, we designed DScribe to avoid generating clutter in the documentation by grouping similar constraints. This goal led to the following format of the information fragments: they consist of combinations of predicates, themselves subdivided into objects and conditions, so that similar parts can be easily assembled. Finally, DScribe is configured using a single text file and simple patterns to define shapes and functions to generate the information fragments.

We tested our implementation on two test classes of Weka. We manually refactored the unit tests to apply a consistent and explicit style, and wrote a 72-line configuration file. DScribe was able to analyze the unit tests almost instantaneously, and produce documentation fragments such as *After calling this method, the content of the object is the same as array*, where *array* is the name of the parameter of the method.

References

[1] Google Inc. Guava: Google Core Libraries for Java. [Online]. Available: https://github.com/google/guava

[2] M. Sulír and J. Porubän, "Generating Method Documentation Using Concrete Values from Executions," in *Symposium on Languages, Applications and Technologies*, 2017, pp. 3:1–3:13.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[8] 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.

[9] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *IEEE 15th International Working Conference on Source Code Analysis and Manipulation*, 2015, pp. 61–70.