WikiWiki Weaving Heterogeneous Software Artifacts

Ademar Aguiar
FEUP, Universidade do Porto
INESC Porto
Rua Dr. Roberto Frias s/n
4200-465 Porto, Portugal
ademar.aguiar@fe.up.pt

Gabriel David
FEUP, Universidade do Porto
INESC Porto
Rua Dr. Roberto Frias s/n
4200-465 Porto, Portugal
gtd@fe.up.pt

ABSTRACT

Good documentation benefits every software development project, especially large ones, but it can be hard, costly, and tiresome to produce when not supported by appropriate tools and methods.

The documentation of a software system uses different artifacts, namely *source code*, for low-level internal documentation, and specific-purpose *models* and *documents*, for higher-level external documentation (e.g. requirements documents, use-case specifications, design notebooks, and reference manuals). All these artifacts require continual review and modification throughout the life-cycle to preserve their consistency and value.

Good software documents are often *heterogeneous*, i.e., they combine *different kinds of contents* (text, code, models, images) gathered from *separate software artifacts*, a combination usually difficult to maintain as the system evolves over time, considering that source code, models and documents are typically produced and maintained separately in multiple sources using different environments and editors.

This paper presents a wiki that helps on quickly *weaving* different kinds of contents into a single heterogeneous document, whilst preserving its semantic consistency. The fundamental goal of this wiki (XSDoc Wiki) is to reduce the *development–documentation* gap by making documentation more convenient and attractive to developers. An example taken from the JUnit framework documentation helps to illustrate the features more relevant to do such weaving.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; I.7.2 [**Document and Text Processing**]: Document Preparation—*Hypertext/Hypermedia, Markup Languages*

General Terms

Wiki-based software documentation

WikiSym '05, October 16-18, San Diego, CA, U.S.A.

Keywords

Software documentation, web-based documentation, wiki

1. INTRODUCTION

Although not always recognized, documentation plays a central role in many software development tasks. Most of the development effort is spent on formalizing information, i.e., on reading and understanding requirements, specifications, and other informal documents, with the intent of producing concrete source code and models.

Good software documentation usually provides *multiple views* of a system (static, dynamic, external, internal) at *different levels of abstraction* (architecture, design, implementation). Depending on the concrete aspect to document, it may be convenient to use and mix contents represented in *different notations* (text, code, models, images) gathered from *different types of artifacts* (requirements documents, use-case specifications, design notebooks, reference manuals, source code files, and model files).

As a result of this complexity, software documentation usually results hard, costly, and tiresome to do, especially when not supported by appropriate tools and methods.

This paper presents a wiki (XSDoc Wiki) that addresses the problem of gathering different kinds of contents from separate software artifacts and weaving them together into a single document — here called an *heterogeneous software artifact* — whilst ensuring the semantic consistency between the contents.

After a brief overview of the most relevant approaches to the problem of combining source code and documentation in a single artifact, namely literate programming and its alternatives, we present our wiki-based approach based on the XSDoc Wiki.

Along the paper, it will be used the article "JUnit: A Cook's Tour" [6], part of the JUnit's framework documentation, as an example of an heterogeneous document, hereafter referred in this paper as the "JUnit cook's tour". This article explains how the framework itself is constructed and presents its goals, describes its design in terms of patterns, and its implementation as a literate program, for which it interleaves text with several source code fragments and models (Figure 1). This example will help to illustrate the pros and cons of each of the possible approaches and show how to use the XSDoc Wiki to create, integrate and present contents.

The paper concludes with considerations about the pros and cons of our wiki-based solution.

2. SOFTWARE DOCUMENTATION

The software artifacts produced during development can be categorized in *source code*, *models*, and *documents*, all of which require continual review and modification throughout the life-cycle in order to preserve its consistency and value.

Typically, these artifacts are cooperatively produced and maintained in separate sources by different team elements, for which they often use different environments and editors, namely text editors, source code editors, model editors, and document editors.

This diversity of tools often requires constant switching between working environments and results inappropriate to maintain the semantic consistency between the artifacts, as it disturbs developers and originates process inefficiencies.

2.1 Internal and External Documentation

The understanding of a software system can be documented internally in the source code or externally in documents.

Typical internal documentation captures the understanding of a program, preserving it over time, but is limited to low-level textual explanations usually included in source code comments that are not convenient to document global system understanding, i.e., one that crosses several sections of a software system.

On the other hand, higher-level external documentation is capable of capturing the components and connectors of an architecture and the interactions of cooperating classes but the consistency between external documents and source code can be difficult to maintain as the system evolves over time.

