Towards a taxonomy of tools for documenting code design

Sergio Castro,Kim Mens,Johan Brichau
Université Catholique de Louvain
Louvain-la-Neuve, Belgium
{Sergio.Castro — Kim.Mens — Johan.Brichau}@uclouvain.be

Abstract

Numerous tools for documenting code design exist. Each of these proposes different techniques and attempts to deal with different aspects regarding code design documentation. Though this abundance of tools and techniques is a clear sign of the importance of such kind of tools in real life software implementation, it also creates confusion when trying to decide which one best suites the specific needs of a project. Adding to this confusion is the fact that different tools often use a different terminology for similar problems. Furthermore, the support offered by most of these tools is limited, since it is often based either on verifiable documentation that is not highly customizable, or customizable documentation that cannot easily be verified. This paper takes initial steps in the direction of establishing a common vocabulary for describing code design documentation tools, and highlights the features that a verifiable and highly customizable tool should provide in order to satisfy most of the code design documentation requirements present in the implementation of complex software systems.

1 Introduction

Appropriate documentation of the program design of complex software systems is crucial for its late maintenance and evolution. However, in order for this type of documentation to be truly useful, it must be verifiable so that it can be used to test the quality of software, it should minimize duplication and scattering of documentation artifacts, it should be malleable enough for describing different kinds of implementation details, it should avoid repetitive and error prone activities as much as possible.

There exists a large number of code design documentation tools and techniques that attempt to cope with subsets of these ideal properties, ranging from very sophisticated tools relying on advanced code querying techniques and meta-programming as a mechanisms for defining and verifying constraints over source code elements, to very simple ones that just describe ways of writing code design documentation in a textual representation and a mechanism for exporting this documentation to a human-readable format.

This work describes an initial taxonomy of the most relevant features of these tools, trying to center our attention on the concepts used by the most sophisticated tools, but without ignoring completely the conceptually simple ones, since even those occasionally describe a feature that is not taken into account by the more complete ones. Our second goal is to build a common vocabulary for describing all these features, providing in this way basis for further discussion. We have structured this paper as follows: Section 2 explains our proposed taxonomy, section 3 describes how sixteen documentation tools were classified according to this taxonomy, and section 4 mentions our conclusions and future work.

2 Towards a taxonomy of code documentation tools

We intensionally focused this work on features of implementation-level program-design documentation[1] tools. Therefore, we excluded from our study features related to reverse engineering activities that, although associated with documentation tasks, would force us to deviate the discussion to other phases of the development life cycle (i.e., analysis and design) that are out of the scope of this paper (i.e., implementation).

With this focus in mind, our study and comparison of different program documentation tools allowed us to classify these tools according to different documentation dimensions, each of which will be discussed in more detail in the remainder of this section. A list of all the found dimensions and the researched tools is shown in the appendix A.

[1]referred as just *documentation* in the rest of this paper

2.1 Documentation retrieval

This dimension describes how documentation is retrieved. At a very high level, we found that all tools we examined produce the documentation by either extracting it from an already documented system or by attempting to generate new documentation. We can thus distinguish:

Extracted documentation: Existing documentation is extracted from code already documented (e.g., a legacy system), and usually formatted in a human readable format (e.g., [22], [1], [19])

Generated documentation: Newly generated documentation is added to the source code.

The latter category, generated documentation techniques, can be further decomposed:

Mined documentation: New documentation is mined from the code with little manual intervention (e.g., [21]). However this generated documentation is often incomplete.

Library oriented documentation: A reusable library of documentation patterns is applied over different pieces of software [10]. Typically this documentation library covers implementation design patterns or bad smells in code that are applicable across different software projects (e.g., [3], [15], [11]).

Manual documentation: The documentation is written from scratch. No mining nor documentation libraries are present.

2.2 Documentation kind

Whereas the previous dimension focused on how to document, this dimension focusses on what possible artifacts can be documented. We distinguish:

Structural documentation [16]: The documentation describes structural and static relationships in the program (e.g., [13], [11], [18]).

Behavioral documentation [16]: The documentation describes the behavior of the application (e.g., [12], [13]).

Evolutionary documentation [5]: Documents the evolution of software artifacts or concepts. Note that this dimension is somewhat orthogonal to the previous two since we could document the evolution of structural, behavioral or both kind of documentations. However, this dimension focusses on different aspects of the evolution history of source code elements, rather than being fixed to one single version of the code (e.g., [17]).

2.3 Declarative properties

This dimension describes how declaratively and modularized the documentation can be expressed. At a high level we can classify this dimension in:

Extensional documentation [7] techniques document different software artifacts separately, even if they all share common properties that could be documented together (e.g., If all the factory methods should begin with the word ‘factory’, and extensional documentation technique would force us to find every factory method and document this constraint there).

