The Future of Library Specification

Gary T. Leavens

CS-TR-10-09
August 2010

Keywords: Library, framework, specification, documentation.
2010 CR Categories: D.2.1 [*Software Engineering*] Requirements / Specifications — langauges, tools; D.2.2 [*Software Engineering*] Design Tools and Techniques — modules and interfaces, software libraries; D.2.4 [*Software Engineering*] Software / Program Verification — formal methods, programming by contract; D.2.5 [*Software Engineering*] Testing and Debugging — debugging aids, testing tools; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — documentation; D.2.13 [*Software Engineering*] Reusable Software — reusable libraries; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — specification techniques;

Dept. of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, FL 32816-2362 USA

The Future of Library Specification

Gary T. Leavens
University of Central Florida
Orlando, FL, USA
leavens@eecs.ucf.edu

ABSTRACT

Programming language technology has started to achieve one of the dreams of software engineering — large scale utilization of reusable components. This is due to the standardization of large libraries and frameworks in popular programming languages such as C++, Java, C#, and Python. This standardization and widespread use of libraries will continue to make module specification more and more important. Yet most libraries and frameworks are only specified informally using natural language. This position paper explores research questions related to the specification of such libraries and frameworks.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications—*languages, tools*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces, software libraries*; D.2.4 [**Software Engineering**]: Software / Program Verification—*formal methods, programming by contract*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, testing tools*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*documentation*; D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*

General Terms

DOCUMENTATION, VERIFICATION

Keywords

library, framework, specification, documentation

1. INTRODUCTION AND PROBLEM

For a long time I have been interested in the problem of specification language design. In this paper I want to make the case that this problem will be increasingly important in the future, and to explore ways that specification languages might become increasingly useful.

Module specification, which I and others have also called behavioral interface specification, is important and will become more important because of an accelerating trend in programming. This trend is that programming is and will be increasingly dependent on large frameworks and code libraries.[1] This is clear from successful programming languages such as C++, Java, C# (and .NET), Python, and Ruby, all of which provide programmers with very large and extensive libraries. For example, the Java 1.6 Standard Edition platform has 219 packages covering many domains, such as GUIs, I/O, mathematics, collection data structures, remote message invocation, databases (SQL and XML), concurrency control, logging, regular expressions, image processing, cryptography, etc. The increased productivity that results from the use of such libraries helps explain the popularity of these languages. The size and breadth of such libraries will likely increase in the future, as this is one of the principal areas where different languages compete with each other.

A fundamental research problem is thus the following: To what extent do libraries increase software productivity and decrease the total cost of software built using them? Such a study, which I hope someone else will do, could also lead to recommendations for key features that contribute to productivity and other software engineering qualities, such as reliability.

My own interest in this area is based on the assumption that, when programming with a large library, users have two fundamental problems:

1. A learning problem, which is understanding the purpose, architecture, and capabilities of the library, so that one knows which parts of a software system can be profitably delegated to the library.

2. A reference problem, which is quickly finding answers to detailed questions about the library, so that one can find and use a part of the library to correctly accomplish some specific programming task.

To summarize, libraries are a welcome and increasingly important part of programming. An important research question for software engineering is: *What technology can we use to specify libraries in a way that solves the learning*

[1]Hereafter, I will just say "libraries" instead of "frameworks and code libraries."

and reference problems, but which lowers the total cost of software built using these libraries? This same technology might also be useful for documentation of other software entities, but the economic case for reusable code bases such as libraries will be the best one possible, since even small savings during each reuse have the potential to offset the one-time fixed costs of developing the documentation.

2. SOLUTION APPROACHES

There are several candidate technologies for addressing the learning and reference problems, which I discuss below. It is not my intention to dismiss these alternatives or any other technologies from consideration. (Judging between these technologies is a research problem!) Indeed, many, if not most, of these have unique benefits.