The JUnit cook's tour (Figure 1) of our example falls in this last category of external documentation: it is a standalone document, is structurally independent of the source code, and is intended to capture the design and implementation of JUnit.

As we will see, this external document suffers from the typical problem of inconsistency with source code, which, however, in this case is not very problematic because only the understanding of the JUnit's implementation is partially affected, and the understanding of its design is practically not affected.

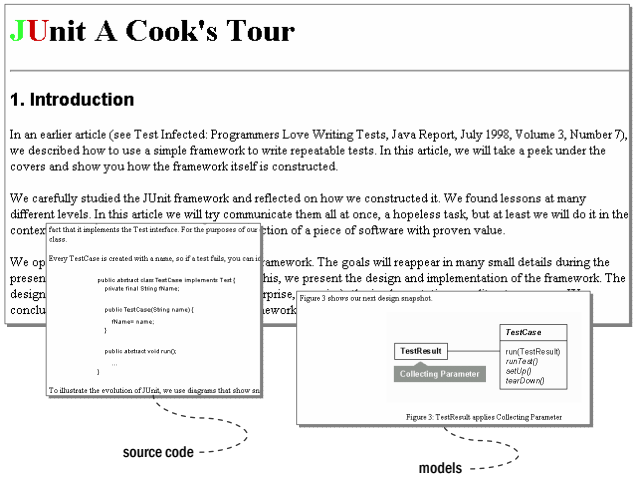

Figure 1: Example: the article "JUnit: A Cook's Tour".

2.2 Literate Programming Approach

The literate programming approach [19] is a possible solution to the problem of having source code and documentation always consolidated. The technique was invented by Donald Knuth and involves writing documentation and code in a *single source document* (*verisimilitude*), *psychologically organized for comprehension by humans* rather than computers.

Literate documents are *tangled* to produce computer understandable code, and *woven* to produce human readable and comprehensible typeset documents (Figure 2). The technique provides significant incentives for developers to document while they code, potentially leading to programs of higher quality and maintainability.

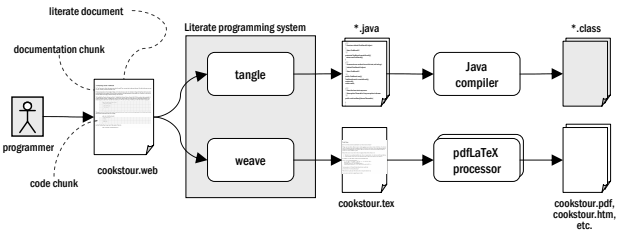

Figure 2: Literate Programming approach.

Despite its advantages, the technique is not widely used. This is mainly due to the integration difficulties of literate programming tools in mainstream development environments, but not exclusively:

- it requires the combined use of three languages, a programming language, a documentation language and an interconnection language;
- it introduces too much overhead for small programs;
- the format of literate documents can be complex, what seriously compromises their on-screen readability and understandability during development;
- and most importantly, the organization of the source code seen by the compiler is different from the original, as written by the programmer, what often causes integration problems with tools that directly analyse and manipulate source code files, such as integrated development environments (IDEs), debuggers, code generators, reverse-engineering tools, and refactoring tools.

For example, to produce the JUnit cook's tour, we would need to start by writing a literate document (e.g. `cookstour.web`), containing the text and the source code fragments used, organized in the way we would prefer. After being processed by the tangle and weave programs we would have the Java source code files (e.g. `TestCase.java`, `TestSuite.java`, etc.) the TEX files (e.g. `cookstour.tex`), and then the compiled program and final document.

As mentioned before, one of the biggest problems of literate documents to programmers is that the source code files are not primary artifacts, which programmers actually write, but derived artifacts, automatically generated, not intended to be edited or changed by hand or any tool external to the literate programming system.

In short, the literate programming approach is very elegant and effective but it often lacks *real world usage* by ordinary programmers and teams who have not themselves designed the tools [24].

Alternative documentation approaches to literate programming can be classified into *single source* approaches and *multiple source* approaches.

2.3 Single Source Approach

In the single source approach, code and documentation are integrated together in a single file, so no consistency problems are possible between them as there is no replication of contents in the whole documentation bundle.

The Sun's Javadoc utility [26, 20] is an example of a single source approach that *weaves* Java source files to produce interface documentation in HTML using simple Java commenting conventions (Figure 3). Another very popular example of tool that uses this single source approach is the Doxygen tool [28].

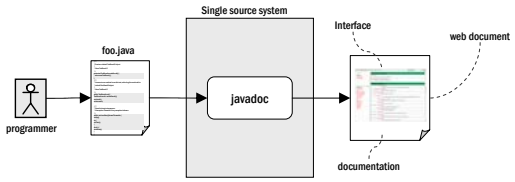

Figure 3: Single source approach.