Intensional documentation [14] techniques provide mechanisms for declaratively documenting common properties of a set of software artifacts. (e.g., using an intensional documentation technique we could declaratively say with one single rule that every method belonging to the concept of a factory method, should begin with ‘factory’). This technique, in addition of being more concise and less error prone, has the additional advantage that the documentation of conceptually equivalent objects is modularized in one single place, instead of being scattered through and tangled with the program code. We found that most of the researched tools implementing support for intensional documentation, provide some kind of source code querying facility, that allow the declarative assertion of constraints and properties over distinct source code elements.

Intensional documentation techniques can be further classified by the type of source code querying mechanism they offer:

Predefined intensions: Where the intensions are selected from a library of predefined intensional predicates (e.g., [17], [8], [6]).

Customizable intensions: Where the intensions are user-customizable and highly adaptable, usually with the use of a declarative meta-programming language (e.g., IntensiVE). Evidently, neither of these categories are mutually exclusive and a simple documentation tool may provide support for all the above (e.g., [13], [11]).

2.4 Documentation verification

This dimension describes how documentation can be verified in source code. We will call any documentation that can be automatically verified *active documentation* In the case of tools that provide no automated support for verifying the documentation we talk about *passive documentation* (e.g., [22], [1], [19]) . We have found that active documentation can be further classified in three different dimensions.

2.4.1 Reactive, proactive and retroactive verification

Reactive verification: When documentation can be checked upon explicit request by a user (e.g., [20], [2], [3]) .

Proactive verification: When documentation is enforced when an inconsistency between the documentation and the source code is detected (e.g., [8], [12]).

Retroactive verification: When the documentation is checked automatically (e.g., in batch) after a series of changes to the code. The verification is triggered by specific changes in the code or in the documentation.

Note that these properties are complementary, since different verification mechanisms can be offered by the same tool.

2.4.2 Static and dynamic verification [4]

Static verification: The source code is verified statically, i.e., without executing it (e.g., [9] [15]).

Dynamic verification: The documentation is verified during the execution of the code (e.g., [12]).

Note that these two properties are complementary, since not all the implementation artifacts can be fully statically verified, nor can all of them can be dynamically verified.

2.4.3 Symptomatic, diagnostic and corrective verification

Symptomatic verification: Shows where the documentation is not consistent with the source code (e.g., [13]).

Diagnostic verification: Evaluates the inconsistency between documentation and code, and determines the cause of the problem (e.g., [6], [15], [12]).

Corrective verification: Attempts to (semi-)automatically solve the detected inconsistencies (e.g., [8]).

Note that if a tool supports corrective verification, this implies that it support diagnostic verification as well (since no correction could be done if the cause of the problem has not been diagnosed before). In a similar way, if a tool supports diagnostic verification, this implies that it also supports symptomatic verification (since diagnosis cannot be done if the problem is not known).

2.5 Documentation quality measuring

This dimension is about evaluating the quality of the documentation itself. It is most of the time present in documentation tools as a quantitive property (e.g., [20], [15]). The main measure of quality is the average or the proportion of source code artifacts that are documented in the system. Evidently, other documentation quality measures may be supported as well.

Though we think that this dimension can be classified in *customizable quality measuring* and *non customizable quality measuring*, we have classified the tools we researched only as customizable or non-customizable tools. We took this decision since very few tools provided quality measuring support, and from the ones which did, they all offered customizable quality measuring.

2.6 Independence of implementation language

This dimension is about the flexibility offered by the documentation tool for being used with more than one implementation language. We can further classify this dimension in:

Single language: The documentation targets just one programming language (e.g., [6], [19]).

Single paradigm: The documentation targets one object paradigm and a subset of languages written in this paradigm (e.g., [13]).

Language and paradigm independent: The documentation targets multiple paradigms and languages (e.g., [1]).

3 Applying our taxonomy to real tools

Appendix 5 shows how different documentation tools were described according to the documentation dimensions we found. Although space limitations prohibit us to discuss this table in detail here, we do highlight some of the main conclusions drawn from analyzing it.

- Almost no tool exists that provides support for certain features, such as measuring the quality of the documentation, proactive verification techniques, and dynamic verification techniques.

- Conversely, other dimensions, such as the use of reusable documentation and symptomatic and reusable verification, show a high density of tools that support them.

- Furthermore, we observed that some dimensions are present in almost all the tools studied, such as structural and behavioral documentation, and that the tools we studied that were more close to include all properties are Semmle, KlocWork, NDepend and IntensiVE.

- Finally, from the tools that were more close to include most of the properties discussed here, we discovered that all of them make use of a code querying language as a mechanism for the declarative documentation of source code artifacts. That fact triggered the observation that the use of a sophisticated query language for a documentation tool, in fact guarantee the support for other documentation features. For example, extracted documentation is easily supported, since most of the times the textual documentation that has to be extracted from code, is represented by special tags or comments (e.g., Java Doclets [19]) that can be easily queried, examined and formatted with an appropriated query language. Also, extensional documentation is automatically supported, since we can consider the declarative documentation of source code artifacts referenced by a query that returns exactly one artifact, as an extensional strategy.