2.1 Natural Language Approaches

Currently, the main technology for addressing the learning and reference problems is documentation written in some natural language (such as English). However, the use of natural language as the primary documentation technology causes several problems, all of which relate to the reference problem described above:

- It is often imprecise or ambiguous.
- The use of natural language poses translation problems, which makes it hard for non-fluent speakers to understand precisely and quickly.
- It is hard for automated tools to use natural language descriptions in testing, debugging, or reasoning about programs.[2]

Nevertheless, natural language documentation does have its advantages. It seems inexpensive to produce, since the people writing documentation are often not specially trained. It allows a high degree of freedom in structuring the documentation, which helps with the learning problem. Also, when done well, natural language documentation can summarize behavior at an appropriate level of abstraction. Further, for those fluent in the language, such documentation can be processed very quickly, without much interruption to one's thought process. The ability of integrated development environments, such as Eclipse (or Microsoft's Visual Studio) to bring up a few sentences of documentation summarizing the behavior of a method is often all that is needed by an expert programmer to answer a reference question.

Documentation Comments.

Often natural language documentation is found in slightly structured comments, kept with the source code, as in javadocs. Keeping the documentation with the code helps keep it up to date with code changes. However, being up to date with the code cannot be automatically enforced, as there is no formal connection between the comments and the code.

[2]There is some recent work aimed at automated extraction of information from natural language descriptions, such as the work of Hill *et al.* [9], and Jim [10], but clearly there is more to be done to address the problem of using natural language documentation as specifications for purposes of testing, debugging, or reasoning about libraries.

On-line developer forums or social networks.

On-line developer forums or social networks (such as LinkedIn) can be very useful for resolving thorny debugging problems. However, for novel problems their main disadvantage is that they need human experts and can be very slow to produce answers (several days). On the other hand, often a search can reveal answers to an analogous question, and one can translate that into an answer to one's own question, given a bit of time. Since searching for and interpreting analogous problems may take some time, on-line forums and social networks are not ideal for daily use in solving routine reference problems.

On-line forums can also be of some help in the learning problem, since they tend to collect examples, but are not typically geared towards systematic collection of teaching material.

Phone Hotlines.

Phone hotlines are not quite as slow as on-line forums, but seem quite a bit more expensive, since they must be staffed by experts. Like developer forums, they are not appropriate for daily use or for solving routine reference problems. They are certainly are inappropriate for solving learning problems.

Video Tutorials or Lectures.

Video tutorials or lectures are quite useful for orientation, background, and general information. Thus videos may be very good for initial learning. However, the pace of a video means that some details must inevitably be omitted, and it seems hopeless to try to sequentially search a video for specific answers to particular (reference) problems. (On the other hand, perhaps better indexing facilities for videos could help solve this problem.)

Books.

Books can be very good for both learning and reference purposes, and there are several books about popular libraries. However, the books I have of this sort are usually too dry to read except when first learning a library, and thus usually only get used as a reference. Even as reference materials physical books (on paper) are often not ideal, since indexing is often not done well and the reader often does not know the appropriate index terms. However, electronic books can be searched more easily. Books may be equipped with several different facilities for accessing information (outlines, indexes, etc.). Books can also collect not only natural language documentation but can also present examples of proper usage (code). (Electronic books could potentially also run or animate such examples.)

2.2 Code-Based Approaches

In theory, source code written in some programming language allows for communication between humans and machines, so programs have the potential to be both precise and communicative. Code in a programming language is mathematically precise, and is often more detailed than is typical in mathematics. Being mathematically precise, code can also be manipulated by various automated tools, including debuggers and verification tools. In addition, programmers fluent in different natural languages can all understand the same programming language (although names for variables and methods may need translation).

Examples.

Examples of proper usage of a library can be very illuminating, especially for novice users. Such sets of examples may be collected and presented in tutorial fashion (a good example being Campione and Walrath's Java tutorial [2]) or may be found in the wild by searching in code repositories (such as sourceforge.net or Google Code).

One problem with finding examples in the wild is that these examples are necessarily open source, since proprietary code will not be available for browsing.[3] Another problem is that searching for existing examples does not help in creating an initial set of examples.

Source Code.