This kind of tools, like Javadoc and Doxygen, cannot be considered literate programming tools because the structure of the documentation they can produce is restricted to follow the code's structure, therefore lacking the support for psychological arrangement of the documentation.

For example, this kind of approach is not convenient to produce the JUnit cook's tour because the structure of the document doesn't fit in the code of any single class, as it crosscuts several classes.

2.4 Multiple Source Approach

Multiple source approaches maintain documentation and code in separate files (Figure 4).

Due to the physical separation of code and documents, documenters usually combine the contents of both kinds of artifacts by copying source code into document editors, or by referring to a specific range of lines of a program file. But, as soon as the code changes, the two artifacts become inconsistent.

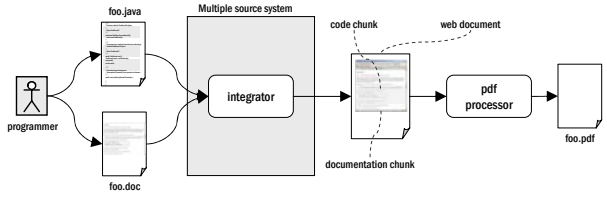

Figure 4: Multiple source approach.

Traditional documentation written externally to source code is the most common example of such technique.

For example, we can easily write the JUnit cook's tour using a simple text editor, or HTML editor, and copy-paste the fragments of code from source code files into it. However, very soon this would lead to inconsistencies between the document and the source code base, as actually happens in the JUnit cook's tour with some fragments of the `TestCase` class (e.g. the class definition).

Unless these inconsistencies are not considered problematic, they must be avoided. Without adequate tool support, developers often postpone the act of documenting from the *right moment* to the *last moment*, to reduce the effort of manually preserving the semantic consistency between documents and code.

Considering that the most complete mental model of the problem understanding and design of a solution is formed by the developer at implementation time, most of this understanding is often lost in the code, or in the developers head if it is not adequately documented at the right moment, thus requiring a large effort to recover it later.

More sophisticated systems, such as DOgMA [25] and Elucidators [22, 23], simulate the verisimilitude characteristic of literate programming with tools that automatically manage the relationships between code and documents. The most typical problem with this kind of tools is that the edition of code and documents outside their specific environments is often not convenient, a serious obstacle for easy integration in mainstream development environments.

2.5 XML-based Documentation

The hierarchical nature of XML documents is very useful to implement structuring mechanisms for software documents. An evidence of this, is the diversity of work applying XML specifically to software documentation, which include the following, to mention only a few of the more relevant to this paper: representations of source code (JavaML [2, 5], cppML [21], srcML [10]), UML [14], mechanisms to integrate heterogeneous documents from different sources [16, 15, 4], and literate programming systems [31, 9].

XML is an open format, with many standards, and it provides powerful querying capabilities with standard tools widely available in many platforms, such as XSLT [33] and XQuery [34]. Therefore, XML is definitely a very important technology to consider when looking for solutions to contents interoperability and combining heterogeneous contents of software documents.

The JUnit cook's tour is an example of a document that can be produced in XML (e.g. XHTML) with less extra effort, when compared to its HTML version, but with some additional advantages in terms of querying and processing capabilities of their contents. For example, if the Java source code fragments are represented in XML using JavaML 2.0 [2], it would be simple to add full hyperlinking capabilities to each source code symbol, improving its navigability both to internal and related external contents and documents.

2.6 Web-based Documentation

Most of the documentation now delivered with software systems is web-based. Web-based documentation has several advantages: low cost, immediate accessibility, always up-to-date information, search and query facilities to quickly locate information, attractive presentation, multiple navigation, and multimedia. Effective and attractive web-based documentation increases the speed with which users acquire proficiency.

Although not being so proeminent anymore, the paper-format still retains its own place and importance: it is tangible, customizable, more comfortable to read off-line and off-screen, and is calm. Research revealed that people retain more if the information is presented as printed text rather than displayed on a screen.

2.7 Wiki-based Documentation

One of the problems of producing web-based documentation is that web browsers are not (yet) capable of editing pages and they still do not support the rich structuring, navigation, and annotation features of hypertext documents [30].

A wiki [12] is a very simple and appealing collaboration tool capable of presenting and editing web-based information using a simple web browser. Wiki documents have several good properties: they are open, can evolve incrementally and organically, are easy to edit and organize, promote convergence of contents and consistency of terms, are tolerant, and are easily observable by other users.