4 Conclusions and Future Work

In this paper we have presented an initial taxonomy of different properties of documentation tools.

We have defined with this an initial common vocabulary for describing this kind of tools, as well as found which are the properties more common or infrequent and highlighted some interesting relationships among these properties. We discovered also that even the most sophisticated tools do not offer all the features we explored. Taking this into account, as part of our future work we will attempt to give an additional step towards the development of a complete documentation tool. We think that the obvious choose for this objective is extending the IntensiVE tool suite, since it already provides most of the properties discussed and it is an open source project that provide complete access to its source code. In order to do these, we will look for mechanisms for providing in IntensiVE support for documentation quality measuring, proactive and retroactive documentation, dynamic documentation support, diagnostic and corrective verification, and support for evolutionary documentation.

5 Acknowledgements

This work was partially funded by the Interuniversity Attraction Poles Programme of the Belgian Science Policy.

References

[1] E. Artzt. Autoduck. http://helpmaster.info/hlp-developmentaids-autoduck.htm.
[2] AxTools. Codesmart. http://www.aivosto.com/codesmart/net.html.
[3] T. Copeland. Pmd. http://pmd.sourceforge.net/.
[4] M. Elaasar and L. Briand. An overview of uml consistency management. Technical Report SCE-04-18, Carleton University, Ottawa, Canada, 2004.
[5] G. Engels, J. M. Küster, R. Heckel, and L. Groenewegen. Towards consistency-preserving model evolution. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 129–132, New York, NY, USA, 2002. ACM.
[6] I. GrammaTech. Code sonar. http://www.grammatech.com/products/codesonar/.
[7] K. D. Hondt. *A novel approach to architectural recovery in evolving object-oriented systems*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1998.
[8] K. Inc. klocwork. http://www.klocwork.com/.
[9] S. Johnson. Lint. http://en.wikipedia.org/wiki/Lint_programming_tool.
[10] W. Liu. *Rule-based Detection of Inconsistency in Software Design*. PhD thesis, University of Toronty, Canada, 2002.
[11] S. Ltd. Semmle. http://semmle.com/.
[12] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language.
[13] K. Mens and A. Kellens. Intensive, a toolsuite for documenting and testing structural source-code regularities. *10th Conference on Software Maintenance and Re-engineering (CSMR)*, pages 239–248, 2006.
[14] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intional source-code views. In *Int. Conf. Software Engineering and Knowledge Engineering*, pages 289–296. ACM Press, 2002.
[15] A. Oy. Project analyzer. http://www.aivosto.com/project/project.html.
[16] Ragnhild, T. Mens, J. Simmonds, and V. Jonckers. Using description logics to maintain consistency between UML models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 326–340. Springer-Verlag, 2003.
[17] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, New York, NY, USA, 2002. ACM Press.
[18] S. S.A.R.L. Ndepend. http://www.ndepend.com/.
[19] Sun. Javadoc technology. http://download.java.net/jdk7/docs/technotes/guides/javadoc/index.html.
[20] toolsfactory software inc. doc-o-matic. http://www.doc-o-matic.com/start.html.
[21] T. Tourwe, J. Brichau, A. Kellens, and K. Gybels. Induced intentional software views. *European Smalltalk Users Group conference*, 2003.
[22] D. van Heesch. Doxygen. http://www.stack.nl/ dimitri/-doxygen/.

Appendix A

Comparing documentation tools and documentation dimensions

In this appendix we list all the documentation dimensions we found in our research, as well as the classified programming design documentation tools. These dimensions and tools are showed in table 1.

Table 1. Documentation dimensions Vs. documentation tools

	Quality	Multi language	Extracted	Mined	Manual	Library	Extensional	Predefined Intensions	Customizable Intensions	Passive	Reactive	Proactive	Retroactive	Static	Dynamic	Symptomatic	Diagnostic	Corrective	Structural	Behavioral	Evolutionary
Doxygen		X	X	X	X		X			X									X	X	
Autoduck		X	X		X		X			X									X	X	
Javadoc			X		X		X			X									X	X	
Doc-O-Matic	X	X	X	X	X		X				X			X		X			X	X	
FEAT					X			X			X			X		X	X	X	X	X	X
PMD						X					X			X		X			X	X	
PQL					X				X		X	X		X	X	X	X	X	X	X	
IntensiVE		X	X	X	X	X	X	X	X		X			X		X			X	X	
Project Analyzer	X			X		X					X			X		X	X	X	X	X	X
Code Smart		X				X					X			X		X			X	X	
NDepend		X	X		X	X	X	X	X		X			X		X			X	X	X
Lint						X					X			X		X			X	X	
Code Sonar					X	X	X	X	X		X			X		X	X			X	X
PolySpace		X				X					X			X		X				X	
KlocWork		X				X	X	X	X		X	X		X		X	X	X	X	X	
Semmle		X	X		X	X	X	X	X		X			X		X			X	X	