A library's source code is a precise specification of the library. However, source code is not available for proprietary libraries. Furthermore, even if it is available, source code does not distinguish between its effect (what it accomplishes) and its intent. This makes source code often be too detailed. This detail makes updates to the library difficult, since an implementer can never be sure what detail of algorithms or data structures a client may have relied on. Such violations of information hiding principles can also make it difficult for the client to extract an easily remembered summary of the code's effect, making source code less than ideal for solving either the learning or the reference problem.

Tests.

Tests (in code form) for the library are not often used as documentation by clients. Nevertheless, they are a kind of weak specification, and are often produced in the course of normal development. Unit tests can be considered a specification of the unit being tested, since they precisely describe a small amount of behavior. The weakness of unit tests is that they do not cover all possible test cases, unless care is taken to generalize them (e.g., by parameterization [18]). Corner cases involving aliasing patterns are particularly hard to set up and may be felt to be rare enough to not be worthy of testing.

Larger system tests often have the complexity of applications and could serve as tutorial examples if they were also designed to be understandable.

A problem with using testing code as documentation is that, in imperative programming languages, there may be a significant amount of state that is passed to the test implicitly (e.g., in the heap), and this implicit state may be obscure to a person reading the test. Similarly, there can be effects (e.g., changes to the heap) that are hidden, for good reasons (abstraction and information hiding), which may make such tests difficult to understand.

2.3 Mathematical Approaches

Formal (i.e., mathematical) specifications can be more general than test cases. For example, a predicate-based specification of a method can be thought of as a generalization of all possible test cases. Thus formal notations can specify more of the state space of a module than is easily achieved with tests. Another advantage of formal notations is that they can also be designed to call attention to all inputs, outputs, and effects involved in a behavior, including subtle changes in the heap.

Since formal notations are mathematical, they are usually unambiguous and can be very precise. They can also be used by automatic tools, such as test case generators or verifiers.

However, the price of these advantages is that the people reading and writing formal specifications need some mathematical training. In particular, they have to be reasonably fluent in logic. It also seems helpful if they have a background in functional programming, as they need to be comfortable with describing changes to state (as opposed to making changes to state). Some experience with using formal notations is also necessary to choose the right level of abstraction, so that specifications are not too detailed and yet can still be used with the appropriate automatic tools.

Hoare-Style (Data) Specifications.

Most of my research has been with Hoare-style specifications, such as specifications of object-oriented classes using invariants, and pre- and postconditions for each method. Such specifications are good for answering reference questions about particular methods, but are not good at solving problems related to the effect of long sequences of method calls.

Hoare-style specifications also have some technical weaknesses. First, they are difficult to reason about when there is a lot of aliasing, since the notation tends to make it easy to confuse names and denotations. But there are various technical fixes (such as Separation Logic) for dealing with this problem. Second, Hoare-style specifications that are written using logics like first-order logic are also not very constraining of implementations in higher-order situations, such as methods that call other methods (e.g., mapping a method over the elements of an array). I favor the greybox approach to solving this kind of specification problem [16].

Temporal (Control) Specifications.

Specifications written in temporal logic, such as CTL or LTL [14] have somewhat complimentary advantages compared to Hoare-style specifications. In particular, temporal logics are very good at specifying (long or even infinite) sequences of operations or method calls. Such specifications are also useful for describing properties of execution histories, such as absence of deadlock or lack of starvation. However, to facilitate their use in model checking, temporal logic specifications often have very abstract (i.e., weak or nonexistent) descriptions of data values.

3. CONCLUSIONS

The main conclusion I draw from the above examination of different documentation technologies is that none of them is ideal. I have thus come to think that future documentation systems should combine the best features of each, and that this will probably require specifications that use a combination of technologies.

To get an idea of what this combination might look like, consider the Z specification language [17]. Z was designed from the start to be combined with natural language descriptions, a feature that makes it easy to integrate Z into books, lectures, and other natural language documentation. In this way one can use natural language to give motivation and perspective, and when it is useful, precisely describe some

[3]On the other hand, having a searchable set of proprietary examples may be desirable for internal use in a business that wants to gain a competitive advantage.

concepts in formal notation. It may be that literate programming [11] technology can also aid in the combination of code, natural language, and specifications.