Wikis normally provide a very simple markup language to support text formatting and a simple mechanism based on wiki names (e.g. AnExample, or WikiName) to automatically link pages. Despite its simplicity, wiki names are very powerful because they provide a dynamic-linking mechanism [8, 7], where the link targets are not statically defined, but only calculated on the fly, on page load-time, based on contextual information, thus supporting the notion of adaptive web pages. Other kinds of linking can be defined using lexical conventions.

There are a lot of wikis available to use and install. Mainly due to their attractiveness, especially in the software community, they are frequently used to produce informal software documents: drafts of designs and implementations, design-trade-off discussions, requirements gathering, etc.

Among the many different implementations of wikis, we can find some enhanced with specific features to support activities of software development, namely: bug tracking, tests, and source code formatting [11, 17, 27, 29].

In terms of features specific for source code, the existing wikis simply support language-specific formatting of text, either directly written or copy-pasted from source code files, marked-up as code of a specific language (e.g. Java, C++, SQL, etc.), a solution that surely leads to inconsistencies as soon as the code changes. Among the already existing, no wiki was found supporting features to preserve the semantic consistency between source code and documentation to the extent possible by the software documentation approaches described before, namely the literate programming and the multiple source approaches.

In terms of features to support UML, the SnipSnap wiki [17] has a nice support for sketching UML diagrams.

It is our belief that a wiki specifically enhanced to document software, easy to integrate in open development environments, provides incentives for developers to document what they need, when they need, be it before, while, or after implementation. Therefore, we claim that wikis are a very promising tool to support software documentation.

3. XSDOC WIKI

Considering the advantages and disadvantages of the several software documentation approaches previously presented, we have devised a wiki-based approach to weave and keep in-sync source code, models, and documents.

Our approach is a multiple source approach that combines the advantages of the Knuth's approach, using a kind of reverse literate programming [18], and the attractive hypertext capabilities of wiki and XML technologies. The approach was implemented and tested in the XSDoc Wiki [3, 1], a wiki extended with several features to make convenient the agile documentation of object-oriented software, which proved to have a good real world usage.

3.1 Overview of XSDoc

XSDoc is an open and extensible documentation infrastructure based on wiki and XML technologies that ensures the semantic consistency between different kinds of contents, namely source code, models, and documents.

In addition to the wiki, the XSDoc infrastructure is composed by plugins for seamless integration in open IDEs (currently exists a simple one for the Eclipse IDE [13]), and a set of document templates, markup languages, and converters of different kinds of contents to and from XML. Figure 5 shows the key components of XSDoc as well as their interconnections.

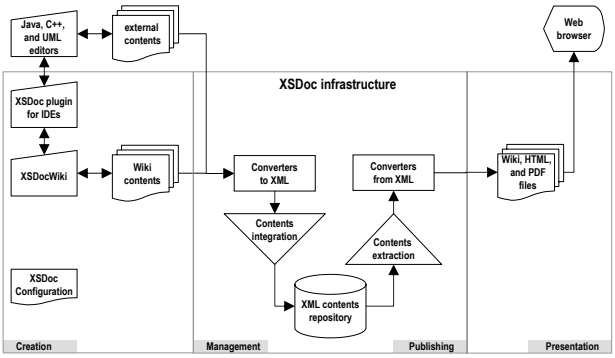

Figure 5: XSDoc architecture.

XSDoc was created to overcome the typical limitations of existing literate programming systems and alternative techniques used to solve the semantic consistency problem of software documentation, namely tools interoperability, extensibility, and integrability of heterogeneous contents (not only source code) [1].

XSDoc aims at closing the gap between *development* and *documentation* to make the activities of documenting more convenient and attractive to developers. XSDoc assists developers on documenting while they code and design, even in development environments very restrictive to documentation activities. Taking advantage of the wiki front-end, XSDoc provides a simple cooperative web-based environment for the creation, integration, publishing and presentation of software documentation.

Wiki-based software documentation is the key idea behind XSDoc. The idea was previously evaluated in earlier prototypes and was considered very attractive for developers. As a result, XSDoc combines the best features of the previous prototypes and an evolved architecture, more flexible and easier to extend.

The current implementation of XSDoc supports the integration of source code in Java and C++, UML diagrams via graphic files, plain text documents, and XML documents. Under development is the support to UML

diagrams via XMI, and source code in C#, AspectJ, and PL/SQL languages.

3.2 Support for Software Documentation

The XSDoc Wiki was developed using the VeryQuickWiki engine [11] as a starting base, which was then extended with several features to make convenient the edition and visualization of typical software documentation contents, including the following:

- linking and inlining source code fragments and UML diagrams;
- instantiation and validation of XML documents;
- access to version control systems repositories;
- easy addition of support for new styles of documents;
- and browsing controls to help on rendering contents according to user preferences.

Probably, the most important extensions to a typical wiki are the extension of the automatic linking and inlining mechanisms, originally restricted to wiki pages, to support also linking to source code, models and structured contents.

To be flexible, the XSDoc Wiki can be plugged with new styles of documents (e.g. use-cases, patterns, requirements). A XSDoc plugin typically includes: a document-template, a set of converters to map that style of document to and from XML, if necessary, a declaration of the elements to be parsed for automatic linking of wiki names, and some lexical rules to use during the automatic linking phase.

When properly configured and integrated in a specific development environment, the XSDoc Wiki promotes the collaboration of technical and non-technical people on the incremental edition and revision of software documents. Additionally, it ensures high availability of contents (always online), uses simple features, provides automated archiving, and only requires a simple web browser, a tool currently very easy to integrate in the majority of development environments.

3.3 XML Processing

As most of the documentation contents can be comfortably edited and linked using the wiki, most of the contents will reside on wiki pages stored in a contents repository, be it the file system, a version control system, or a database.

As source code programs and UML diagrams need special processing, they must be converted from their original format to XML using XSLT transformers, that convert them to specific vocabularies, namely JavaML 2.0 [2], Doxygen [28] and SVG/XMI vocabularies.

At a later stage, the contents are filtered and formatted to be published and presented. Currently, XSDoc outputs HTML files for web-based browsing, and PDF files for high-quality printing.

3.4 Contents Integration

The components of XSDoc are closely integrated, both in terms of functionality and of the information they exchange.

To merge and preserve the semantic consistency between heterogeneous kinds of contents, it is usually required an integration language, in addition to the languages or notations of the contents to integrate. For example, to integrate Java source code with TEX, literate programming systems use a third language, often a macro language, to specify how the Java contents are woven in the TEX code.

The integration language used in XSDoc is the markup language of the wiki, which is very simple to use and learn. Source code, UML models and structured documents are integrated using a multiple source approach, what means that source code and documentation reside in separate files. While this separation preserves source code files and UML files, it requires a way of managing the relationships between their contents.

In addition to the hyperlinking mechanisms provided by wikis, XSDoc provides two dynamic mechanisms to integrate and synchronize the possible kinds of document contents (source code, UML diagrams, XML files) using simple lexical conventions easy to learn and use. The mechanisms are *inlining* of contents and *linking* to contents.

3.5 Inlining Heterogeneous Contents

The inlining of contents is possible through the use of predefined tags for each kind of contents. To inline Java source code with the XSDoc Wiki, we can use the tag `[<javaSource>]` and a Javadoc reference to the fragment desired. Other tags are provided for C++ source code `[<cppSource>]`, UML diagrams `[<uml>]`, and text from wiki topics `[<include>]`.

For example, the text below is rendered by XSDoc Wiki to the source code fragment corresponding to the method `run()` of class `junit.framework.TestCase`, with all its comments removed and showing only its first and last lines.

```
Here is the template method,
initially named run and renamed as runBare:
[<javaSource>]
junit.framework.TestCase#runBare(); comments=no;
[</javaSource>]
```

The text above produces a wiki page partially shown below in Figure 6.

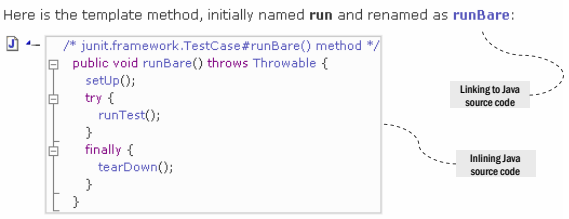

Figure 6: Inlining and Linking to Java source code.

3.6 Linking to Heterogeneous Contents

In addition to the possibility of including source code and UML contents in a wiki document, the XSDoc Wiki also provides the possibility of linking to such kinds of contents. The definition of links is possible through the use of wiki names and references with predefined formats. Similarly to the inlining, to link to Java source code with the XSDoc Wiki, we can use the prefix `javaSource:` and a Javadoc reference to the fragment desired. Other prefixes are provided for C++ source code (`cppSource:`), and UML diagrams (`uml:`).

For example, the text below is rendered by XSDoc Wiki as a link to the source code of the method `runBare()` of class `junit.framework.TestCase`.

```
Here is the template method,
initially named run and renamed as
[[javaSource:junit.framework.TestCase#runBare()][runBare]]:
```

The text above produces the link shown in the web page of Figure 6.