The connection between formal specifications and testing can also be exploited more fully [7, 8, 19]. For example, it is possible to use tests to aid proofs of correctness [6] or as representations of regions of a program's state space that have been explored [1]. One can also use specification ideas to make tests more general [18]. Conversely formal specifications can be used as test oracles [4, 15]. More generally, combinations of static and dynamic approaches promise to be fruitful, because their strengths and weaknesses are complimentary [1, 3, 5].

I think there is also great potential in exploiting tests as examples of proper usage of a library, but this is just a hunch.

There also seems to be considerable potential value in the combination of Hoare-style and temporal logic specifications. The main advantage is that each kind of specification is weak where the other is strong. However, the fruitful combination of these two techniques may require some technical innovations.

For solving the reference problem, indexing and searching are traditional techniques that have been used with some natural language technologies (e.g., books), but these could be more systematically exploited with other technologies (e.g., test cases or formal specifications).

Another thing that formal techniques can take from natural language techniques is the importance of rhetorical emphasis. Formal specifications are often very flat, and it seems hard to emphasize some parts of the specification as particularly important to readers. But JML [13] includes some features along these lines [12], which are designed to emphasize certain rhetorical points. For example, in JML one can state examples in the formal specification and one can also state redundant consequences of the specification as a means to point out important facts to the reader. Still, formal specification techniques are not as good as test cases at describing the relative importance of certain behaviors. That is, it is hard to say that certain input states are less important than others, perhaps because they are less likely to be produced by real users.

In summary, my hope is that software engineering research can find economically viable ways to combine the many technologies available for documentation and specification of libraries, and that the combination will be more useful and productive than each technology individually.

4. ACKNOWLEDGMENTS

The author's work was supported in part by NSF grants CNS 08-08913, CCF-0916350, and CCF-0916715. Thanks to the anonymous reviewers for their comments and in particular for pointing out some work to cite on natural language understanding. Thanks to Sumit Jha for comments on an earlier draft.

5. REFERENCES

[1] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. D. Tetali, and A. V. Thakur. Proofs from tests. *IEEE Transactions on Software Engineering*, 36(4):495–508, July/August 2010.

[2] M. Campione and K. Walrath. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[3] R. Cartwright and M. Fagan. Soft typing. *ACM SIGPLAN Notices*, 26(6):278–292, June 1991. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (Toronto, Canada).

[4] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[5] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis, Portland, OR*, pages 24–27, New Mexico State University, May 2003. Jonathan Cook.

[6] M. Geller. Test data as an aid in proving program correctness. *Commun. ACM*, 21(5):368–375, May 1978.

[7] P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In J. Romijn, G. Smith, and J. van de Pol, editors, *Integrated Formal Methods, 5th International Conference, IFM 2005, Eindhoven, The Netherlands, November 29 - December 2, 2005, Proceedings*, volume 3771 of *Lecture Notes in Computer Science*, pages 20–32. Springer-Verlag, 2005.

[8] E. L. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Aspects of Computing*, 17(2):201–221, 2005.

[9] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, Nov. 2007.

[10] T. Jim. Yakker: A parser generator for network protocol messages. PDF of talk, Nov. 2005. `http://www2.research.att.com/~trevor/talks/yakker-njpls.pdf`.

[11] D. E. Knuth. *Literate Programming*, volume 27 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford University, 1992.

[12] G. T. Leavens and A. L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999.

[13] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. JML Reference Manual. Available from `http://www.jmlspecs.org`, Sept. 2009.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, NY, 1992.

[15] D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In *Proceedings of*

ISSTA 94, Seattle, Washington, August, 1994, pages 58–65. ACM Press, Aug. 1994.

[16] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Canada*, pages 351–367, New York, NY, Oct. 2007. ACM.

[17] J. M. Spivey. *Understanding Z: a Specification Language and its Formal Semantics*. Cambridge University Press, New York, NY, 1988.

[18] N. Tillmann and W. Schulte. Parameterized unit tests with unit meister. In *ESEC/FSE'05*, pages 241–244, New York, NY, Sept. 2005. ACM.

[19] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In L. L. Pollock and M. Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine*, pages 145–156. ACM, July 2006.