Having presented the features of XSDoc Wiki most relevant to the problem of weaving heterogeneous software artifacts (inling and linking), it is easy to conclude that they are very simple to use, and its basics can be learned very fast by people already familiar with the use of a web browser.

4. WEAVING EXAMPLE

The XSDoc Wiki will now be applied to the JUnit cook's tour, as an example to illustrate how it can be used to create, integrate and present the different kinds of contents involved.

4.1 Creating and Integrating Contents

The creation of *documentation contents* can be done *internally* with the XSDoc Wiki using a web-based collaborative environment, or *externally* with editors not included with XSDoc. *External contents*, e.g. source code contents and UML diagrams, always require the use of external editors.

To help on the creation of normalized documents, it is possible to use *template documents*, which can be associated with specific topic name patterns, thus being automatically instantiated at topic creation time. For example, the wiki names ending in `Pattern` can be configured to be associated with the template `DesignPattern`.

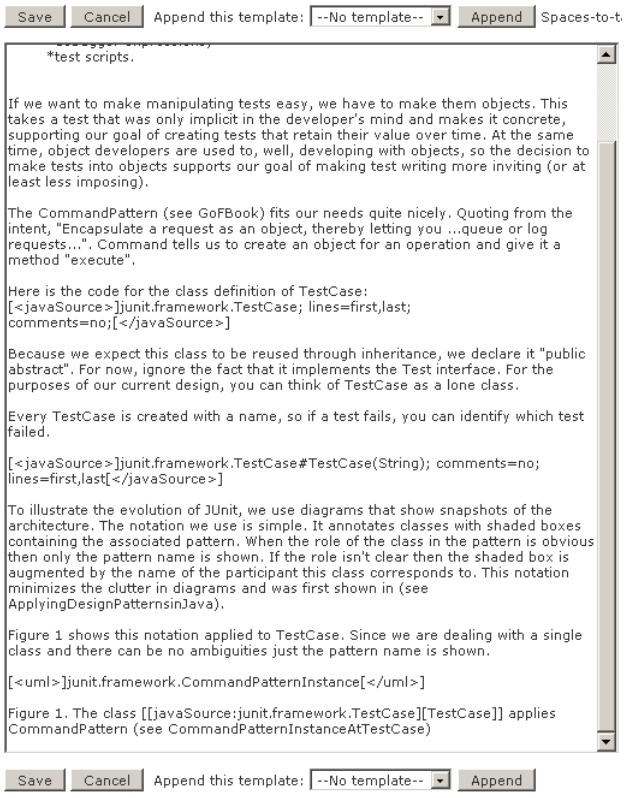

Figure 7: Editing the JUnit's cook's tour document.

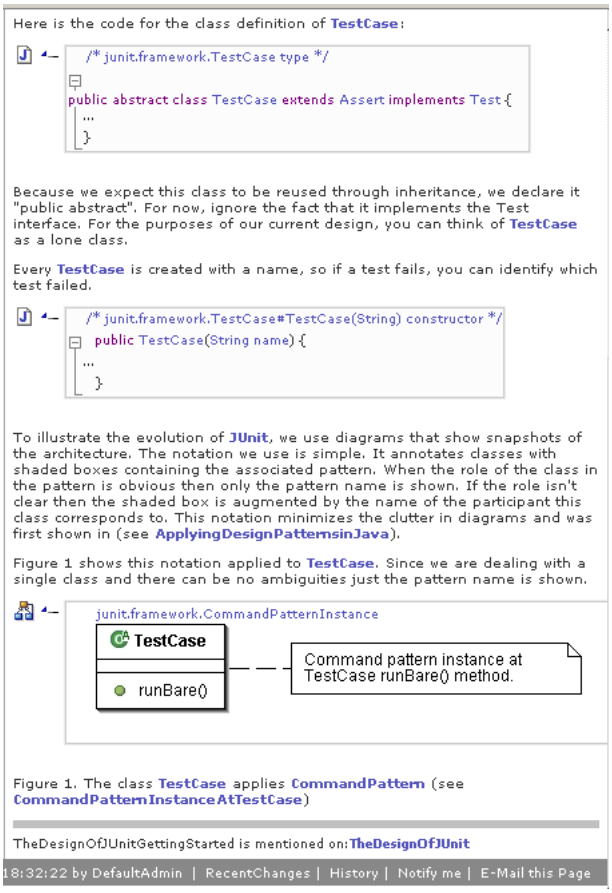

Figure 8: Viewing the JUnit's cook's tour document.

As an example, Figure 7 and Figure 8 show parts of the JUnit cook's tour that describes the instantiation of the Command pattern by the class `TestCase` in the JUnit framework. The text written in the wiki document is shown in Figure 7, and the resulting documentation is represented in Figure 8.

With XSDoc configured and integrated in an IDE, the developer has access to a web browser from where she can use the XSDoc Wiki. When documenting, the developer creates new pages, writes documents, uses IDE features such as copy-paste and drag-and-drop, browses project resources, and defines links to other pages or special contents, such as Java source code or UML diagrams, using predefined tags and linking mechanisms. In Figure 9 is represented a snapshot of XSDoc integrated in the Eclipse IDE. It is worth to mention that this possibility of editing all kinds of contents without switching environments is a very important incentive for documenting while designing and coding.

Once created, the contents are stored in the XSDoc contents repository in a textual format, XML or free-text. The contents are preserved intact as originally created.

4.2 Presenting Contents

When an heterogeneous document is requested for presentation in the wiki, the contents are then retrieved from the repository and weaved together on the fly, at page load-time, possibly from both external contents and wiki contents, and converted to the format requested.

The contents are always available for web-based browsing

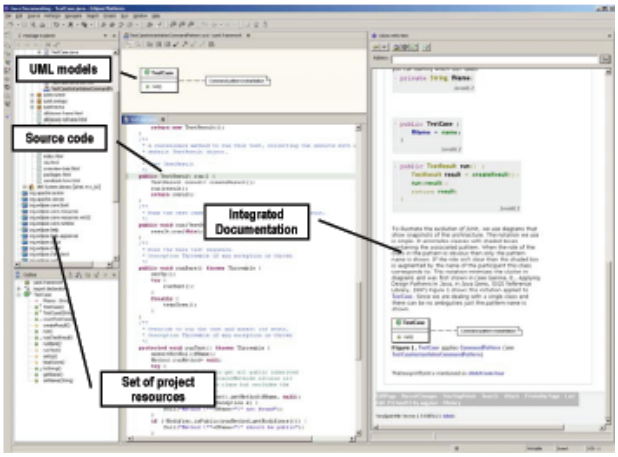

Figure 9: A snapshot of XSDoc integrated in Eclipse

through the XSDoc Wiki, but they can also be exported to static HTML, for off-line browsing, or to PDF files, for high-quality printing. Source code is presented with syntax-highlighting, and smart hyperlinking to other source code files, formal documentation contained in Javadoc or Doxygen comments, and other related documents, thus providing a good navigability in the overall existing software artifacts (code, models and documents).

Any change on external contents, such as the referenced code or models, is automatically reflected in the documentation when the web page is refreshed by the browser, or when the involved contents are modified and saved.

5. CONCLUSIONS

In this paper, we have presented a wiki-based approach for quickly weaving heterogeneous software artifacts from multiple sources of contents, in concrete, from source code files, models, and documents. The dynamic nature of the approach ensures that the contents are always up-to-date and semantically consistent.

The approach is based on well-know software documentation techniques, namely literate programming and some of its most contemporary alternatives. It combines the simplicity, easiness and versatility of the collaborative document edition with wikis, the well-known qualities of XML technology in terms of information integration, processing and presentation, and the powerful development features of open IDEs.

From our experience with XSDoc Wiki and the earlier prototypes, we are convinced that the idea of wiki-based software documentation, combining wiki and XML technologies, is very attractive and enforces the idea that *documenting-while-developing* can be simplified if supported with appropriate tools, especially when they are integrated in an open IDE. XSDoc Wiki helps on stepping forward to the direction of *documentation-enabled development environments*, which eventually would change the attitude of developers on the usefulness of documenting.

The idea of wiki-based software documentation and XSDoc are at the moment of this writing under research in the context of the `DocIt!` project, at FEUP, where XSDoc is being improved with browsing features to help adapt documentation contents to the user needs (zoom, exploration mode, error recovery, extensive search), new plugins for integration with other popular IDEs (Microsoft VisualStudio), and integration in popular wikis (TWiki [27], MediaWiki [29], SnipSnap [17], EclipseWiki [32]).

In order to quantitatively and qualitatively evaluate the impact of these ideas and tools on the quality, understandability and usability of the resulting software documentation, user tests and experiments are also being carried on in different settings, ranging from the academia to industry.

6. REFERENCES

[1] A. Aguiar. *A minimalist approach to framework documentation*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, September 2003.

[2] A. Aguiar, G. David, and G. Badros. JavaML 2.0: enriching the markup language for Java source code. In *Proceedings of XATA 2003, XML: Aplicaes e Tecnologias Associadas*, February 2004. http://www.fe.up.pt/~aaguiar/javaml/.

[3] A. Aguiar, G. David, and M. Padilha. XSDoc: an Extensible Wiki-based Infrastructure for Framework Documentation. In E. Pimentel, N. R. Brisaboa, and J. Gómez, editors, *JISBD*, pages 11–24, 2003.

[4] K. M. Anderson, S. A. Sherba, and W. V. Lepthien. Towards large-scale information integration. In *Proceedings of the 24th international conference on Software engineering*, pages 524–534. ACM Press, 2002.

[5] G. J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):159–177, 2000.

[6] K. Beck and E. Gamma. Junit: A cook's tour, 2003. Available from http://www.junit.org.

[7] R. Bodner and M. Chignell. Dynamic hypertext: querying and linking. *ACM Comput. Surv.*, 31(4es):15, 1999.

[8] R. Bodner, M. Chignell, and J. Tam. Website authoring using dynamic hypertext. In *Proceedings of Webnet'97, Toronto: Association for the Advancement of Computing in Education*, pages 59–64, 1999.

[9] A. B. Coates and Z. Rendon. xmLP a Literate Programming Tool for XML & Text, 2002. http://xmlp.sourceforge.net/.

[10] M. L. Collard, J. I. Maletic, and A. Marcus. Supporting Document and Data Views of Source Code. In *Proceedings of DocEng'02, McLean, Virginia USA*, November 2002.

[11] G. Cronin and B. Barnett. Very quick wiki homepage, 2003. Available from http://veryquickwiki.sourceforge.net/.

[12] W. Cunningham. Portland pattern repository., 1999. Available from http://c2.com/cgi/wiki.

[13] Eclipse. Eclipse, an open and extensible integrated development environment, 2003. Available from http://www.eclipse.org.

[14] O. M. Group. XML Metadata Interchange (XMI), 2005. Available from http://www.omg.org/.

[15] S. C. Gupta, T. Nguyen, and E. V. Munson. The software concordance: A user interface for advanced software documents. In *Proceedings of 6th IASTED International Conference on Software Engineering and*

Applications, MIT, Cambridge, MA, USA, November 2002.

[16] J. Hartmann, S. Huang, and S. Tilley. Documenting software systems with views II: an integrated approach based on XML. In *Proceedings of the 19th annual international conference on Computer documentation*, pages 237–246. ACM Press, 2001.

[17] M. L. Jugel and S. J. Schmidt. SnipSnap Wiki homepage, 2003. Available from http://www.snipsnap.org/.

[18] M. Knasmüller. Reverse Literate Programming. In *Proceedings of the Software Quality Conference, Dundee*, 1996.

[19] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[20] D. Kramer. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153. ACM Press, 1999.

[21] E. Mamas and K. Kontogiannis. Towards Portable Source Code Representations Using XML. In *Proceedings of WCRE'00, Brisbane Australia*, pages 172–182, November 2000.

[22] K. Nørmark. An elucidative programming environment for Scheme. In *Proceedings of NWPER'2000 - Nordic Workshop on Programming Environment Research*, pages 109–126, May 2000.

[23] K. Nørmark, M. Andersen, C. Christensen, V. Kumar, S. Staun-Pedersen, and K. Sørensen. Elucidative programming in Java. In *Proceedings on the eighteenth annual international conference on Computer documentation (SIGDOC)*, pages 483–495. IEEE Educational Activities Department, September 2000.

[24] H. P. Report. Towards modern literate programming.

[25] J. Sametinger and G. Pomberger. A hypertext system for literate C++ programming. *Journal of Object Oriented Programming*, 4(8):24–29, 1992.

[26] Sun Microsystems. Javadoc Tool Home Page, 2003. http://java.sun.com/j2se/javadoc/.

[27] P. Thoeny. Twiki homepage, 1998. Available from http://www.twiki.org/.

[28] D. van Heesch. Doxygen — a documentation system for C++, Java and other languages, 2002. Available from http://www.doxygen.org.

[29] B. Vibber. Mediawiki homepage, 2001. Available from http://www.mediawiki.org/.

[30] F. Vitali and M. Bieber. Hypermedia on the web: what will it take? *ACM Comput. Surv.*, 31(4es):31, 1999.

[31] N. Walsh. Literate Programming in XML, Oct. 2002. http://nwalsh.com/docs/articles/xml2002/.

[32] C. Walton. EclipseWiki homepage, 2003. Available from http://eclipsewiki.sourceforge.net/.

[33] World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0, November 1999. Available from http://www.w3.org/TR/xslt.

[34] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model, November 2002. Available from http://www.w3.org/TR/2002/WD-query-datamodel-20021115.